IBM Red Brick Warehouse

Administrator's Guide

Version 6.3

SC18-7395-00

IBM Red Brick Warehouse

Administrator's Guide

Version 6.3

SC18-7395-00

Note

Before using this information and the product it supports, read the information in "Notices" on page 443.

First Edition (March 2004)

This document contains proprietary information of IBM. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this manual should not be interpreted as such.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1996, 2004. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this book xiii
Who should read this book xiii
Software dependencies xiii
Conventions and terminology used in this book xiv
Typographical conventions. xiv
Syntax diagrams xiv
Keywords and punctuation xvii
Identifiers and names xvii
Comments xvii
Platform markup xviii
Software support xviii
Report a problem xviii
Existing cases xix
Troubleshooting tips xix
Related documentation xix
Additional documentation xxi
How to send your comments xxi

Chapter 1. Overview of IBM Red Brick Warehouse 1
Database server technology 1
Database server components. 2
IBM Red Brick Warehouse 3
Parallel Table Management Utility 4
RISQL Entry Tool and RISQL Reporter 4
Administrator tool 5
Client Connector Pack 5
Database server 5
Interprocess communication 6
Warehouse API process 7
Database server processes 8
Administration daemon process 8
Performance monitor daemon 8
Log daemon process 8
Process checker daemon 9
Vacuum cleaner daemon 9
Listener thread (Windows only) 9
CTRL-C Coordination thread (Windows Only) 10
Shared memory 10
Database administration overview 10
Installing Red Brick Warehouse 10
Planning the database design 11
Implementing the database 11
Providing user access. 12
Initialization files 12
Macros 12

© Copyright IBM Corp. 1996, 2004

Loading and unloading data . . . 13
Loading concurrently with queries . . . 13
Exporting query results . . . 13
Maintaining the database and tuning for performance . . . 13
Planning backup and restore procedures . . . 14
Aroma sample database . . . 14
Database limits . . . 14

Chapter 2. Key concepts . . . 17
Data loading . . . 17
Parallel processing . . . 18
Physical implementation of databases . . . 19
Indexes and retrieval strategies . . . 19
Segmented storage . . . 20
Named and default segments . . . 20
Implementation . . . 21
Distributing data among segments . . . 22
Online and offline segments . . . 24
Partial availability of tables and indexes . . . 25
Precomputed views for increased query performance . . . 25
Query sampling for application input . . . 26
Database directories and files . . . 26
Logical database names . . . 27
Segment names . . . 31
Configuration and initialization . . . 31
Configuration file . . . 31
Initialization files . . . 31
.rbwrc files . . . 31
.rbretrc files . . . 32
ODBC initialization files . . . 33
SET commands . . . 34
Environment variables . . . 34
Administrator tool . . . 34
Server locale . . . 35
Components of a locale . . . 36
Language . . . 36
Territory . . . 36
Code set . . . 37
Collation sequence . . . 37
Defining the server locale . . . 38
System table references to locales . . . 39
Nontranslated text . . . 39
Overriding the server locale . . . 39
Specifying locales for RISQL clients . . . 40
Setting the RB_NLS_LOCALE Environment Variable . . . 41
Ensuring client/server compatibility . . . 42
Code set conversions . . . 42
Message system . . . 43
File ownership and permissions . . . 44
Database authorizations and privileges . . . 44

Versioned databases 45
Referential integrity 46
Load and insert operations 46
Delete operations and cascaded deletes 46

Chapter 3. Schema design 55
Transaction processing versus decision support 55
Transaction-processing databases 55
Decision-support databases 56
Star schemas 57
Performance of star schemas 58
Terminology 58
Simple star schemas 58
Multiple fact tables 60
Multicolumn foreign key 62
Outboard tables 63
Multistar schemas 63
Views 65
Considerations for schema design 65
Schema building blocks 67
Example: Salad dressing database 70
Analyzing your schema 71
Browsing the dimension tables 71
Querying the fact table 72
Determining which attributes to include 72
Schema examples 74
Reservation system database 74
Investment database 75
Health insurance database 76

Chapter 4. Planning a database implementation 79
Organizing data into databases 79
Determining when to create additional indexes 79
STAR indexes 81
Order of columns 82
Queries against multiple fact tables 83
B-TREE indexes 84
TARGET indexes 85
Understanding weakly selective constraints 85
Choosing the right domain size 85
Locally segmented indexes 87
No indexes 87
Planning for segmented storage 88
Determining when to use default and named segments 88
Using named segments 89
Deciding how to segment 90
PSU size and growth 91
Planning for a time-cyclic database 92
Segment indexes like the fact table 93
Characteristics of a local index. 94

Small segment at the end of the table and each index . . . 94
Planning disk storage organization . . . 95
Estimating the size of user tables . . . 96
Estimating the size of indexes . . . 97
Index fill factors . . . 97
STAR indexes . . . 99
B-TREE indexes . . . 99
TARGET indexes . . . 100
Example: calculating table, index, and system table sizes . . . 101
Fact1 table and its indexes . . . 102
Market table and its indexes . . . 103
Product table and its indexes . . . 105
Estimating the size of system tables . . . 106
Size: system tables . . . 107
Total space for user tables, indexes, and system tables . . . 107
Estimating temporary space requirements . . . 108
How optimized index-building operations use temporary space . . . 109
How temporary space is allocated . . . 110
Random directory sequence . . . 110
File creation and use . . . 111
Full and out-of-space conditions . . . 112
Estimating temporary space values for index-building operations . . . 112
DIRECTORY location values . . . 113
THRESHOLD value . . . 114
MAXSPILLSIZE value . . . 115
Temporary space requirements for TARGET indexes . . . 118
Determining current values . . . 119
Removing temporary files . . . 119
How query operations use temporary space . . . 120
Estimating a QUERY_MEMORY_LIMIT value for queries . . . 120
Estimating a MAXSPILLSIZE value for queries . . . 121
Setting the result buffer for long-running queries . . . 121
RESULT BUFFER parameter . . . 122
RESULT BUFFER FULL ACTION parameter . . . 122
Considerations for growing tables . . . 123
Effect of table growth on STAR indexes . . . 123

Chapter 5. Creating a database . . . 125
Overview . . . 125
Creating the database structure . . . 126
Database creation syntax . . . 126
Initializing the database . . . 127
Defining a logical database name . . . 129
Changing the DBA account password . . . 130
Creating the database objects . . . 131
Creating segments . . . 132
Rules for segmenting . . . 133
How to segment . . . 134
Segmenting a STAR index . . . 135
Setting segment and partial availability behavior . . . 138

Location of default segments . . . 138
Segment drop behavior . . . 139
Creating tables . . . 140
Setting the MAXSEGMENTS and MAXROWS PER SEGMENT options . . . 140
Naming constraints for primary and foreign keys . . . 141
Maintaining referential integrity with ON DELETE . . . 141
Setting the VARCHAR column fill factor . . . 142
How the database server uses the VARCHAR fill factor . . . 142
Effect of fill factor on performance . . . 143
Monitoring accuracy of the VARCHAR fill factor . . . 146
Obtaining the current fill factor value . . . 147
Using CHECK TABLE with the VERBOSE option . . . 147
Modifying the VARCHAR fill factor . . . 148
Creating indexes . . . 149
INDEX TEMPSPACE parameters . . . 150
Parallel indexes . . . 150
Setting the index fill factor . . . 151
Finding the fill factor used for a specific index . . . 154
Deciding whether to change default fill factors . . . 154
Changing an index fill factor . . . 155
Creating a time-cyclic database . . . 155
Creating the data and index segments . . . 157
Creating the tables . . . 162
Creating the STAR index . . . 163
Creating the local TARGET indexes. . . . 165
The data. . . . 167
Creating views . . . 168
Creating and managing macros . . . 170
Guidelines for macro definitions. . . . 170
Availability and scope . . . 171

Chapter 6. Working with a versioned database . . . 175
Determining whether you need versioning . . . 175
Load window . . . 175
Increased recoverability . . . 176
Load with periodic commit . . . 176
Providing load recoverability . . . 177
Versioning with Vista . . . 177
Dimension table cleaning . . . 178
Costs of the version log . . . 178
Loading data into versioned databases . . . 179
Understanding the version log . . . 180
Structure of the version log . . . 181
Versioned DELETE operations . . . 183
Understanding frozen versions . . . 183
Loading data with the high recoverability option . . . 185
Controlling versioning . . . 186
Creating the version log . . . 187
Cleaning the version log . . . 188
Dropping the version log and adding space . . . 188

Controlling frozen versions . . . 189
Freezing the database . . . 189
Overriding frozen versions . . . 190
Unfreezing the database . . . 190
Unloading data with frozen versions . . . 190
Using frozen versions and high recoverability . . . 190
Maintaining a versioned database . . . 190
Monitoring the version log . . . 191
Backup and recovery . . . 191
Controlling the vacuum cleaner . . . 192
Example: Creating a versioned Aroma database . . . 193

Chapter 7. Providing database access and security . . . 195
Adding database users . . . 195
Creating operating-system accounts for users . . . 195
Granting database access . . . 197
Changing passwords . . . 198
Granting access with system roles . . . 198
DBA, RESOURCE, and CONNECT capabilities . . . 198
Granting and revoking the DBA and RESOURCE system roles . . . 199
Granting database object privileges . . . 200
Granting access with role-based security . . . 201
Task authorizations . . . 201
Role capabilities . . . 203
Creating roles . . . 204
Granting task authorizations . . . 205
Granting object privileges to roles . . . 205
Granting roles . . . 207
Revoking task authorizations, object privileges, and roles . . . 210
Tracking role authorizations and members . . . 212
Administering password security . . . 214
Enforcing password changes . . . 216
Warning users of password expiration . . . 217
Limiting reuse of previous passwords . . . 217
Limiting frequency of password changes . . . 218
Restricting use of login name in password . . . 219
Requiring user to change initial password at first login . . . 219
Enforcing differences from previous passwords . . . 219
Enforcing password complexity and length . . . 220
Locking user accounts after failed connection attempts . . . 222
Specifying the lock-out period . . . 223
Locked account status . . . 223

Chapter 8. Managing database activity in an enterprise . . . 225
Task authorizations for managing database activity . . . 226
Administration database . . . 226
Monitoring database activity with dynamic statistic tables . . . 227
Read and write statistics . . . 228
Definition of read statistics . . . 228
Definition of write statistics . . . 229

Platform dependency . . . 230
Controlling database activity . . . 230
Bringing a database to a quiescent state . . . 230
Activating a database . . . 231
Resetting accumulated statistics . . . 231
Canceling a user command . . . 231
Closing a user session . . . 231
Changing user priorities for the current session . . . 232
Administration daemon process . . . 232
Statistics collection interval . . . 233
DST refresh interval . . . 234
Event logging . . . 234
Logging subsystem . . . 235
Log daemon . . . 235
Log viewer . . . 236
Event log messages . . . 239
Event severity . . . 239
Event category . . . 240
Audit events . . . 240
Error events . . . 240
Operational events . . . 240
Schema events. . . . 241
Usage events . . . 241
Log files. . . . 241
Configuring the logging subsystem. . . . 242
Setting the startup state . . . 242
Specifying the location of log files . . . 242
Specifying the maximum log file size . . . 243
Setting the log severity filter level . . . 243
Controlling logging operations . . . 243
Starting logging . . . 244
Stopping logging . . . 244
Switching log files . . . 244
Changing log filter levels . . . 244
Terminating the log daemon . . . 244
Query logging. . . . 244
Controlling advisor logging . . . 245
Advisor log files . . . 245
What the Advisor logs . . . 246
Starting and stopping the Advisor log . . . 246
ADMIN ADVISOR_LOGGING . . . 246
ALTER SYSTEM . . . 247
ADMIN ADVISOR_LOG_DIRECTORY . . . 247
ADMIN ADVISOR_LOG_MAXSIZE . . . 247
Accounting. . . . 248
Accounting process . . . 249
Format of accounting records. . . . 249
Accounting files . . . 250
Configuring accounting . . . 251
Setting the startup state . . . 251

Specifying the location of accounting files . . . 252
Specifying the maximum accounting file size . . . 252
Setting the accounting mode . . . 253
Controlling accounting . . . 253
Starting accounting . . . 253
Stopping accounting . . . 253
Switching accounting files . . . 254
Changing accounting mode . . . 254

Chapter 9. Maintaining a data warehouse . . . 255
Locking tables and databases . . . 255
Manual table or database locks . . . 256
READER PRIORITY option . . . 256
Types of table locks . . . 257
Locking and segments . . . 258
Determining when to lock a table or database . . . 258
Specifying wait behavior for database server and TMU locks . . . 259
No-wait behavior . . . 259
Deadlocks . . . 260
Setting isolation level for versioned transactions . . . 260
Obtaining information on tables and indexes . . . 261
CHECK TABLE segment statistics . . . 262
Checking and repairing table structure . . . 264
CHECK INDEX size and configuration information . . . 264
Size and space information for B-TREE index . . . 265
Configuration information for B-TREE indexes . . . 267
Size and space information for STAR indexes . . . 268
Configuration information for STAR indexes . . . 269
CHECK INDEX output for TARGET index . . . 270
Size and space information for TARGET indexes . . . 272
Configuration information for TARGET indexes . . . 274
Checking for index corruption . . . 275
Monitoring growth of tables and indexes . . . 275
STAR indexes . . . 275
MAXSIZE column . . . 276
USED column . . . 277
TOTALFREE column . . . 278
Pseudocolumns . . . 278
Freespace field . . . 280
How space is allocated to a segment . . . 281
Dynamic adjustment of MAXSIZE . . . 282
Altering segments . . . 283
Using ALTER SEGMENT . . . 283
Ensuring no users are active . . . 284
Attaching and detaching segments . . . 285
Attaching a segment to a single-segment table or index . . . 286
Specifying a range . . . 288
Taking a segment offline or online . . . 288
Clearing a segment . . . 289
Renaming a segment . . . 289

Releasing and dropping storage from a segment . . . 289
Changing PSU sizes . . . 293
Moving entire segments . . . 293
Changing PSU location . . . 293
Verifying a segment . . . 294
Forcing a segment into an intact state . . . 294
Maintaining a time-cyclic database . . . 294
Rolling off and reusing data and index segments . . . 295
Adding a new segment . . . 300
Deleting the oldest data . . . 303
Recovering a damaged segment . . . 305
Managing optical storage . . . 307
Assigning optical storage . . . 308
Specifying access behavior for optical segments . . . 308
Specifying index selection with optical segments . . . 309
Altering tables . . . 310
Adding and dropping columns . . . 310
ALTER TABLE location . . . 311
ALTER TABLE space requirements . . . 311
Changing a column name . . . 313
Changing the default value for a column . . . 313
Changing the MAXSEGMENTS and MAXROWS PER SEGMENTS values . . . 313
Changing the way referential integrity is maintained . . . 314
Changing the data type for a column . . . 314
Adding and dropping foreign keys . . . 315
Changing the fill factor for a VARCHAR column . . . 315
Changing the start and step values of a SERIAL column . . . 316
Recovering from an interrupted ALTER TABLE operation . . . 316
Recovering the table . . . 316
Interruptions: causes and prevention . . . 317
Copying or moving a database . . . 318
Compatibility . . . 318
Full versus relative pathnames . . . 319
Copying a database that contains only relative pathnames . . . 320
Copying a database that contains full pathnames . . . 320
Moving a database that contains only relative pathnames . . . 321
Moving a database that contains full pathnames . . . 321
Specifying parameters with rbw.config file entries or SET commands . . . 322
Modifying the configuration file . . . 322
Monitoring and controlling a database server . . . 330
Monitoring and controlling a database server on UNIX . . . 330
Daemon processes . . . 330
Log files . . . 331
Monitoring and controlling a database server on Windows . . . 332
Database service threads . . . 332
Log files . . . 333
Determining version information . . . 333
Deleting database objects and databases . . . 334
Dropping database objects . . . 334
Indexes . . . 334

Hierarchies . . . 335
Macros . . . 335
Roles . . . 335
Segments . . . 335
Synonyms . . . 336
Tables . . . 336
Views . . . 336
Deleting a database . . . 336
Managing memory requirements for a database catalog . . . 339
Operational considerations for compacting a system catalog . . . 339
Removing empty blocks from the system catalog . . . 340
Recovery considerations for system catalog compaction . . . 342
rb_syscompact syntax . . . 342

Appendix A. Example: Building a database . . . 343

Appendix B. Configuration file . . . 365

Appendix C. System tables and dynamic statistic tables . . . 405

Appendix D. Configuring federated access to Red Brick data sources . . . 435
Federated data access on Windows platforms . . . 435
To configure federated access to a Red Brick data source: . . . 435
Federated data access on UNIX and Linux platforms . . . 438
To set up the DB2 environment: . . . 438
Additional setup required on AIX, HP-UX, and Linux platforms . . . 439
To configure federated access to a Red Brick data source: . . . 439
Example of an .odbc.ini file . . . 442

Notices . . . 443
Trademarks . . . 446

Index . . . 447

About this book

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

This guide contains important information on administering IBM® Red Brick Warehouse™ and provides information about how to plan, configure, and use the database server. It describes features, database server concepts, and procedures for performing database server management and database maintenance tasks.

Who should read this book

You should read this guide if you are one of the following users:

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

You should 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

You should be using IBM Red Brick Warehouse, Version 6.3, as your database server.

Red Brick 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.

© Copyright IBM Corp. 1996, 2004

For information about how to create and populate the demonstration database, see Appendix A, "Example: Building a database," on page 343. 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®, where `redbrick_dir` is the Red Brick directory on your system.

Conventions and terminology used in this book

This guide uses typographical and terminology conventions to help you gather information from this and other volumes in the documentation set.

Typographical conventions

The following conventions introduce new terms, illustrate screen displays, describe command syntax, and so forth.

Convention	Meaning
KEYWORDS	All elements in a programming language statement (keywords) appear in uppercase letters in a serif font.
italics	Within text, new terms and emphasized words appear in italics.
`italics`	Within syntax and code examples, variable values that you are to specify appear in italics.
boldface	Interface elements (such as icons, menu items, and buttons) as well as commands appear in boldface.
`monospace`	Names of program entities (such as classes, and events), environment variables, directories, file names, and pathnames appear in monospace.
>	This symbol indicates a menu item. For example, "Choose **Tools > Options**" means choose the **Options** item from the **Tools** menu.

Tip: Press Enter immediately after your are instructed to enter characters or to execute a command. When you are instructed to type the text or to press other keys, no Return is required.

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.

Throughout this book, syntax is described using the structure defined as follows:

Read the syntax diagrams from left to right and top to bottom, following the path of the line.

The ▸▸— symbol indicates the beginning of a syntax diagram.

The —▸ symbol indicates that the syntax is continued on the next line.

The ▸— symbol indicates that the syntax is continued from the previous line.

The —▸◂ symbol indicates the end of a syntax diagram.

Syntax fragments start with the |— symbol and end with the —| symbol.

Required items appear on the horizontal line (the main path).

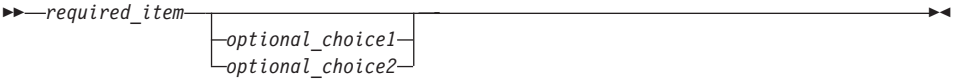
Optional items appear below the main path.

If an optional item appears above the main path, that item has no effect on execution, and is used only for readability.

If you can choose from two or more items, they appear in a stack.

If you *must* choose one of the items, one item of the stack appears on the main path.

If choosing one of the items is optional, the entire stack appears below the main path.



If one of the items is the default, it will appear above the main path, and the remaining choices will be shown below.

An arrow returning to the left, above the main line, indicates an item that can be repeated. In this case, repeated items must be separated by one or more blanks.

If the repeat arrow contains a comma, you must separate repeated items with a comma.

A repeat arrow above a stack indicates that you can make more than one choice from the stacked items or repeat a single choice.

Keywords appear in uppercase (for example, FROM). They must be spelled exactly as shown. Variables appear in lowercase (for example, column-name). They represent user-supplied names or values in the syntax.

If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Sometimes a single variable represents a larger fragment of the syntax. For example, in the following diagram, the variable parameter-block represents the whole syntax fragment that is labeled **parameter-block**:

parameter-block:

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.

Comments

Comments identify three types of information, as the following table describes. This information always appears in bold.

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 markup

Feature, product, and platform icons identify paragraphs that contain platform-specific information. Some examples are:

UNIX Only

Identifies information that is specific to the UNIX and Linux operating systems.

End of UNIX Only

Windows Only

Identifies information that is specific to Windows platforms.

End of Windows Only

Software support

If you have technical questions about IBM Red Brick Warehouse but cannot find the answer in the appropriate document, contact IBM Software 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
www-3.ibm.com/software/data/informix/support/

Report a problem

To report a new problem, you must provide the following information:

- IBM Red Brick Warehouse version
- Platform and operating-system version
- Error messages returned by IBM Red Brick Warehouse or the operating system
- Concise description of the problem, including any commands or operations performed before you received the error message
- List of IBM Red Brick Warehouse 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 the Red Brick ODBC Driver or 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 IBM Red Brick Warehouse includes the following documents.

Document	Description
Administrator's Guide	Describes warehouse architecture, supported schemas, and other concepts relevant to databases. Describes the Administrator tool, which provides graphical database administration. Includes procedural information for implementing and maintaining a database. Includes a description of the system tables and the configuration file.
Client Installation and Connectivity Guide	Includes procedures for installing ODBC, Red Brick JDBC Driver, RISQL Entry Tool, and RISQL Reporter on client systems. Describes how to access IBM Red Brick Warehouse using ODBC for C and C++ applications and JDBC for Java™ applications.

Document	Description
IBM Red Brick Vista User's Guide	Describes the IBM Red Brick Vista aggregate computation and management system. Illustrates how Vista improves query performance by automatically rewriting queries to use aggregates, describes how the Advisor recommends the best set of aggregates 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 IBM Red Brick Warehouse. Customized for either UNIX and Linux platforms or Windows platforms.
Locale Support	Identifies the locale components (languages, territories, code sets, and sort orders) supported by IBM Red Brick Warehouse products.
Messages and Codes Reference Guide	Contains a complete listing of all informational, warning, and error messages generated by IBM Red Brick Warehouse products, including probable causes and recommended responses. Also includes event log messages that are written to the log files.
Query Performance Guide	This guide describes the determinants of query performance and shows how to tune a database for optimal query performance. Examples show how to evaluate query performance using Red Brick tools: SET STATS, Dynamic Statistic Tables, EXPLAIN, and the Query Performance Monitor.
Release Notes	Contains information pertinent to the current release that was unavailable when the documents were printed.
RISQL Entry Tool and RISQL Reporter User's Guide	Is a complete guide to the RISQL Entry Tool, a command-line tool used to enter SQL statements, and the RISQL Reporter, an enhanced version of the RISQL Entry Tool with report-formatting capabilities.
SQL Reference Guide	Is a complete language reference for the Red Brick SQL implementation and RISQL extensions for IBM Red Brick Warehouse 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 activities related to loading, maintaining, and backing up data. Also includes information about data replication and the rb_cm copy management utility.

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:

Documentation	Description
Online documents	A documentation CD that contains IBM Red Brick Warehouse documents in electronic format is provided with your Red Brick products. You can copy the documentation to your computer or access it directly from the CD. You can also view documentation online by going to: www-3.ibm.com/software/data/informix/pubs/
Printed documents	To order printed documents, contact your sales representative.
Online help	IBM 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.

How to send your comments

To help us with future versions of our documents, let us know about any corrections or clarifications that you would find useful. Include the following information:

- The full name and version of your document
- Any comments that you have about the document
- Your name, address, and phone number

Send electronic mail to us at the following address:

comments@vnet.ibm.com

This address is reserved for reporting errors and omissions in our documentation. If you prefer, you can fill out a comment form by going to:

www.ibm.com/software/data/rcf/

For immediate help with a technical problem, contact IBM Software Support.

Chapter 1. Overview of IBM Red Brick Warehouse

IBM Red Brick Warehouse is a relational database management system (RDBMS) designed for data warehouse, data mart, and online analytical processing (OLAP) applications. Red Brick Warehouse 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 Warehouse 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. Red Brick Warehouse supports two sets of analytic functions: the original RISQL *display functions* developed exclusively for the Red Brick server and a set of *SQL OLAP functions*, which adhere to and extend the ANSI SQL-99 standard. 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 Warehouse 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 Warehouse meets these requirements through the following features:

© Copyright IBM Corp. 1996, 2004

- 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 multiprocessor (SMP) systems. Parallel processing can be used in query processing, multiuser relation scans, index building, and data loading.
- Two sets of analytic functions: the original RISQL *display functions* developed exclusively for the Red Brick server and a set of *SQL OLAP functions*, which adhere to and extend the ANSI SQL-99 standard. The Red Brick server supports all of the functions defined by the standard's "Elementary OLAP" package.

  For more information, see 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, see the *Table Management Utility Reference Guide*.
- 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, see the *IBM Red Brick 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:

- IBM Red Brick Warehouse
- Parallel Table Management Utility (Parallel TMU)
- RISQL Entry Tool, a command-line entry tool, and RISQL Reporter, a report generator
- Administrator tool

- Vista, an aggregate computation and management system. All Vista features are fully integrated into the database server, the Parallel TMU, and the Administrator tool.
- IBM 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 IBM Red Brick Warehouse.

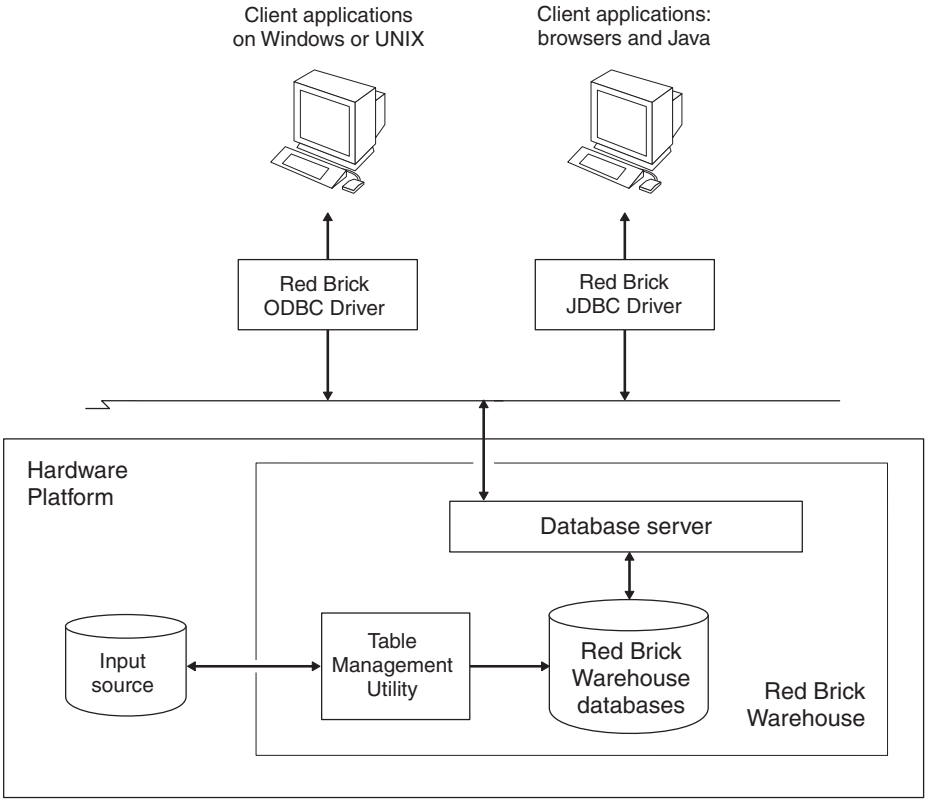


Figure 1. IBM Red Brick Warehouse components

IBM Red Brick Warehouse

IBM Red Brick Warehouse 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, IBM Red Brick Warehouse runs as a Windows service. The database server functions are performed by the processes described in "Database server" on page 5.

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. The TMU program can be invoked remotely, allowing DBAs to load data from an input file on a networked client machine into a database table on the production server machine. 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 IBM Red Brick Warehouse.
- 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, see Appendix A, "Example: Building a database," on page 343, 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 IBM Red Brick Warehouse. 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 Windows client connected to IBM Red Brick Warehouse. For more information about these tools, see 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, see "Administrator tool" on page 34.

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 IBM Red Brick Warehouse. This API allows you to use front-end applications, such as Microsoft Access, to access information in IBM Red Brick Warehouse.

The Red Brick JDBC Driver allows a wide variety of JDBC-compliant database applications to work with IBM Red Brick Warehouse. 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, see the *Client Installation and Connectivity Guide*.

Database server

UNIX Only

IBM Red Brick Warehouse 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, see "Monitoring and controlling a database server on UNIX" on page 330.

Windows Only

IBM Red Brick Warehouse is implemented on Windows by a single multithreaded 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 IBM Red Brick Warehouse service.

IBM Red Brick Warehouse 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, see "Monitoring and controlling a database server on Windows" on page 332.

End of Windows Only

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.

Windows Only

On Windows, the IBM Red Brick Warehouse service runs as a single multithreaded process. The threads run as processes with names corresponding to the processes on UNIX.

End of Windows Only

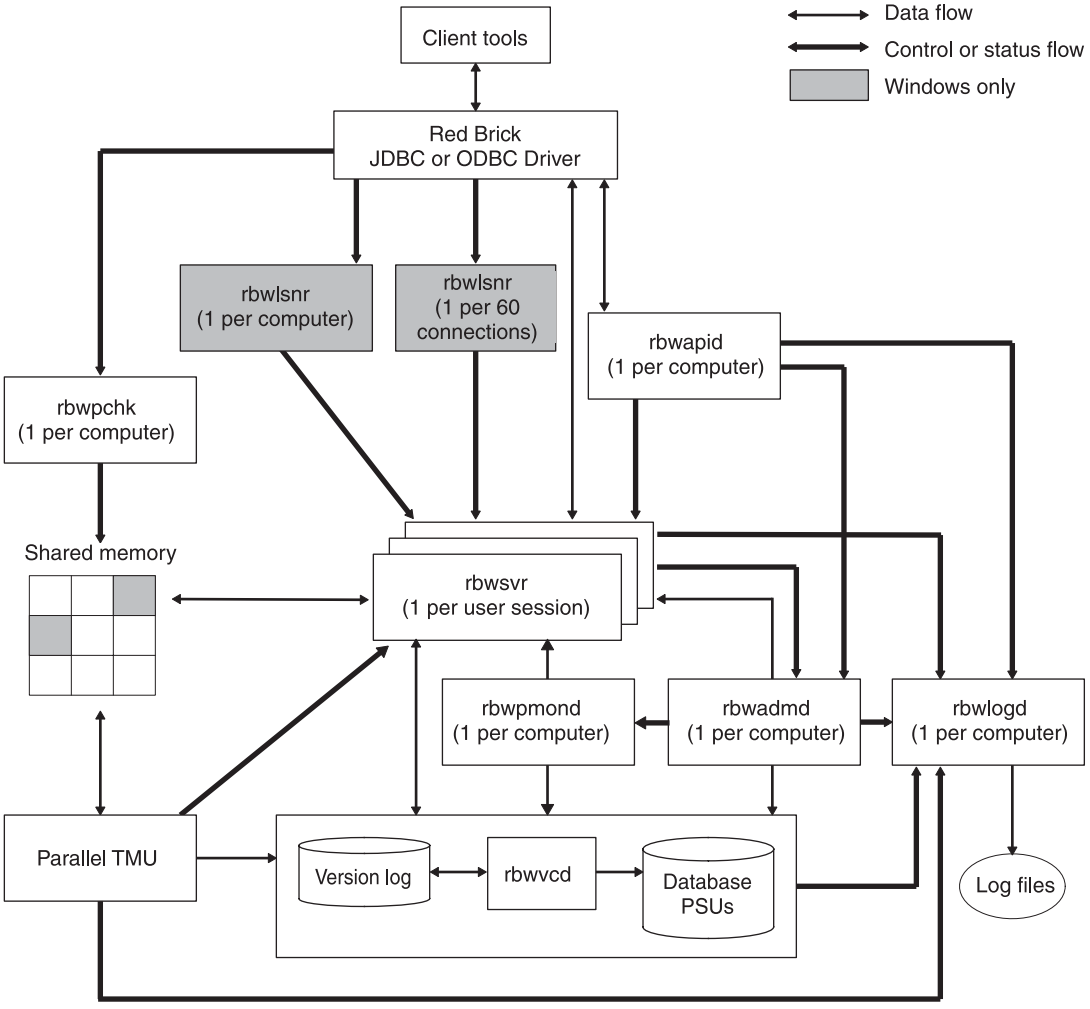


Figure 2. Interprocess Communication

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, see "Monitoring and controlling a database server" on page 330.

Database server processes

IBM Red Brick Warehouse 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 Warehouse 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, see the *IBM Red Brick 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, see "Administration daemon process" on page 232.

Performance monitor daemon

The performance monitor daemon process (rbwpmond) collects additional statistics for the performance DSTs when the database administrator wants to profile the execution of queries. The rbwadmd process is started by the database administrator with the ALTER SYSTEM START PERFORMANCE MONITOR statement.

For more information about using the performance monitor, see the *Query Performance Guide*.

Log daemon process

The log daemon process (rbwlogd) writes records to the log file when various events occur in Red Brick Warehouse. 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 IBM Customer Support Center diagnose problems.

For more information about the rbwlogd process, see "Log daemon" on page 235.

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 Warehouse (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, see Chapter 6, "Working with a versioned database," on page 175.

Listener thread (Windows only)

The listener thread (rbwlsnr) is a thread in the Red Brick Warehouse 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.

CTRL-C Coordination thread (Windows Only)

The CTRL-C coordination thread (rbwconc) listens for Ctrl+C interrupts from the various clients connected to Red Brick Warehouse. 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 Warehouse 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 Warehouse

IBM Red Brick Warehouse software is installed and verified from a CD-ROM device using a menu-driven script provided by IBM. 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 IBM Red Brick Warehouse publications as the redbrick user. The installation process ensures correct file access for all users.

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

Planning the database design

Red Brick Warehouse 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, see Chapter 2, "Key concepts," on page 17. For information about schema design, see Chapter 3, "Schema design," on page 55. For information about index strategies, physical storage, and size estimation, see Chapter 4, "Planning a database implementation," on page 79.

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, see Appendix C, "System tables and dynamic statistic tables," on page 405. For information about the RISQL Entry Tool, see the *RISQL Entry Tool and RISQL Reporter User's Guide*.

For information about using the rb_creator or dbcreate utility, see "Creating the database structure" on page 126. For a complete description of the CREATE and GRANT statements, see 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, see "Deleting a database" on page 336.

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, see Chapter 7, "Providing database access and security," on page 195. For information about managing your database, see Chapter 8, "Managing database activity in an enterprise," on page 225.

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 see "Initialization files" on page 31.

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, see "Creating and managing macros" on page 170 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, see the *IBM Red Brick Vista User's Guide*.

For information about the TMU and loading or unloading tables, see Appendix A, "Example: Building a database," on page 343, 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," on page 175.

Exporting query results

To export query results efficiently to specified files, use the EXPORT feature. For information about the EXPORT statement, see the *SQL Reference Guide*. For information about task authorization for this feature, see "Task authorizations" on page 201.

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. The Query Performance Monitor can help improve performance of specific queries.
- 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, see Chapter 9, "Maintaining a data warehouse," on page 255. For more information about managing

databases in an enterprise, see Chapter 8, "Managing database activity in an enterprise," on page 225. For information about performance tuning and the Query Performance Monitor, see the *Query Performance Guide*.

Planning backup and restore procedures

Most databases are routinely modified by table loads and server-based DDL and DML operations. Such databases should be backed up on a regular schedule in case of system or software failure. In the event of a failure, the presence of a recent backup makes it possible to fully recover the database. The larger the database, the more important it is to back it up. Before deciding when and how often to back up a database, you must understand the types of Table Management Utility (TMU) backups you can perform and anticipate the time and effort involved in each case. As your database evolves, you must consider the amount of data at risk and be prepared to restore the database if necessary.

The TMU supports both online and checkpoint backups, as well as partial and full restore operations. For details, see the *Table Management Utility Reference Guide*.

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," on page 343.

Database limits

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

- A database can contain a maximum of 32,764 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 248 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.

Important: Do not use these figures to estimate database file sizes. For information on calculating space requirements for tables and indexes, see Chapter 4, "Planning a database implementation," on page 79.

Chapter 2. Key concepts

To perform database administration functions effectively, you should understand the key concepts of IBM Red Brick Warehouse. 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 IBM Red Brick Warehouse database is as fast as or faster than loads for other relational databases.

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

© Copyright IBM Corp. 1996, 2004

- A pipe from another system program

UNIX Only

- TAR format or ANSI standard label tapes

End of UNIX Only

In addition, the TMU program can be invoked remotely, allowing DBAs to load data from an input file on a networked client machine into a database table on the production server machine.

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, see "Creating indexes" on page 149.

Parallel processing

IBM Red Brick Warehouse 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, IBM Red Brick Warehouse 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 IBM Red Brick Warehouse dimensional segmentation, parallel processes can perform disk accesses simultaneously, improving performance significantly.

For more information on parallel query processing, see the *Query Performance Guide*.

UNIX Only

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.

End of UNIX Only

Physical implementation of databases

IBM Red Brick Warehouse 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. IBM Red Brick Warehouse 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 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 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.

For large data warehouses, IBM recommends that you use named segments because they offer extensive control over disk space allocation, file size and placement, and database growth even though they require more effort to plan, create, and manage than default segments. Using named segments, a database administrator can partition a table horizontally, distributing the data across multiple segments.Control over the individual segments simplifies maintenance tasks and provides better access.

Important: You must use named segments to obtain parallel query processing.

Example:

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. For more information, see "Planning for a time-cyclic database" on page 92, "Creating a time-cyclic database" on page 155, and "Maintaining a time-cyclic database" on page 294.

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).

Default segments are convenient for small and test databases which can fit into the default size of two gigabytes. If circumstances change and you require additional control over the size and growth of tables in a default segment, you can alter the default segments to attach named segments. For the detailed procedure, see "Attaching a segment to a single-segment table or index" on page 286.

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.

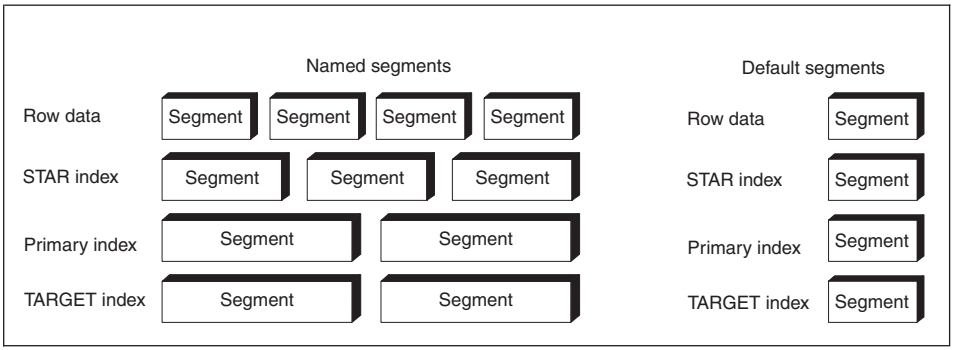


Figure 3. Use of Named and Default Segments

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.

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.

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.

Index entries in TARGET and B-TREE indexes can be distributed across segments in one of two ways:

- By ranges of values in the leading column of the index

  For a B-TREE, the leading column is the column named first in the CREATE INDEX statement. A primary index is a B-TREE index and can also be segmented by the key values of the first column of the index.

  For a TARGET index, the leading column is the only column specified in the CREATE TARGET INDEX statement.

  For information about the different SQL clauses to define this segmentation of the index, see "How to segment" on page 134.
- By the same column values used to segment the table

  In this case, the column does not need to be a column in the index. When an index is segmented this way, it is known as a *local index* where each index segment points to row data in a corresponding table segment. Only

B-TREE and TARGET indexes can be local indexes. A primary index cannot be a local index. For information about how to define a local index, see "How to segment" on page 134..

A local index facilitates the maintenance of TARGET and B-TREE indexes used in TARGETjoin processing of time-cyclic data. A local index can improve TARGETjoin performance but can affect performance of other query processing unless you tune parameters. For details, see the *Query Performance Guide*. For information on how a local index facilitates the maintenance of time-cyclic data, see "Characteristics of a local index" on page 94.

Index entries in a STAR index are not the values of the indexed data but the row numbers of the referenced table containing the data. STAR index entries can be distributed across segments in one of two ways:

- Evenly across the segments (default method)

  In this case, each segment contains approximately the same number of entries.
- By ranges of values in the leading column of the index when the leading column of the index is the segmenting column of the table

  The index entries in a STAR index are row numbers of the referenced column. For more information about the contents of a STAR index, see "Segmenting a STAR index" on page 135. For information about the different SQL clauses to define this segmentation of a STAR index, see Table 4 on page 134.

Examples:

Figure 4 illustrates correspondence between the row data segments and the index entry segments for a table with two indexes. This table has a STAR index and a TARGET index, which are both segmented by the same values as the data. The leading column in the STAR index is the same column used to segment the table. A local TARGET index does not need to contain the column used to segment the table.

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. This segmentation facilitates the time-cyclic example on page 21. A local index also facilitates maintenance of a time-cyclic database (for details, see "Characteristics of a local index" on page 94).

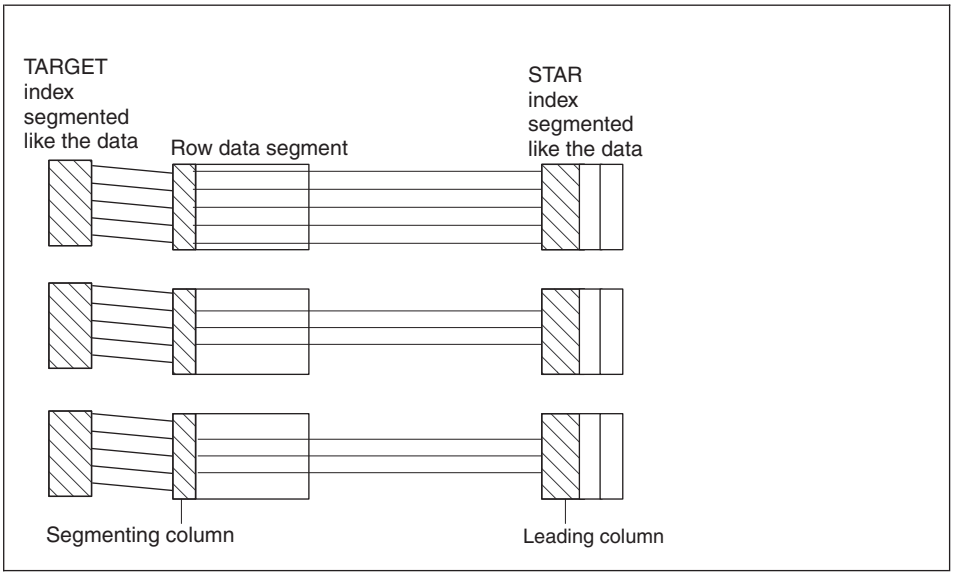


Figure 4. Row Data Segments and Indexes Segmented Like the Data

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. By definition, a default segment contains an entire table or index and cannot be taken offline. However, the administrator can attach named segments to a table or index that resides in a default segment after it has been altered to specify a segmenting column (for details, see "Attaching a segment to a single-segment table or index" on page 286). In this case, the default segment is one of multiple segments in the table, and it can 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, see the *Query Performance Guide*.

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, see the *IBM Red Brick Vista User's Guide*.

Query sampling for application input

IBM Red Brick Warehouse 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. IBM recommends the use of the default ADVISOR_SAMPLE_SIZE parameter to optimize candidate analyses, as described in the *IBM Red Brick 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 segment directories and files is controlled by IBM Red Brick Warehouse.

Logical database names

Users specify a logical database name for the database they want to access. IBM Red Brick Warehouse 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.

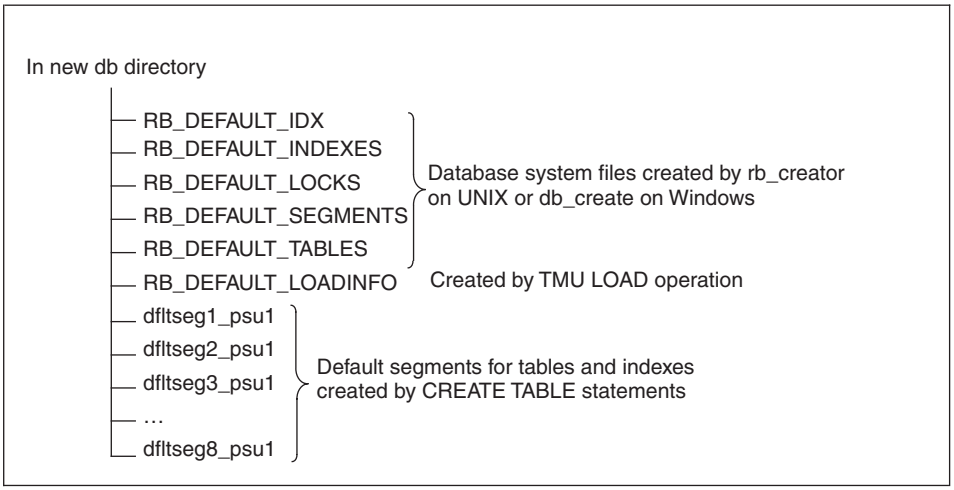


Figure 5. Directory of Logical Database Names

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.

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 Only

```
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.

End of UNIX Only

Windows Only

```
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)
   PRIMARY 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.

End of Windows Only

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

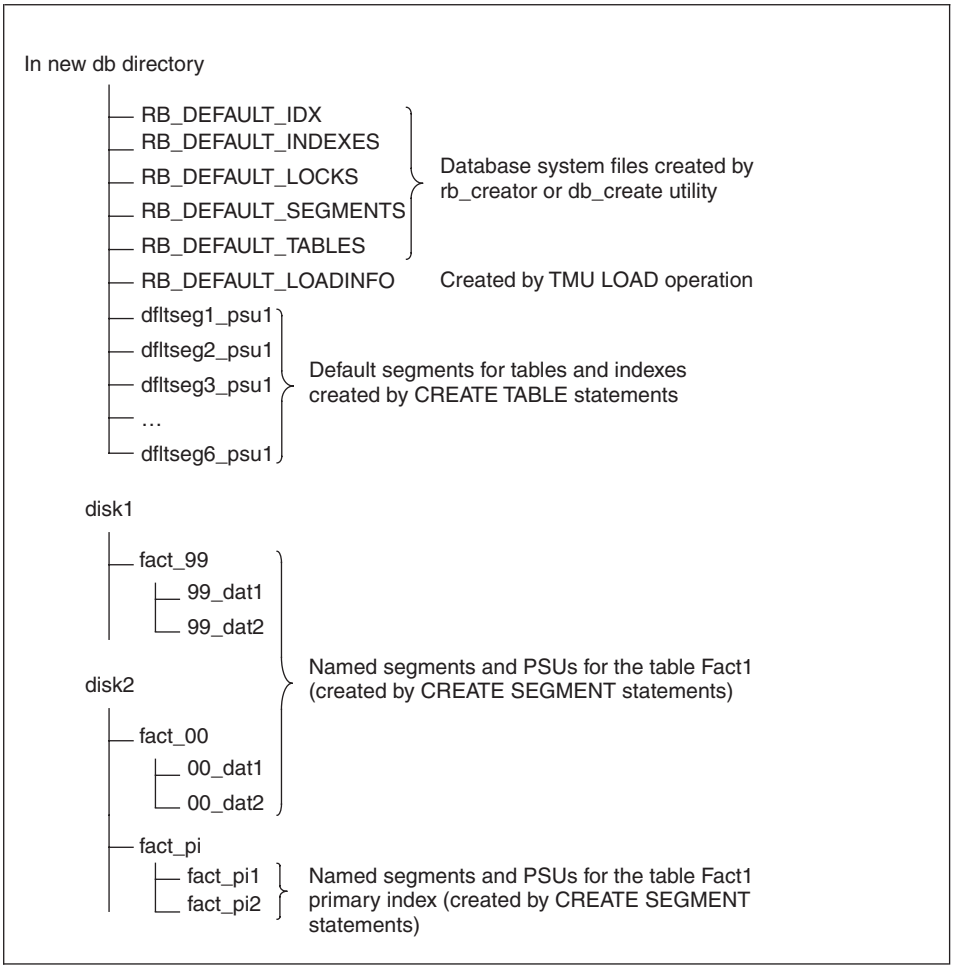


Figure 6. Example of Segmentation Schema of Database System Files

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_PI               FACT1      FACT1_PK_IDX
```

Configuration and initialization

IBM Red Brick Warehouse 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 IBM Red Brick Warehouse service and TMU on Windows. It can be edited with a text editor.

Initialization files

IBM Red Brick Warehouse 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%\ .rbwrc	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 IBM Red Brick Warehouse.
$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	%HOMEDRIVE% %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 IBM Red Brick Warehouse 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 Only

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.

End of UNIX Only

Windows Only

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.

End of Windows Only

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, see the *RISQL Entry Tool and RISQL Reporter User's Guide* and the *Table Management Utility Reference 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 IBM Red Brick Warehouse is installed.
RB_HOST	Logical name used to identify the warehouse daemon process on UNIX or the IBM Red Brick Warehouse 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 IBM Red Brick Warehouse 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 IBM Red Brick Warehouse product CD-ROM. For detailed information about specifying a server locale, see "Defining the server locale" on page 38.

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, IBM Red Brick Warehouse 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, see the `locales.pdf` file in the `relnotes` directory on your IBM Red Brick Warehouse product CD.

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. IBM Red Brick Warehouse supports both types of comparisons, as defined in the locale specification during installation.

Defining the server locale

During the installation of the IBM Red Brick Warehouse 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 IBM Red Brick Warehouse 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 IBM Red Brick Warehouse, 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, see 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 IBM Red Brick Warehouse 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, see the `locales.pdf` file in the `relnotes` directory on your IBM Red Brick Warehouse product CD.

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, see the *Table Management Utility Reference Guide*.

Specifying locales for RISQL clients

This section applies to the Red Brick client tools, the RISQL Entry Tool (risql) and the RISQL Reporter (risqlrpt). These clients can run in their own locale rather than the database locale. The client locale environment is defined by the locale definition itself (country, territory, and so on) and the path to the `.nls` file that defines each code set.

If a client locale is specified, it is used to format output based on the appropriate code set 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 the database 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.

To set the operating locale for RISQL clients, use the RB_NLS_LOCALE environment variable. If no client locale is specified, the behavior depends on the operating system.

UNIX Only

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

End of UNIX Only

Windows Only

On Windows, the client locale is specified in the `rbodbc32.ini` file in the Windows operating system directory. See “ODBC initialization files” on page 33

33.

End of Windows Only

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 Only

For example, from the UNIX C shell prompt:

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

End of UNIX Only

Windows Only

For example, from the MS-DOS shell prompt:

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

End of Windows Only

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, see the `locales.pdf` file in the `relnotes` directory on your IBM Red Brick Warehouse product CD.

- 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

Globalization 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.

- 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: IBM Red Brick Warehouse 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: RB*LLTTT*.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, see Chapter 8, "Managing database activity in an enterprise," on page 225.

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 IBM Red Brick Warehouse 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. IBM Red Brick Warehouse 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, see Chapter 7, "Providing database access and security," on page 195, 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 IBM Red Brick Warehouse 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. IBM Red Brick Warehouse 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, see Chapter 6, "Working with a versioned database," on page 175.

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. IBM Red Brick Warehouse 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.

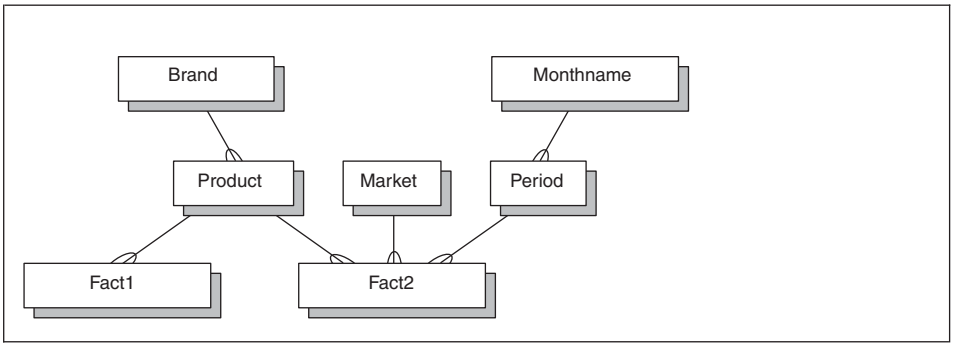


Figure 7. Schema Example

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
```

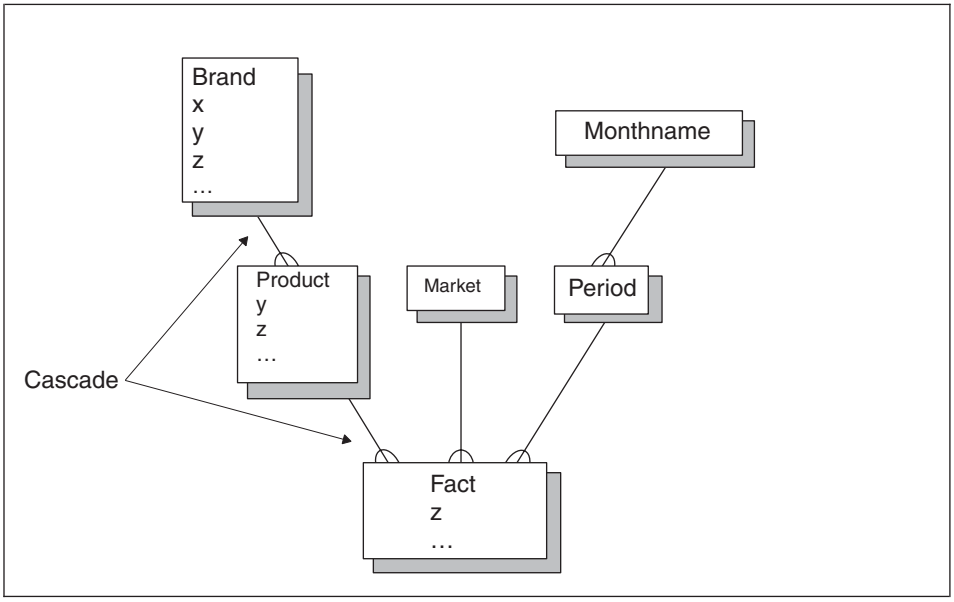


Figure 8. Delete Cascade

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.

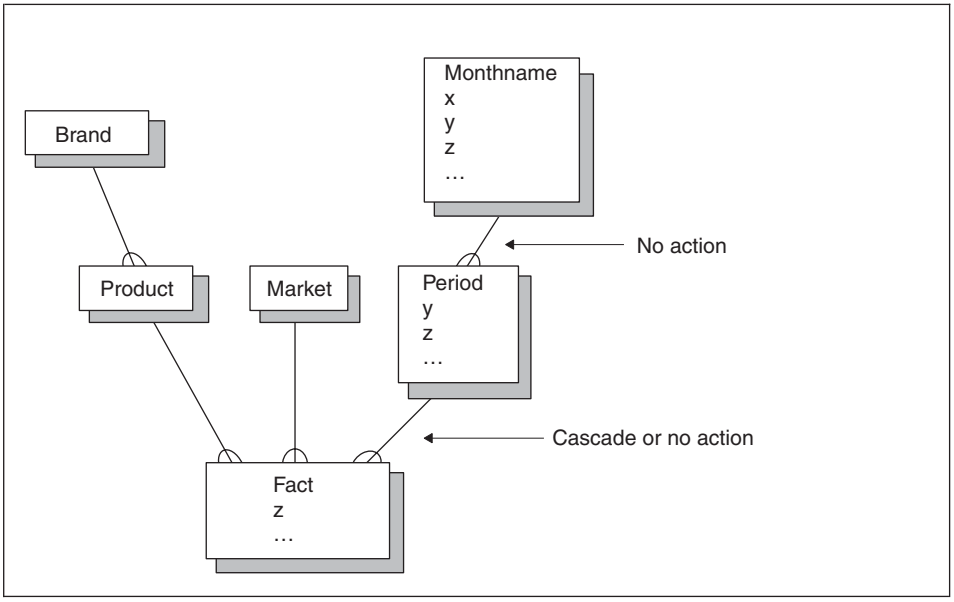


Figure 9. Delete No Action

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.

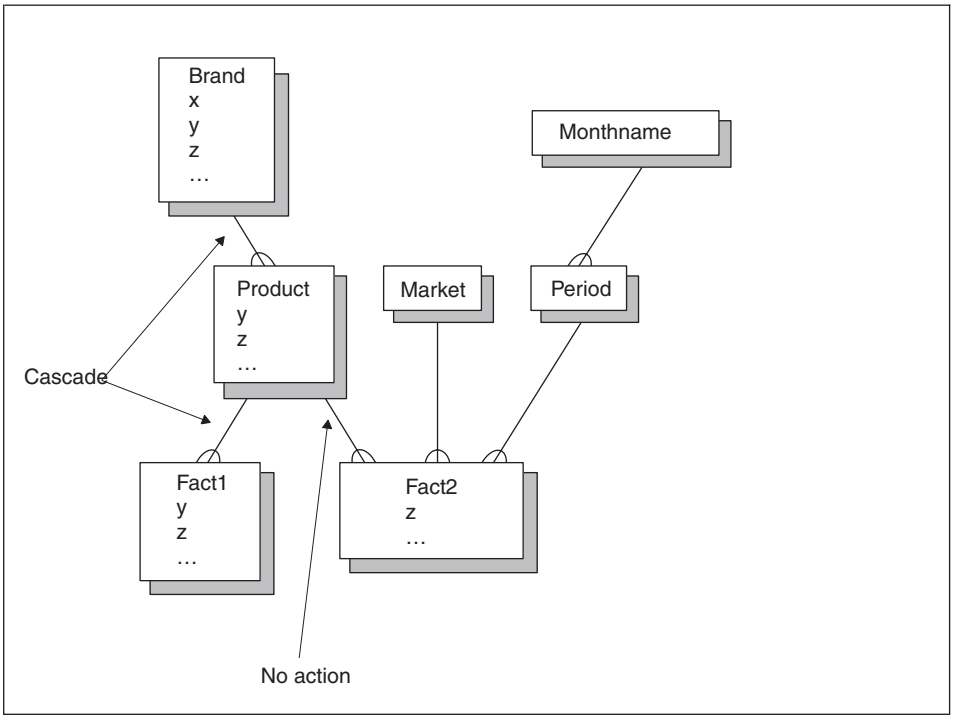


Figure 10. Restricted Delete

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, see the DELETE statement in the *SQL Reference Guide*.

Chapter 3. Schema design

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.

© Copyright IBM Corp. 1996, 2004

- 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.

IBM Red Brick Warehouse 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. IBM Red Brick Warehouse supports all schema designs. However, star schemas tend to perform the best in decision-support applications.

For more information on the performance implications of star schemas, see the *Query Performance Guide*.

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 IBM Red Brick Warehouse, 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.

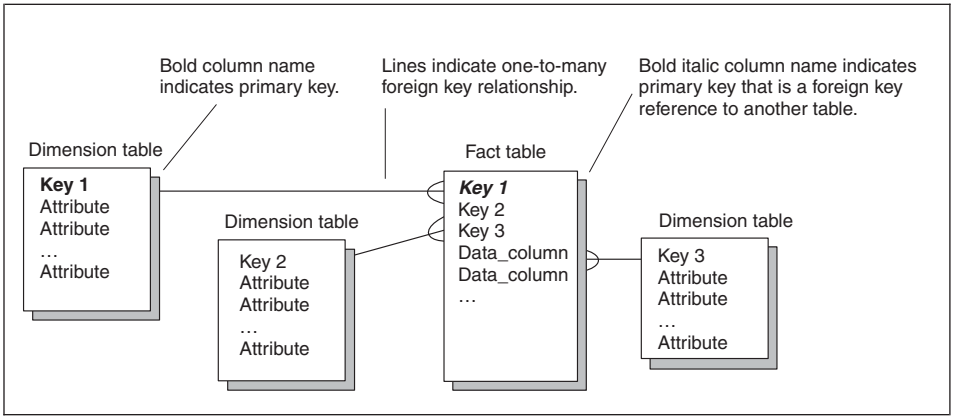


Figure 11. Simple Star Schema

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.

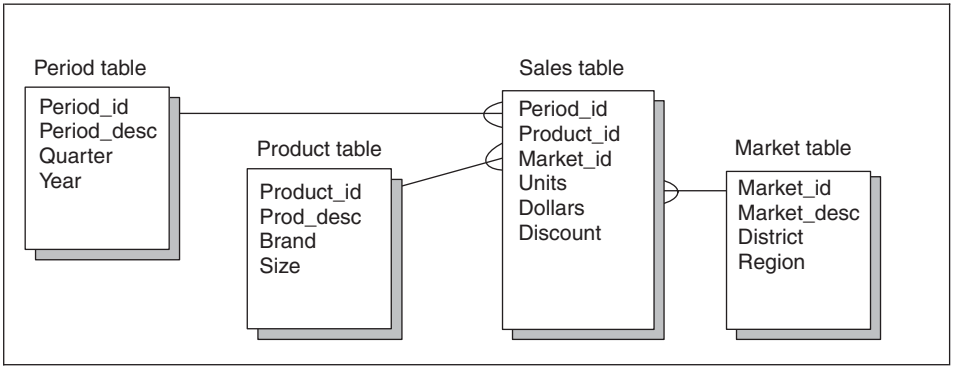


Figure 12. Sales Database

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.

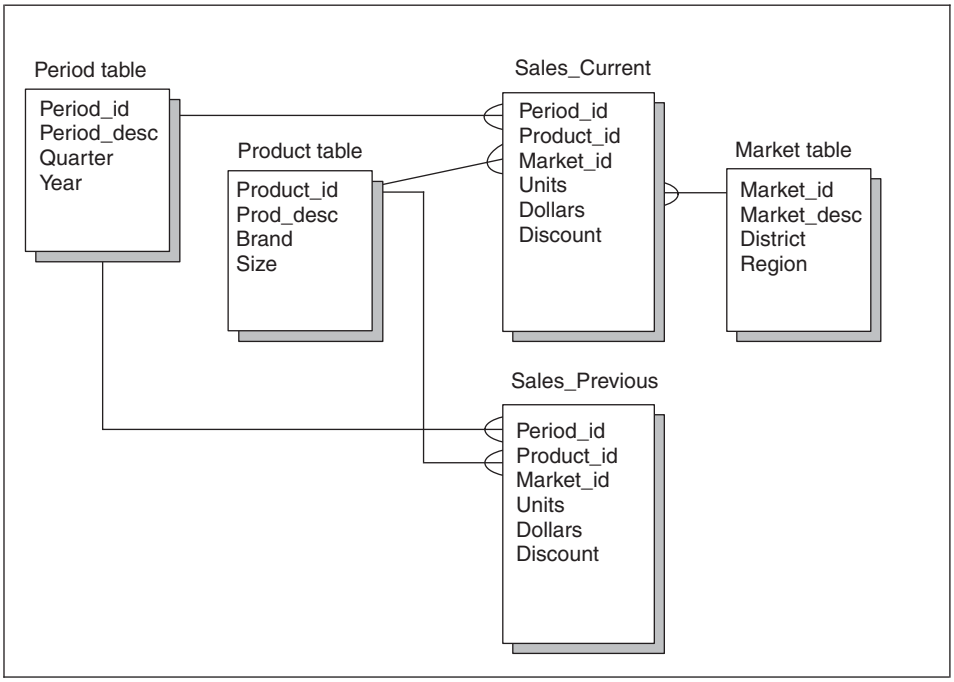


Figure 13. Sales database with additional dimension

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.

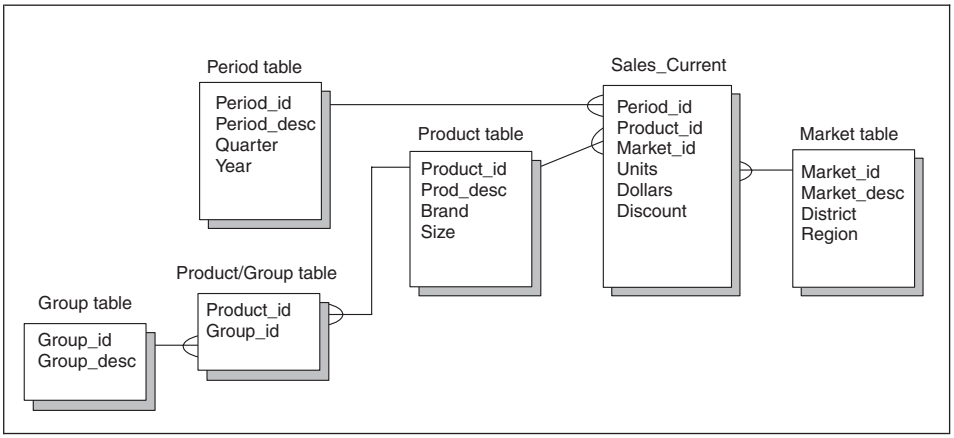


Figure 14. Sales database with cross-reference table

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.

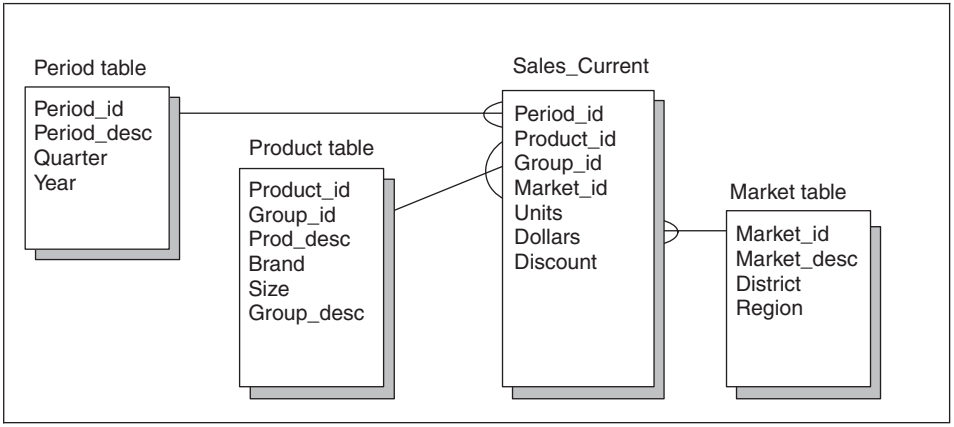


Figure 15. Sales database with multicolumn foreign key

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.

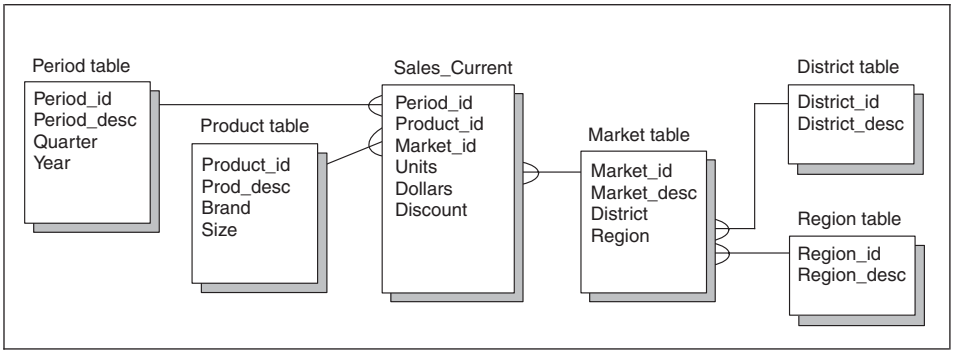


Figure 16. Sales database with outboard tables

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.

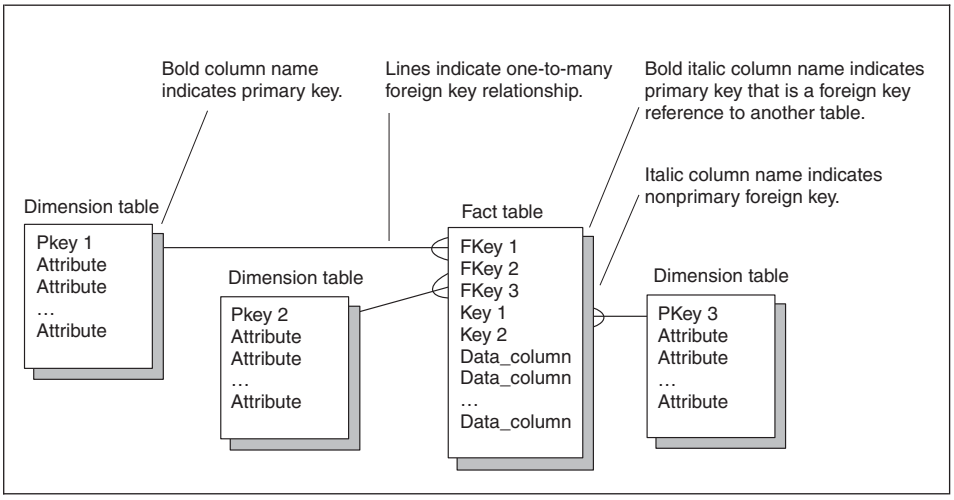


Figure 17. Relationship of fact and dimension tables in multistar schema

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.

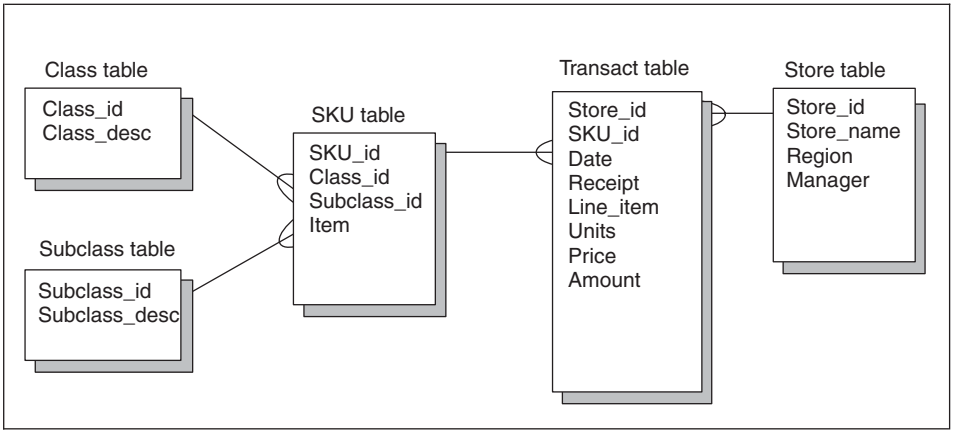


Figure 18. Multistar schema with two outboard tables

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, see the *IBM Red Brick 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.

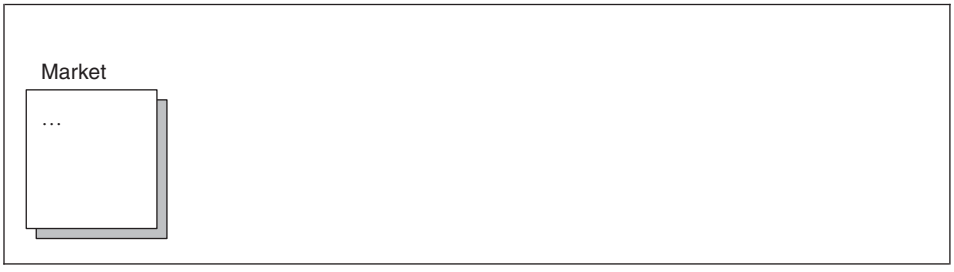


Figure 19. Single dimension table

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

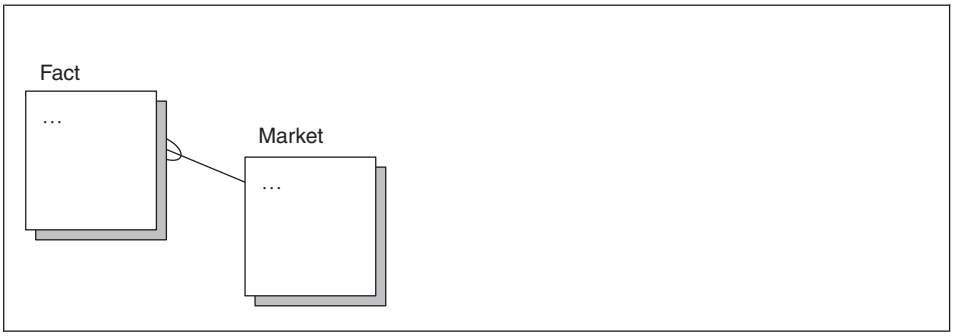


Figure 20. Star schema with one dimension table

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

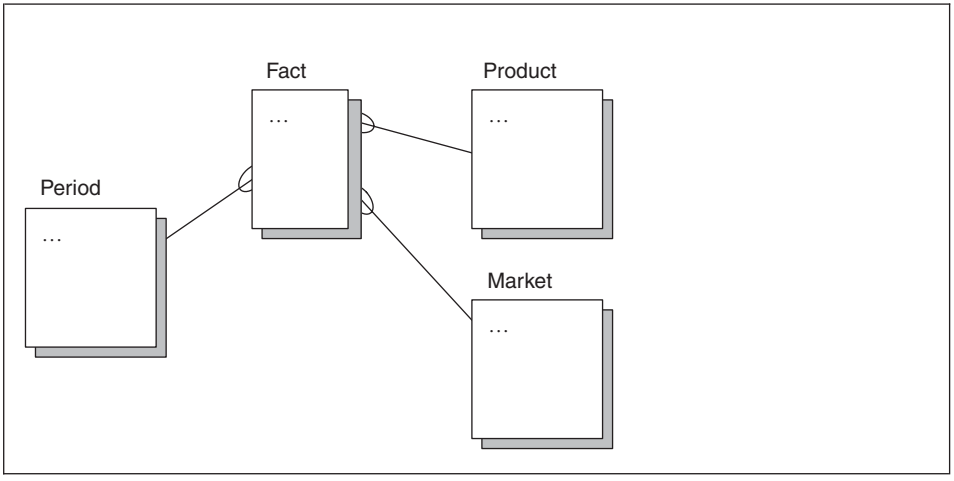


Figure 21. Star schema with several dimension tables

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

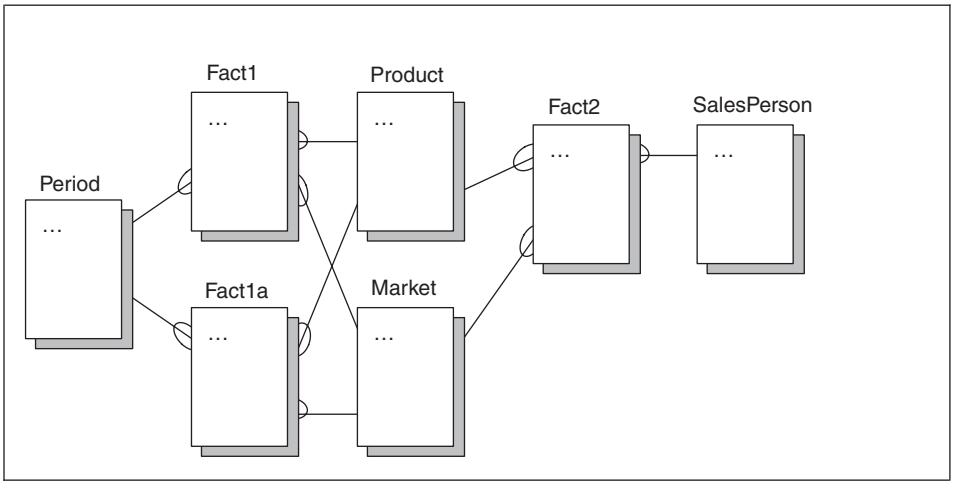


Figure 22. Star schema with several fact tables

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

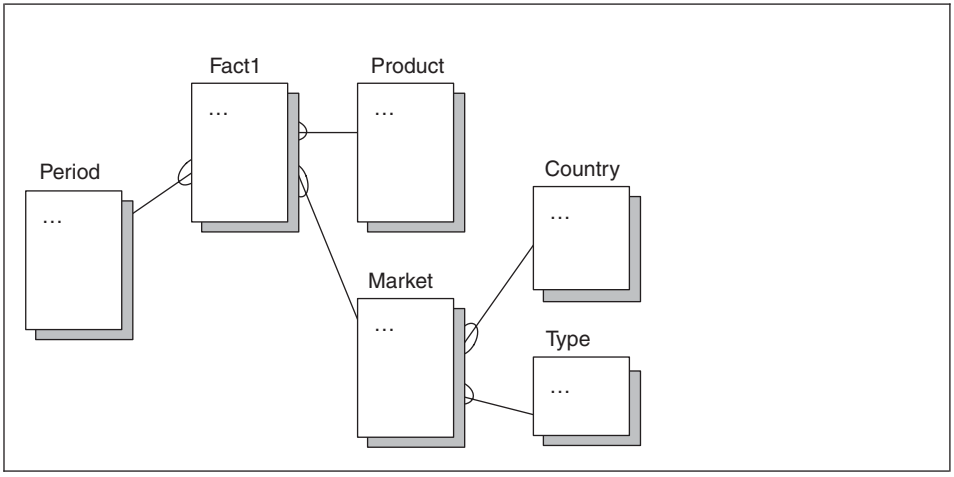


Figure 23. Star schema with outboard tables

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

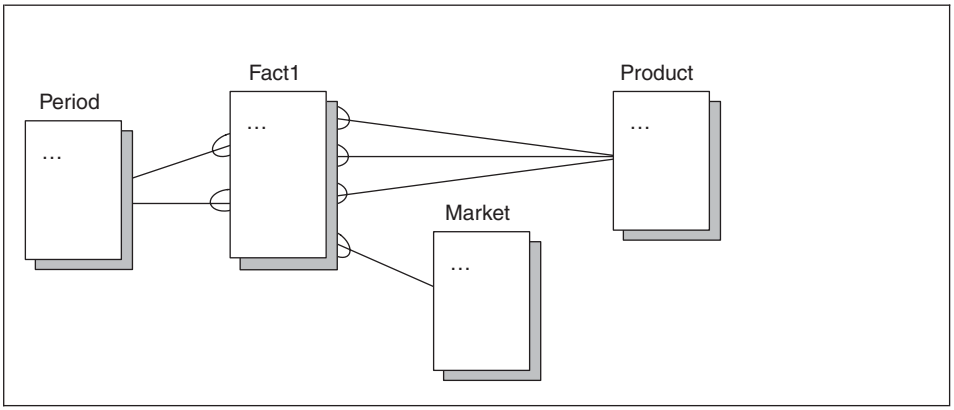


Figure 24. Star schema with multiple foreign keys

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.

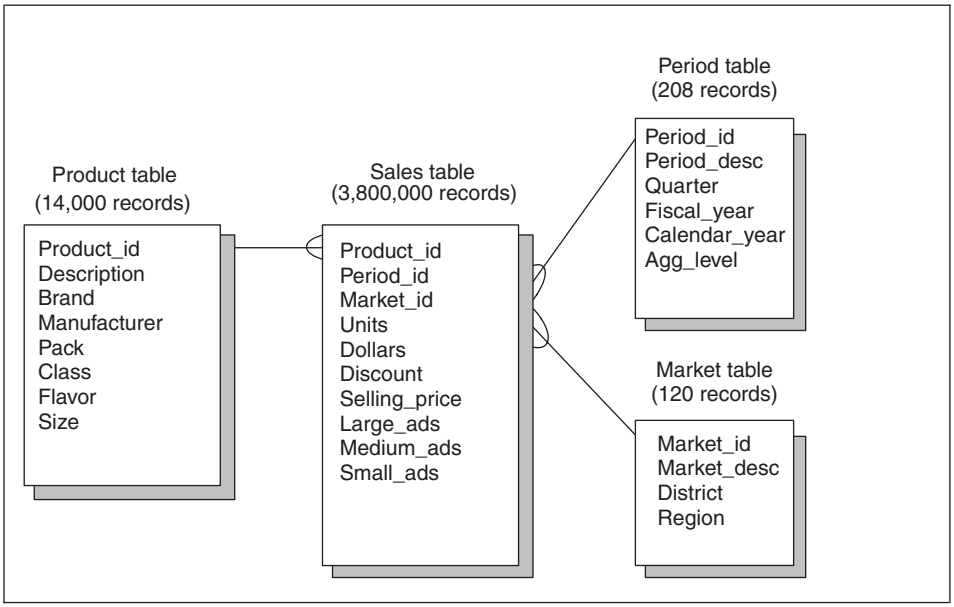


Figure 25. Salad dressing database example

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)
Distribution
Manufacturer
Brand Group
Brand
Class
Viscosity
Flavor
Size
Special Package
Promotion

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.

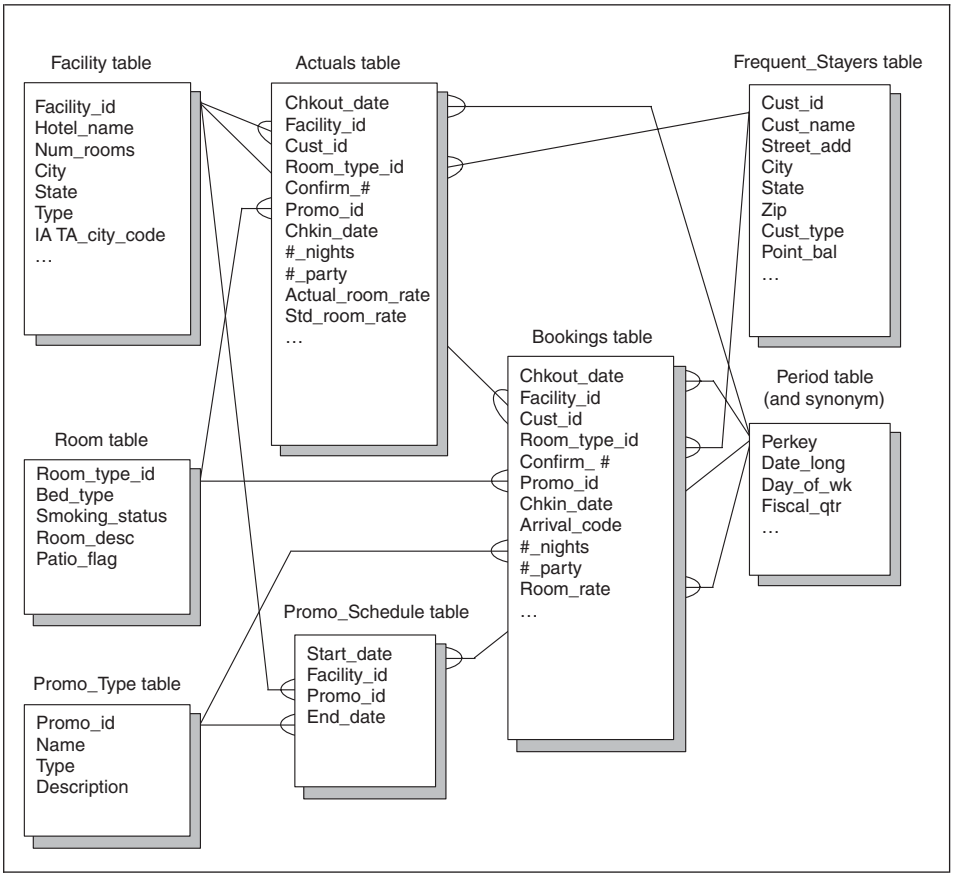


Figure 26. Schema for reservation system

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 nonaggregated fact table. In some situations, this space-for-performance trade-off 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 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, see the *IBM Red Brick 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.

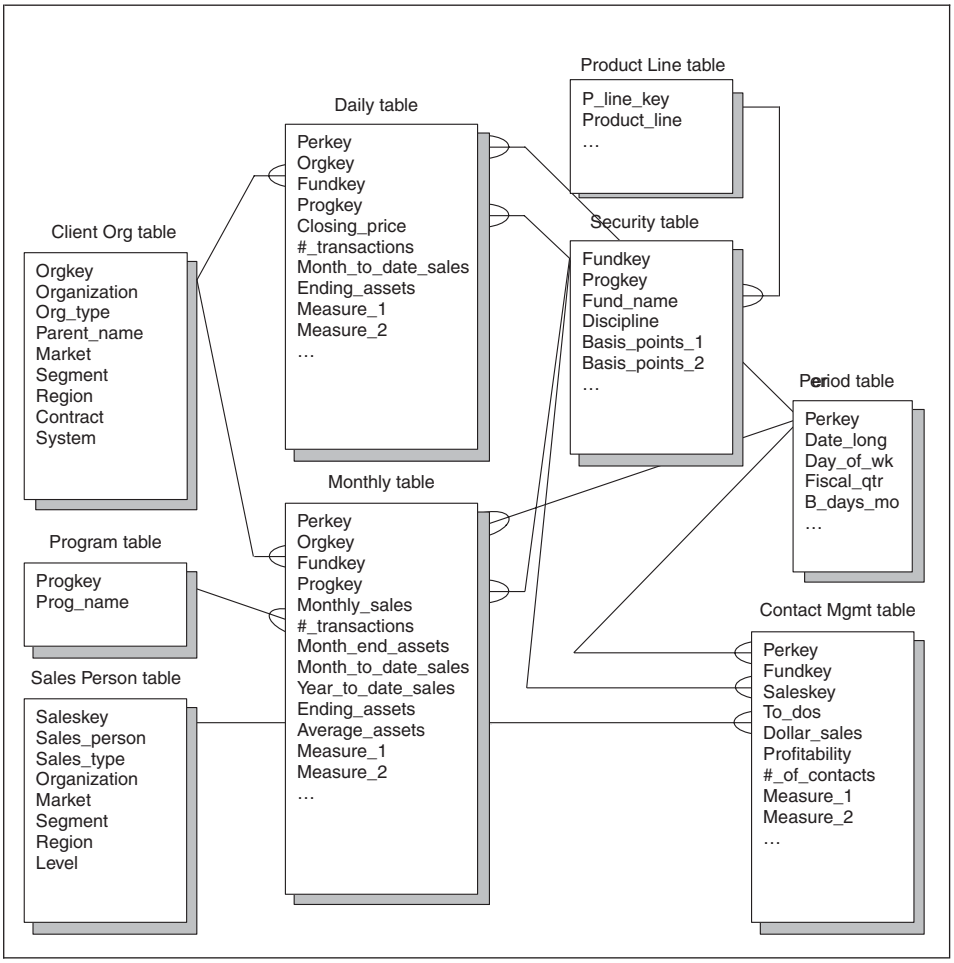


Figure 27. Schema for investment database

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.

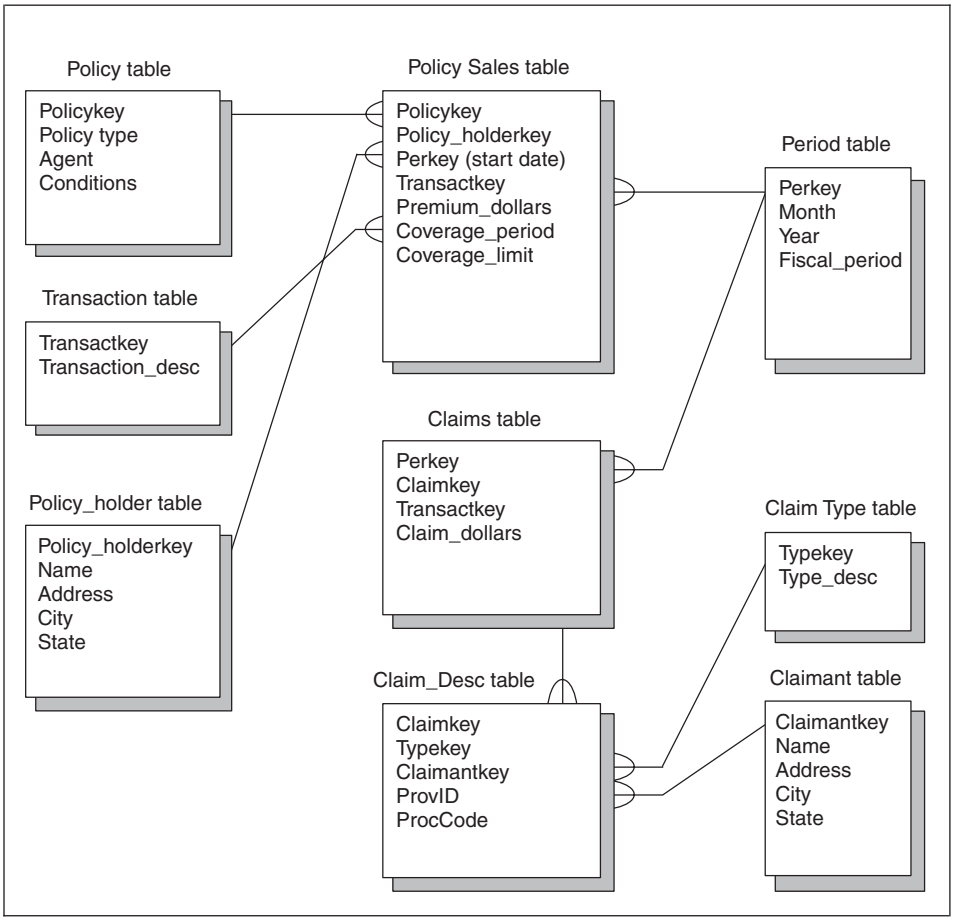


Figure 28. Schema for health insurance company

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.

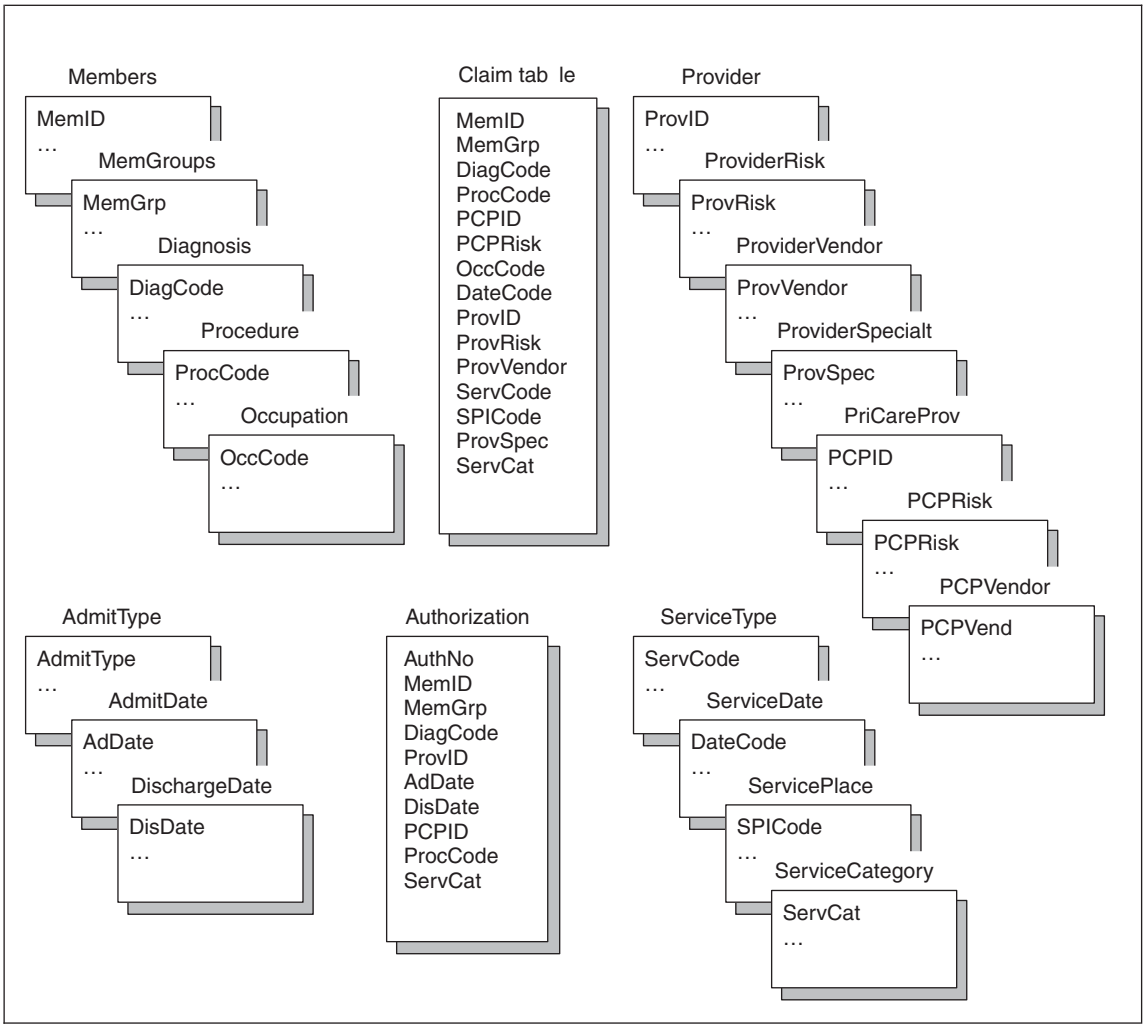


Figure 29. Multistar schema for health insurance company with multiple dimension tables

Chapter 4. Planning a database implementation

This chapter describes the planning that must be done before you implement an IBM Red Brick Warehouse database. It includes the following sections:

- Organizing data into databases
- Determining when to create additional indexes
- Planning for segmented storage
- Planning for a time-cyclic database
- Planning disk storage organization
- Estimating temporary space requirements
- Considerations for growing tables

Organizing data into databases

In planning for a new Red Brick 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,” on page 55.

Determining when to create additional indexes

In a Red Brick server database, a B-TREE index is automatically created on the primary key of a table during table creation. Query performance might improve if you create additional indexes. However, the improved query performance must be balanced against the additional storage space for each

© Copyright IBM Corp. 1996, 2004

index and the additional time to build or update it during each INSERT, UPDATE, DELETE, or LOAD operation. 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 or a STAR index on the primary key.

Creating additional indexes can have a large impact on query performance. For information about how Red Brick chooses the indexes to use in a query and the algorithms used to join tables, see the *Query Performance Guide*.

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.
- STAR indexes are only useful when they cover several foreign-key columns.
- 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.
- 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, see the *Query Performance Guide*.
- If you have queries against multiple fact tables (fact-to-fact table joins), you might need to create additional STAR indexes to satisfy the following requirements:
  - 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.

  If these conditions are not satisfied, the query proceeds using a different join algorithm and might not perform as well. For more information, see "Queries against multiple fact tables" on page 83.
- If you have time-cyclic data stored in multiple segments, segment the indexes on the fact table like the data. When you detach a data segment,

the corresponding index segments can be cleared quickly. When you attach the next index segment, you do not need to rebuild the index.

For more information, see "Planning for a time-cyclic database" on page 92, "Creating a time-cyclic database" on page 155,, and "Maintaining a time-cyclic database" on page 294.

- Define a B-TREE index on each UNIQUE column to enforce the uniqueness constraint.
- If your database contains any outboard tables, create an index on each foreign key of each table referencing an outboard table.
- Create indexes on aggregate tables equivalent to those created on detail tables. For details, see the *IBM Red Brick Vista User's Guide*.
- If queries 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 Warehouse uses STAR indexes to greatly improve performance on queries involving tables that have foreign keys. If the table has more than one foreign key, you can create a STAR index on any combination or subset of those foreign keys.

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.

A STAR index can act as the primary key of the fact table. You should drop the default primary key index when you create a STAR index that covers all the primary key columns of a fact table for the following reasons:

- Reduce the amount of disk space required for indexes.
- Reduce the number of indexes to maintain during INSERT, DELETE, UPDATE, LOAD or REORG operations.

Order of columns

If you have multiple STAR indexes, the order of the foreign key constraints in a query determines which index is used. IBM Red Brick Warehouse 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:

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

Queries against multiple fact tables

If you have queries against multiple fact tables, STAR indexes can be used to join them if the foreign key columns in the index definitions are in the same relative order.

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). For each table, suppose you build a STAR index that includes all the foreign keys, in the order they are specified in the CREATE TABLE statement. 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 for using STAR indexes 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).

- 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 (known as a B-TREE 1-1 match join).

The Red Brick server can use multicolumn B-TREE indexes for TARGETjoins. For more information on TARGETjoins, see the *Query Performance Guide*.

Tip: If you create a multicolumn B-TREE index, all columns included in the index must be declared NOT NULL.

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.

TARGET indexes can improve performance on queries involving columns that have weakly selective constraints. Performance improvements are two-fold. The queries run faster and require less memory to process.

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 information about using TARGET indexes to enable TARGETjoin query processing, see the *Query Performance Guide*.

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 server selects the appropriate storage method, or representation, for the TARGET index information.

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 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.

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.

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. 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 TARGET index.

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

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;
```

Locally segmented indexes

A local index is a TARGET or B-TREE index that is segmented in the same way as the table data, even if the leading column of the index is not the segmenting column of the table. Because the indexes do not need to be rebuilt, local indexes are advantageous in a time-cyclic database where the oldest data is rolled off and the latest data is added at regular intervals. For more information, see "Characteristics of a local index" on page 94, "Creating a time-cyclic database" on page 155,, and "Maintaining a time-cyclic database" on page 294..

A local index can also improve TARGETjoin performance. However, tuning actions must be taken when a local index is used in other query processing. For details, see the *Query Performance Guide*.

No indexes

If you join tables that do not have indexes covering the join path and the join is an equijoin, IBM Red Brick Warehouse 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, see the *SQL Reference Guide*.

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. When you use named segments, you need to plan storage sizes and how to segment the data and indexes.

Determining when to use default and named segments

For large data warehouses, IBM recommends that you use named segments. Named segments give you control over where the data is placed and is required to take advantage of parallel query execution and other advanced features of the IBM Red Brick Warehouse. For a list of advantages, see "Segmented storage" on page 20..

Default segments are useful for small and test databases. End-users and application tools often analyze query results that they store in semi-permanent tables. It is convenient to place these tables in default segments because the Red Brick server creates them automatically.

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.

Important: 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.

Storing tables and indexes in default segments simplifies database creation and administration, but they do not have the flexibility of named segments. In particular, 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," on page 365.

- 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. For detailed procedures, see "Altering segments" on page 283.

For more information about default and named segments, see "Creating segments" on page 132 and "Altering segments" on page 283.

Using named segments

When 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.
- IBM Red Brick Warehouse 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.
- When a table or index is stored in multiple segments, a subset of the segments can be taken offline for loading or restore operations and the other segments 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 concurrent query access and 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, a relation scan of that table cannot be processed in parallel. Therefore, keep in mind the desired degree of parallelism when you define PSUs. For more information about PSUs and parallelism, see the *Query Performance Guide*.

Deciding how to segment

Table 1 shows the different ways to distribute row and index data among multiple segments and when you would use each way.

Table 1. Segment specification and when to use SQL keywords

Contents of segments	How segmented	Keywords in CREATE INDEX	When to use
Row data	By data value of segmenting column	SEGMENT BY VALUES OF	Need to know in which segment the data resides.
	By hashing	SEGMENT BY HASH	Want even distribution of data among segments
STAR indexes	Evenly across segments (Red Brick Warehouse calculates ranges)	none	Want even distribution of index entries among segments
	By row numbers of referenced table containing the segmented column of the STAR index	SEGMENT BY REFERENCES OF	Need to know in which segment the index entries reside and referenced table is not segmented
	By row IDs of segmented referenced table containing the segmented column of the STAR index	SEGMENT LIKE REFERENCED TABLE	Need to know in which segment the index entries reside and referenced table is segmented

Table 1. Segment specification and when to use SQL keywords (continued)

Contents of segments	How segmented	Keywords in CREATE INDEX	When to use
	Logically by the same values as the data (still uses row IDs)	SEGMENT LIKE DATA	Facilitate maintenance of time-cyclic databases
TARGET indexes	By values of the indexed column	SEGMENT BY VALUES OF SEGMENT LIKE DATA	Indexed column is not segmenting column of table Leading column is segmenting column of table
	By the same values as the data	SEGMENT LOCAL	Facilitate maintenance of time-cyclic databases or to improve TARGETjoin processing
B-TREE indexes	By values of the first (leading) indexed column	SEGMENT BY VALUES OF SEGMENT LIKE DATA	Leading column is not segmenting column of table Leading column is segmenting column of table
	By the same values as the data	SEGMENT LOCAL	Facilitate maintenance of time-cyclic databases or to improve TARGETjoin processing

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

  If you do not specify an initial size when you create the segment, it defaults to 16 kilobytes.
- A maximum size, which limits how large the file can grow.

  If you do not specify an maximum size when you create the segment on the Red Brick Administrator tool, it defaults to 16 kilobytes. You must specify a value for the MAXSIZE keyword when you create the segment in RISQL.

- 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

Planning for a time-cyclic database

A common use of a data warehouse is to store time-cyclic data where the fact table is usually segmented by a period dimension such as date. Time-cyclic data is also a major reason for using named segments. For example, suppose your business requires that you keep the current two years worth of data online. You can create a fact table to hold this data and segment it by week so that 104 segments exist. In this sample time-cyclic database, the following tasks are performed each week to keep only the most current 104 weeks in the database:

- Detach the oldest weekly data and its indexes
- Create new segments for the data and indexes or reuse the segments you detached
- Attach the data segment with ranges for the next week to the end of the fact table

- Attach the corresponding index segments.
- Load data into the newly attached segment

When you setup a time-cyclic database, take the following actions to improve the performance of these weekly tasks:

- Segment the indexes on the fact table like the data
- Define local indexes
- Keep a small, empty segment at the end of the table

Segment indexes like the fact table

Segment the indexes on the fact table like the data so that when you detach the data segment the corresponding index segments can be cleared quickly. You can segment the index the same way as the data in the following situations:

- When the leading column of the STAR index is the segmenting column of the fact table, create the index with the SEGMENT LIKE DATA clause.
- When the leading column of a B-TREE or TARGET index is the segmenting column of the fact table, create the index with one of the following clauses:
  - SEGMENT LIKE DATA

    Use this clause when you cannot use a local index. For example, use this clause on unique columns or when you want to change the segmentation of the index separately from the table.
  - SEGMENT LOCAL

    When the local index column is the segmenting column of the table, you obtain the advantages for time-cyclic data maintenance of a local index and the performance advantages of a non-local index. For local index characteristics, see page 94.. For performance advantages of a non-local index, see the *Query Performance Guide*.
- When the leading column of a B-TREE or TARGET index is not the same column by which the fact table is segmented, create a local index.

  The column by which the fact table is segmented does not need to be the leading column of a B-TREE index when you use the SEGMENT LOCAL clause. Similarly, the segmenting column does not need to be the index key of a TARGET index when you use the SEGMENT LOCAL clause.

Tip: To improve TARGETjoin and TARGET scan operations, create local indexes even if the segmenting column of the table is the leading column of the index.

Note that you must populate the dimension table of the leading column in the index before you can create the STAR index with the SEGMENT LIKE DATA clause. For examples of the actual CREATE INDEX statements, see “Creating a

time-cyclic database" on page 155.. For considerations when detaching and attaching segments, see"Maintaining a time-cyclic database" on page 294.

Characteristics of a local index

A local index facilitates the maintenance of time-cyclic data because it has the following characteristics:

- When you detach or clear a table segment, only the local index segment is cleared without specifying the OVERRIDE FULLINDEXCHECK clause in the ALTER SEGMENT statement.
- When you change or move the range of a table segment, the local index segment range is automatically changed or moved without executing a separate ALTER SEGMENT statement.
- If you detach or attach a table segment, you must detach or attach the corresponding local index segments. Otherwise, you receive an error if you try to execute SELECT, INSERT or DELETE statements on the table:

  ```
  ** ERROR ** (1204) The segmentation of local index
   PERKEY_SALES_LOCAL_TIX does not match table segmentation.
  ```

- Local indexes improve the performance of TARGETjoin and TARGET scan operations. However, local indexes can affect performance of other query processing unless you set certain tuning parameters. For details, see the *Query Performance Guide*.

For examples of the CREATE INDEX statement, see "Creating the local TARGET indexes" on page 165.. For procedures to detach and attach segments, see"Rolling off and reusing data and index segments" on page 295.

Small segment at the end of the table and each index

The ATTACH SEGMENT operation uses the range of an existing segment to overlay with the new segment range. The Red Brick server scans this segment whose range is being overlaid to ensure that no rows exist that belong in the new segment.

To improve the performance of the ATTACH SEGMENT operation, keep an empty segment at the end of the table where you attach new segments. This empty segment should be 16 kilobytes (minimum size) so that data cannot accidentally be loaded into it. For examples of the actual CREATE SEGMENT and CREATE TABLE statements, see "Creating a time-cyclic database" on page 155. For examples of ALTER SEGMENT statements to detach the data segment and attach a new segment, see "Maintaining a time-cyclic database" on page 294.

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. For more information, see “Planning for segmented storage” on page 88 and “Planning for a time-cyclic database” on page 92.
5. 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 123.

IBM Red Brick Warehouse includes the dbsize utility in the service directory under the *redbrick_dir\util* directory. For information about this utility, see the `README` file, also in the service directory.

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 to calculate the length in bytes of each row and the number of bytes required to store all the rows.

For information on setting VARCHAR column fill factors for maximum performance, see "Setting the VARCHAR column fill factor" on page 142.

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, see "Estimating temporary space requirements" on page 108.

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. If you want dbsize to calculate the row length, answer the next two questions as follows:
   a. Reply `n` to the question to see examples of how to calculate table sizes.
   b. Reply `y` to the question to let dbsize calculate the row length based on the size of the data types you specify.

4. 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.
5. 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, see "Setting the index fill factor" on page 151.

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 table referenced by a foreign key in the STAR index, the estimated maximum number of rows per segment (MAXROWS PER SEGMENT) and the maximum number of segments for that table (MAXSEGMENTS).
- The estimated number of rows to be included in the referencing (fact) table
- The fill factor for the STAR 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) in the indexed column
- The data type of the column in the TARGET index
- The estimated number of rows in the table
- The number of segments in the table
- The domain for the column: SMALL, MEDIUM or LARGE

  If you do not provide a domain, dbsize calculates the domain that results in the smallest index and tells you which domain it is using.

If the table is segmented, dbsize also asks for the following information:

- Whether the index is a local index
- Whether the column is the segmenting column of the table

The size of a TARGET index is affected by many factors, particularly by the skew in the numbers of rows for each key, which can cause the estimated size of a TARGET index to be inaccurate. Because domain MEDIUM uses compression, the dbsize estimates are less likely to be consistent with the actual index size than the estimates made for domain LARGE and SMALL. When a domain is not specified, dbsize estimates the size with all three domains and suggests the domain that results in the smallest index.

Example: Calculating local TARGET index Size:

This example illustrates how to size a local TARGET index. Suppose the Sales table has approximately 70,000 rows and is defined as follows:

```
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 10
> maxrows per segment 60000;
```

You want to create a local TARGET index on the Promokey column, which has 300 unique values.

Use dbsize to calculate the size of this local TARGET index:

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 Promokey column, which is `300` in this example.
5. Enter the data type for the column, which is `integer` for this example.
6. Enter the estimated number of rows in the Sales table, which is `70000` in this example.
7. Enter the number of segments in the table, which is `2` in this example.
8. For the domain type, hit the Enter key so that dbsize calculates the domain that results in the smallest index and displays the following message:

   ```
   Using domain LARGE
   ```

   For a description of domain size for TARGET indexes, see "TARGET indexes" on page 85.
9. Reply `y` to the question that asks if this is a local index.
10. Reply `n` to the question that asks if this the segmenting column of the table.
11. For the CREATE TABLE statement for Sales, dbsize estimates the size of the local TARGET index on the Promokey column to be 480 kilobytes.

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. Enter the number of columns in the table, which is `6` in this example.
4. If you want dbsize to calculate the row length, answer the next two questions as follows:
   a. Reply `n` to the question to see examples of how to calculate table sizes.
   b. Reply `y` to the question to let dbsize calculate the row length based on the size of the data types you specify.
5. Respond to the prompts for 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 `53000000` in this example.
6. 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, which is `5000` and `520000` rows in this example.
5. Enter the estimated number of rows for the fact table, which is `53000000` 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, which is `integer` for column in this example.
6. Enter the estimated number of rows for the table, which is `53000000` 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. Enter the number of columns in the table, which is `2` in this example.
4. If you want dbsize to calculate the row length, answer the next two questions as follows:
   a. Reply `n` to the question to see examples of how to calculate table sizes.
   b. Reply `y` to the question to let dbsize calculate the row length based on the size of the data types you specify.

5. Respond to the prompts for the data types of the columns and the total number of estimated rows for the table, which is `5000` in this example.
6. 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 number of segments in the table, which is `1` for this example.
9. Enter the domain size, which is `SMALL` in this example. For a description of domain size for TARGET indexes, see "TARGET indexes" on page 85.
10. Reply `y` to the question that asks if this is a local index.

11. Reply `n` to the question that asks if this the segmenting column of the table.
12. 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. Enter the number of columns in the table, which is `2` in this example.
4. If you want dbsize to calculate the row length, answer the next two questions as follows:
   a. Reply `n` to the question to see examples of how to calculate table sizes.
   b. Reply `y` to the question to let dbsize calculate the row length based on the size of the data types you specify.
5. Respond to the prompts for the data types of the columns (`integer` for both columns n this example), and the total number of estimated rows for the table, which is `520000` in this example.
6. 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. If the **RB_DEFAULT_IDX** file grows very large and contains a significant number of free blocks between active blocks, you might need to compact it. For details, see "Managing memory requirements for a database catalog" on page 339.

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. Index-building operations and complex queries can require large amounts of temporary space to store intermediate results. Temporary space for index building is controlled by the INDEX_TEMPSPACE configuration parameters. For queries, it is controlled by the QUERY_TEMPSPACE and QUERY_MEMORY_LIMIT configuration parameters. These configuration parameters specify one or more directories in which to store temporary files, a maximum amount of disk space to be used as temporary space by a single operation, the threshold at which intermediate results spill from memory to disk, and the amount of memory allocated to a query.

This section describes temporary space requirements in terms of:

- How temporary space is used for optimized index-building operations and how to calculate the requirements for these operations
- How temporary space is used for queries and how to calculate the requirements
- How temporary space is used for the results of a query

For information about temporary space requirements for aggregate maintenance, see the *IBM Red Brick Vista User's Guide*.

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 Only

- 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 IBM Red Brick Warehouse 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.

End of UNIX Only

Windows Only

- Physical memory

  Consider the amount of memory on your computer.

  Note: You can configure your Windows server machine to enable up to 3 gigabytes of address space for Red Brick Warehouse applications. With a larger address space, more data can be cached in physical memory and more threads can be spawned and allocated, improving both performance and scalability. For configuration instructions, see the *Installation and Configuration Guide*.

End of Windows Only

- 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 temporary disk space allocation and temporary space parameters, see pages 110 and 112..

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.

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 four temporary space directories—d1, d2, d3, d4—are defined, a sequence for one spill operation might be d2, d1, d4, d3, and the sequence for another concurrent spill operation might be d4, d3, d1, d2. Figure 30 illustrates this concept of the temporary space directories that make up a circular buffer.

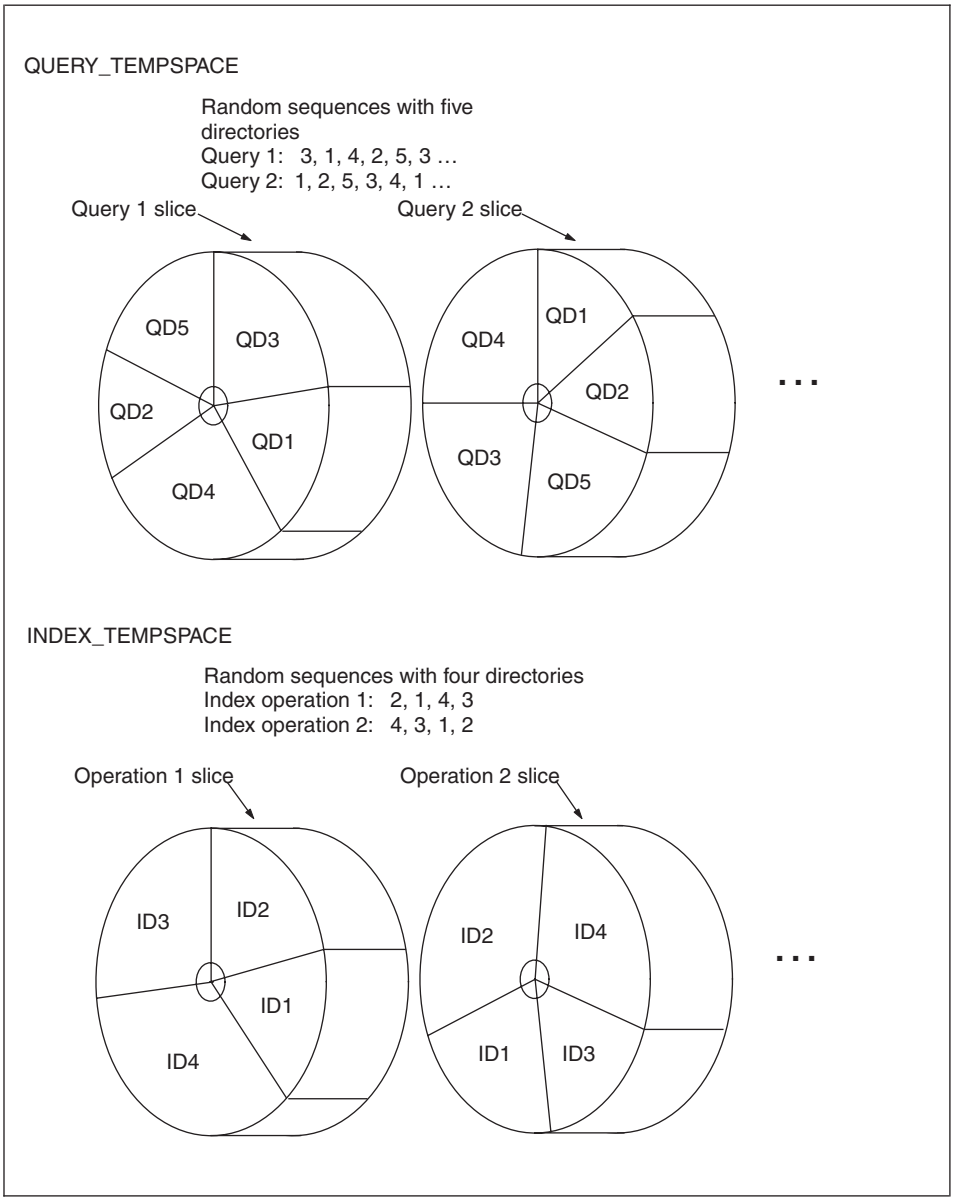


Figure 30. Random directory sequence

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.

The directories are used in the random sequence determined for each spill. However, in directories containing multiple files, 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 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 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 index operations as follows:

- 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.

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.

The following parameters and corresponding SET commands control the location and use of temporary space for index builds.

Table 2. 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. *

Table 2. For index building temporary space (continued)

TUNE parameter and SET command	Function
TUNE INDEX_TEMPSPACE_THRESHOLD, SET INDEX TEMPSPACE THRESHOLD	Specifies size at which index-building operations or CHECK INDEX VALIDATE FULL 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.
SET INDEX TEMPSPACE RESET	Resets the query or index TEMPSPACE parameters to the values specified in the `rbw.config` file.

* IBM Red Brick Warehouse and the TMU automatically spread temporary space usage evenly across the designated directories.

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. You can enter SET commands in the following places:

- `.rbwrc` files
- Anywhere you can enter SQL statements, such as a RISQL session or an ISQL window on the Red Brick Warehouse Administrator tool
- TMU control file

DIRECTORY location values

Temporary space for 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. To define a set of directories using entries in the **rbw.config** file, enter multiple lines.

The following example illustrates entries in the `rbw.config` file that apply to all sessions:

Windows Only

```
TUNE INDEX_TEMPSPACE_DIRECTORY /dbindex/spill1
TUNE INDEX_TEMPSPACE_DIRECTORY /dbindex/spill2
TUNE INDEX_TEMPSPACE_DIRECTORY /dbindex/spill3
```

End of Windows Only

Windows Only

```
TUNE INDEX_TEMPSPACE_DIRECTORY d:\itemp
TUNE INDEX_TEMPSPACE_DIRECTORY e:\itemp
TUNE INDEX_TEMPSPACE_DIRECTORY f:\itemp
```

End of Windows Only

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.

```
SET INDEX TEMPSPACE DIRECTORIES '/qa/local/virginia/aroma/spill1',
   '/qa/local/virginia/aroma/spill2',
   '/qa/local/virginia/aroma/spill3';
```

For more information about how space is allocated among the various directories designated as temporary space directories, see page 110..

THRESHOLD value

The threshold specifies how much memory is used before spilling the intermediate results from an index build operation or CHECK INDEX VALIDATE FULL operation to temporary space on disk.

The default value for INDEX_TEMPSPACE_THRESHOLD is 10 megabytes (`10M`) to accommodate small systems. If you want to change this threshold value, use the SET INDEX TEMPSPACE statement with the THRESHOLD option and keep the default value in the **rbw.config** file.

The goal in selecting a threshold for INDEX_TEMPSPACE is to select a large value relative to the system on which IBM Red Brick Warehouse 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.

To estimate the threshold (UNIX only):

1. The threshold value should never exceed half of the physical memory size on the computer. 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. 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 appropriate for the operation you are performing.
   a. For loads and CREATE INDEX operations, IBM recommends that you set INDEX_TEMPSPACE_THRESHOLD to a value that is one quarter to one half of the maximum data space size for your system.
   b. For CHECK INDEX with the VALIDATE FULL option, IBM recommends that you set INDEX_TEMPSPACE_THRESHOLD to a value that is one quarter of the maximum data space size for your system.

      Keep in mind that INDEX_TEMPSPACE_THRESHOLD is the amount of memory per user. Therefore, if you run CHECK INDEX with the VALIDATE FULL option while several users are running memory intensive operations at the same time, divide the threshold setting by the number of users.

      For example, if two users run memory intensive operations while you run CHECK INDEX with the VALIDATE FULL, divide the threshold setting by three. Divide the threshold value 64 in step 2 and set the threshold to 22:

      ```
      SET INDEX TEMPSPACE THRESHOLD 22M
      ```

      The value you enter is automatically rounded up to the nearest 8-kilobyte block (24 in this example).

To estimate the threshold (Windows only):

1. Determine the amount of physical memory on your computer. The maximum physical memory on Windows is 2 gigabytes.
2. Select a value for INDEX_TEMPSPACE_THRESHOLD that is appropriate for the type of operation you are performing. The same recommendations listed in step 2 of the preceding UNIX section apply to Windows.

MAXSPILLSIZE value

The maximum spill size specifies how much temporary disk space an operation can use. 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 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 disk space. Selecting a value that is larger than the available disk space 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, as the following sections describe.

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 IBM Red Brick Warehouse 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.

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 INDEX_TEMPSPACE parameters in effect for the current session, enter a query similar to the following one:

```
RISQL> select substr(option_name, 1, 30), substr(value, 1, 40)
   from rbw_options
   where username = CURRENT_USER
      and option_name like 'INDEX?_%' escape '?';

INDEX_TEMPSPACE_DIRECTORY      /qa/local/virginia/aroma/spill
INDEX_TEMPSPACE_THRESHOLD      10240 Kbytes
INDEX_TEMPSPACE_MAXSPILLSIZE   1 Gbytes
INDEX_TEMPSPACE_DUPLICATESPILL 2
```

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 IBM Red Brick Warehouse 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

UNIX Only

The default cleanup script, *redbrick_dir/bin/rb_sample.cleanup*, 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 script also removes orphaned temporary segment PSUs. The supplied cleanup script does not find and remove spill files from locations specified with a SET command during a database server or TMU session. You must modify the script to remove spill files from these locations.

End of UNIX Only

Windows Only

The `redbrick_dir\bin\rbclean.bat` file is a template that shows sample commands to erase spill files from the `c:\temp` directory. You must remove the "REM" preceding each ERASE command to execute them. You must also modify the script to remove spill files and orphaned temporary segment PSUs from the directories specified in the QUERY_TEMPSPACE_DIRECTORY,

INDEX_TEMPSPACE_DIRECTORY, TEMPORARY_DATA_SEGMENT, and TEMPORARY_INDEX_SEGMENT configuration parameters.

End of Windows Only

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, see the *Query Performance Guide*.

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.)

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 IBM Red Brick Warehouse 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 a line with the following format in the **rbw.config** file:

```
TUNE RESULT_BUFFER value
```

value Possible values for the **rbw.config** parameter or the SET statement include:

- An integer value, which must be followed by K (kilobytes), M (megabytes), or G (gigabytes).
- The keyword UNLIMITED, which 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.
- The value `0` specifies that no results will be buffered.

The corresponding SQL SET statement has the following format:

```
set result buffer value;
```

RESULT BUFFER FULL ACTION parameter

To specify the behavior when the results buffer size specified with the RESULT_BUFFER parameter is reached, enter a line with the following format in the `rbw.config` file:

```
TUNE RESULT_BUFFER_FULL_ACTION action
```

The corresponding SQL SET statement has the following format:

```
set result buffer value;
```

action Possible actions include:

- The keyword ABORT indicates that the query will abort when the buffer size is reached.
- The keyword 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;
```

Considerations for growing tables

In some applications, database tables grow over time. IBM Red Brick Warehouse 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," on page 365.

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.

IBM 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

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 a time-cyclic database
- 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, see the *SQL Reference Guide*. For a complete example of how to create a database, see Appendix A, "Example: Building a database," on page 343.

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," on page 195, and load

© Copyright IBM Corp. 1996, 2004

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 IBM Red Brick Warehouse is installed. If you want a different locale, you must choose it from the locales that are available. For information about specifying a locale, see "Server locale" on page 35. A list of locales is provided in the `locales.pdf` file in the `relnotes` directory on the IBM Red Brick Warehouse product CD-ROM.

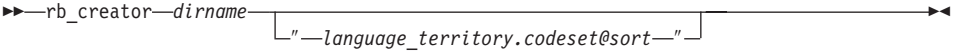
Database creation syntax

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 34..

UNIX Only

To create a database on UNIX:

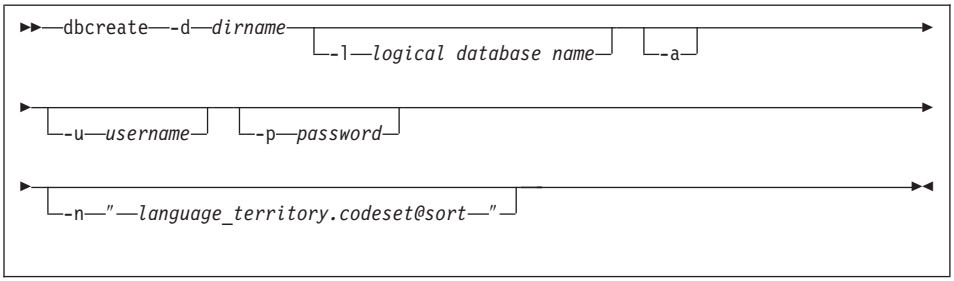


dirname	The database directory
"language_territory.codeset@sort"	Optional. The locale for the database if the locale differs from the IBM Red Brick Warehouse locale (set with the NLS_LOCALE LOCALE parameter in the `rbw.config` file)

End of UNIX Only

Windows Only

To create a database on Windows:



-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 (Default is `SYSTEM`).

-p *password* — Optional (Default is `MANAGER`).

-n *"language_territory. codeset@sort"* — Optional. The locale for the database if the locale differs from the IBM Red Brick Warehouse locale (set with the NLS_LOCALE LOCALE parameter in the `rbw.config` file).

End of Windows Only

For instructions on deleting a database, see "Deleting a database" on page 336.

Initializing the database

Initialize a new database with the rb_creator utility on UNIX or the dbcreate utility on Windows.

To initialize a new database (Unix only):

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 path name 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 126.

   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 Table 3 on page 129..

Warning: *You cannot create a Red Brick database on a Network File Server (NFS) system.*

To initialize a new database (Windows only):

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 path name 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 126.

   Running dbcreate initializes the database by creating the database system files listed in Table 3 on page 129..

   When you use the dbcreate utility to initialize a database, you must fully qualify the path name for the database directory by including the drive letter. For example:

   ```
   c:\> dbcreate -create -l AROMA -d c:\disk1\new_db
   ```

The database path name is added to the rbw.config file automatically. If you edit the database section of this file, be sure that the path name for each database is fully qualified.

If two logical database names in the rbw.config file see the same physical database, the path names 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.

Table 3. 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 path name; 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 path name to the database directory. (On UNIX, this information is case sensitive.)

Windows Only

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.

End of Windows Only

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
   ```

   where *logical_database_name* is the 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*.

Windows Only

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.

End of Windows Only

Example:

This example illustrates how to change the password for the user system from the default password, manager, to the new password, mysecret.

UNIX Only

```
$ risql -d new_db system manager
(C) Copyright IBM Corp. 1991-2003. All rights reserved.
RISQL Reporter Version 06.30.0000(0)
RISQL> grant connect to system with mysecret;
RISQL> quit;
$
```

End of UNIX Only

Windows Only

```
c:\> risql -d new_db system manager
(C) Copyright IBM Corp. 1991-2003. All rights reserved.
RISQL Reporter Version 06.30.0000(0)
RISQL> grant connect to system with mysecret;
RISQL> quit;
c:\>
```

End of Windows Only

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, see the *RISQL Entry Tool and RISQL Reporter User's Guide*. For information about writing CREATE TABLE statements, see "Creating tables" on page 140.

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 149.

   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 *IBM Red Brick Vista User's Guide*.

For a complete description of the SQL syntax, see 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. When you use named segments, you must define them before you create the tables and indexes. For examples of creating segments, see "Creating the data and index segments" on page 157.

When you create the segment, the Red Brick Warehouse allocates the INITSIZE value of each PSU. The first PSU must have an INITSIZE of at least 16 kilobytes (2 blocks), and the other PSUs can have an INITSIZE of 0 or more. As data fills the PSU, space is added in increments of the EXTENDSIZE until the PSU reaches the MAXSIZE. When the PSU reaches the MAXSIZE, storage is allocated from the next PSU in sequence, first using any blocks that were allocated as INITSIZE when the segment was created. For the syntax of the CREATE SEGMENT statement, see the *SQL Reference Guide*.

A segment with large INITSIZE values for its PSUs takes longer to create (because the INITSIZE space is being allocated), but is faster to load than a segment with smaller INITSIZE values.

Segments can be modified as needed with an ALTER SEGMENT statement. For information about modifying segments, see "Altering segments" on page 283..

Rules for segmenting

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. Use the hashing algorithm under the following conditions to obtain a more even distribution of data among segments:
  - The number of segments is a prime number.
  - Only columns defined with the NOT NULL clause exist in the row.
  - Large columns (such as REAL and character) whose values do not vary much among the rows are near the beginning of the table.
- Each TARGET and B-TREE index can reside in multiple segments with index entries distributed in one of two ways:
  - By ranges of values in the leading column of the index

    The Red Brick server provides different SQL clauses to define this segmentation of the index. For a list of these clauses, see Table 4 on page 134..
  - By the same column values used to segment the table

    The segmenting column does not need to be part of the index, but one index segment corresponds to each table segment. This segmentation is defined with the SEGMENT LOCAL clause in the CREATE INDEX statement, and the index is knows as a *local index.*

    A local index is most commonly used for TARGETjoin processing of time-cyclic data and does not apply to STAR indexes. For examples of how to create a local index, see "Creating the local TARGET indexes" on page 165.
- Each STAR index can reside in multiple segments with the index entries distributed in one of the following ways:
  - Sequentially among segments, so each segment contains approximately the same number of entries

This round-robin segmentation is defined when you specify multiple segment names, but do not specify any of the three SEGMENT clauses shown in Table 4 on page 134 for the CREATE STAR INDEX statement.

– By ranges of values in the leading column of the index when the leading column of the index is the segmenting column of the table

The index entries in a STAR index are internal storage locations (row IDs) of the referenced column instead of the column values. Therefore, you specify row IDs to define the segment ranges of a STAR index. However, you can segment a STAR index like its fact table with the SEGMENT LIKE DATA clause in the CREATE STAR INDEX statement.

For information about the different SQL clauses to define this segmentation of a STAR index, see Table 4 on page 134.. For more information about the structure of a STAR index, see "Segmenting a STAR index" on page 135.

- Segments can span multiple PSUs.
- If segmented by range, the segment boundaries can be modified by attaching or detaching segments at either end of the range of an existing segment, but you cannot insert a new segment in the middle of the range of an existing segment. In other words, the modified range must use one of the existing boundary values.
- The range for any given segment can be changed (with an ALTER SEGMENT statement). The new range can either include all rows already in that segment or move to a range that is not adjacent to that segment.
- You cannot change the range or attach a segment if the operation causes data movement from one segment to another. For an index segment, you can choose to allow the change range or attach operation to proceed, but the index is invalidated.

How to segment

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. For information on when you would use each way to segment a table or index, see "Deciding how to segment" on page 90..

Table 4. Segment Specification and SQL Keywords

Contents of segments	How segmented	Keywords in CREATE TABLE or CREATE INDEX	Keywords in ALTER SEGMENT ATTACH
Row data	By data value of segmenting column	SEGMENT BY VALUES OF	RANGE (*literal*:*literal*)
	By hashing	SEGMENT BY HASH	Cannot attach segments by hash

Table 4. Segment Specification and SQL Keywords (continued)

Contents of segments	How segmented	Keywords in CREATE TABLE or CREATE INDEX	Keywords in ALTER SEGMENT ATTACH
STAR indexes	Evenly across segments (Red Brick Warehouse calculates ranges)	none (default)	none (Red Brick Warehouse calculates ranges for new segment)
	By row numbers of referenced table containing the segmented column of the STAR index	SEGMENT BY REFERENCES OF	RANGE (*rownum*:*rownum*)
	By row IDs of segmented referenced table containing the segmented column of the STAR index	SEGMENT LIKE REFERENCED TABLE	RANGE (*rowID*:*rowID*)
	By the same values as the data	SEGMENT LIKE DATA	RANGE LIKE SEGMENT
Primary index	By data values of first (leading) column in the primary index	SEGMENT BY VALUES OF SEGMENT LIKE DATA	RANGE (*literal*:*literal*) RANGE (*literal*:*literal*)
TARGET indexes	By values of the indexed column	SEGMENT BY VALUES OF SEGMENT LIKE DATA	RANGE (*literal*:*literal*) RANGE (*literal*:*literal*)
	By the same values as the data	SEGMENT LOCAL	RANGE LIKE SEGMENT
B-TREE indexes	By values of the first (leading) indexed column	SEGMENT BY VALUES OF SEGMENT LIKE DATA	RANGE (*literal*:*literal*) RANGE (*literal*:*literal*)
	By the same values as the data	SEGMENT LOCAL	RANGE LIKE SEGMENT

Segmenting a STAR index

You can segment a STAR index only by the values in its leading column. Because the values in a STAR index are the row IDs in the dimension tables, the Red Brick Warehouse uses row IDs to define the segment ranges of a STAR index. A row ID consists of the segment name and the row number of the data in a dimension table. If the dimension table is not segmented, you only need to specify the row number in the RANGE clause of the CREATE INDEX statement.

Row numbers are identifiers that Red Brick Warehouse assigns to each row as it is loaded or inserted into the table. If a newly created table has no VARCHAR columns, row numbers are assigned sequentially. Therefore, a row number does not necessarily correspond to the values in that row. To create a correspondence, ensure that the values you load into the dimension table are in sequence and match the ranges.

Use one of the following clauses in the CREATE STAR INDEX statement to specify the segment ranges:

- SEGMENT LIKE DATA clause when you want to segment a STAR index exactly like its fact table.

  Run a query that selects the primary key from the leading dimension table to verify that the rows have been loaded in the order that you expect. For details, see the procedure on page 164.
- SEGMENT BY REFERENCES OF and RANGES (*rownum:rownum)* clauses when the dimension table is not segmented.

  Run a query that selects the primary key, the RBW_ROWNUM and optionally the RBW_segNaMe pseudocolumns from the leading dimension table. The procedure on page 163 includes an example of this query. Use the results of this query to:

  – Verify that the rows have been loaded in the order that you expect.

  – Obtain the row number or row ID to use to segment the STAR index. Each segment includes the starting range value that you specify in the RANGES clause, but does not include the ending range value.

  For more information about pseudocolumns, see “Pseudocolumns” on page 278.
- SEGMENT LIKE REFERENCED TABLE clause when the dimension table is segmented.

In a typical STAR schema, the fact table is segmented by key values in the Period dimension table, and the STAR index is segmented by the corresponding row IDs of the Period table. One approach is to have one segment of the fact table and one corresponding STAR index segment for every group of consecutive rows in the Period table that consists of the dates for one quarter of a year. Figure 31 shows this segmentation approach.

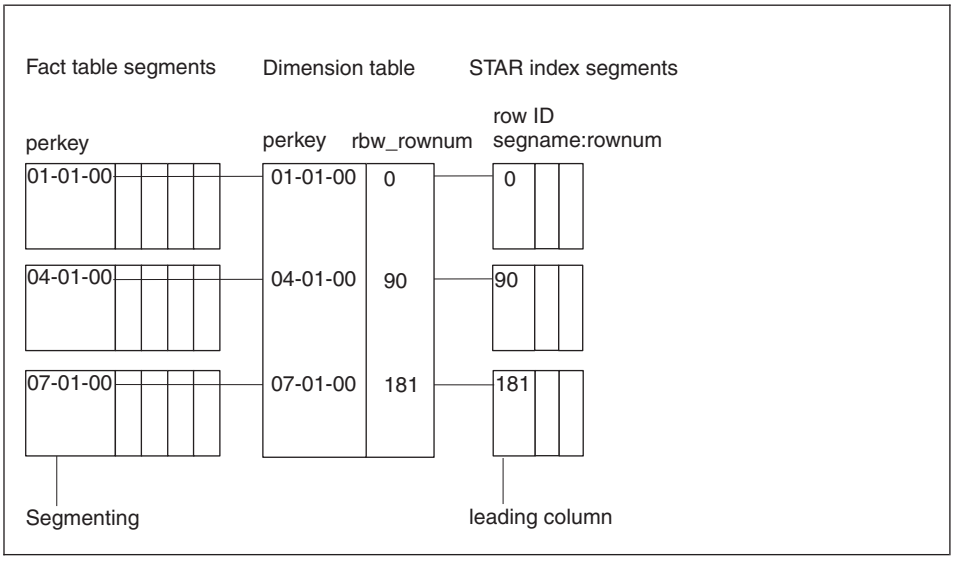


Figure 31. Contents of STAR index segments

Figure 31 shows the first parts of the fact table, dimension table, and STAR index.

- On the left is the fact table segmented by the perkey column. Each segment contains data for one quarter of a year. The first segment contains rows for perkey values `01-01-00` to `03-31-00`, the second segment contains rows for perkey values `04-01-00` to `06-30-00`, and so forth.
- In the center is the Period dimension table whose perkey column is used to create the segments of the fact table. A row for each date exists in the perkey column; however, for simplicity Figure 31 shows only the three values that define the starting range of each segment. The corresponding rbw_rownum is used to create the segments of the STAR index. For the detailed procedure, see "Creating the STAR index" on page 163.
- On the right is the STAR index whose leading column is the perkey column of the Period dimension table. The first perkey value 01-01-00 in the dimension table has a row number of `0`, and the dimension table resides in segment default_segment_15. Therefore, the first STAR index entry has the row ID value `default_segment_15:0`. Similarly, the perkey value 07-01-00 in the referenced table has a row number of 181, and the STAR index entry for this perkey is `default_segment_15:181`.

Segmenting a STAR index like its fact table enables it to be maintained quickly with the corresponding segments of the fact table. For more information, see "Planning for a time-cyclic database" on page 92 and "Maintaining a time-cyclic database" on page 294.

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 `rbw.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).

To set a default directory for default data and index segments for all sessions, enter a lines with the following format in the `rbw.config` file:

```
OPTION DEFAULT_DATA_SEGMENT dir_path
OPTION DEFAULT_INDEX_SEGMENT dir_path
```

The corresponding SQL SET statement has the following format:

```
SET DEFAULT DATA SEGMENT STORAGE PATH 'dir_path';
SET DEFAULT INDEX SEGMENT STORAGE PATH 'dir_path';
```

dir_path Path name 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 Only

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';
```

End of UNIX Only

Windows Only

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';
```

End of Windows Only

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.)

To specify segment drop behavior for user-defined segments for all sessions, enter a line with the following format in the rbw.config file.

```
OPTION SEGMENTS action
```

The corresponding SQL SET statement has the following format:

```
SET SEGMENTS action;
```

action Possible actions include:

- The keyword KEEP specifies that 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.
- The keyword DROP indicates that 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;
```

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, see "Altering Tables" on page 9-39. For more information on data types, see the *SQL Reference Guide*. For information on specifying fill factors for VARCHAR columns, see "Setting the VARCHAR column fill factor" on page 142 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

IBM 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 97 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, see "Setting the VARCHAR column fill factor" on page 142.

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, see 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.

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, see “Pseudocolumns” on page 278.

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, see on page 143.

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. on page 144 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, see 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. on page 145 illustrates this effect.

  For more information on this parameter, see "Setting the MAXSEGMENTS and MAXROWS PER SEGMENT options" on page 140.

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 over-estimating. 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 Name 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 on page 143 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, obtain the current fill factor value and use the CHECK TABLE statement with the VERBOSE option. If necessary, modify the fill factor value.

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
```

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 on page 143 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 142 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. See on page 144..

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. See on page 143.

```
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, see on page 145.
- 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, see the *SQL Reference Guide*.

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, see on page 144.
- 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, see on page 145.

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 on page 144. 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, see the *SQL Reference Guide*.

Creating indexes

The IBM Red Brick Warehouse automatically creates a B-TREE index on the primary key columns of a table at table creation. You can create additional indexes to improve query performance. As explained in "Indexes and retrieval strategies" on page 19, Red Brick server has three types of index technologies: STAR indexes, TARGET indexes, and B-TREE indexes.

You define each additional index in a CREATE INDEX statement with the type of index, a name, the table and column names, segment identifiers, and instructions for parallel index creation. For guidelines about when to create additional indexes and when to use the various types of indexes, see "Determining when to create additional indexes" on page 79.

For examples of creating indexes for a time-cyclic database, see "Creating the STAR index" on page 163 and "Creating the local TARGET indexes" on page 165. For further information on the CREATE INDEX statement, see 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.

If you create several indexes on a table, it is faster to create them in deferred mode and then use the Table Management Utility (TMU) REORG operation to build all the indexes at the same time. You might create multiple indexes at the same time in either of the following circumstances:

- When you create multiple TARGET indexes on the foreign key columns of a fact table for TARGET joins.
- After you have dropped indexes on a dimension table.

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 IBM Red Brick Warehouse 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, see “Estimating temporary space values for index-building operations” on page 112.

Parallel indexes

UNIX Only

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.

End of UNIX Only

Windows Only

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, see the *Table Management Utility Reference Guide*.

End of Windows Only

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, see "Index fill factors" on page 97..

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.

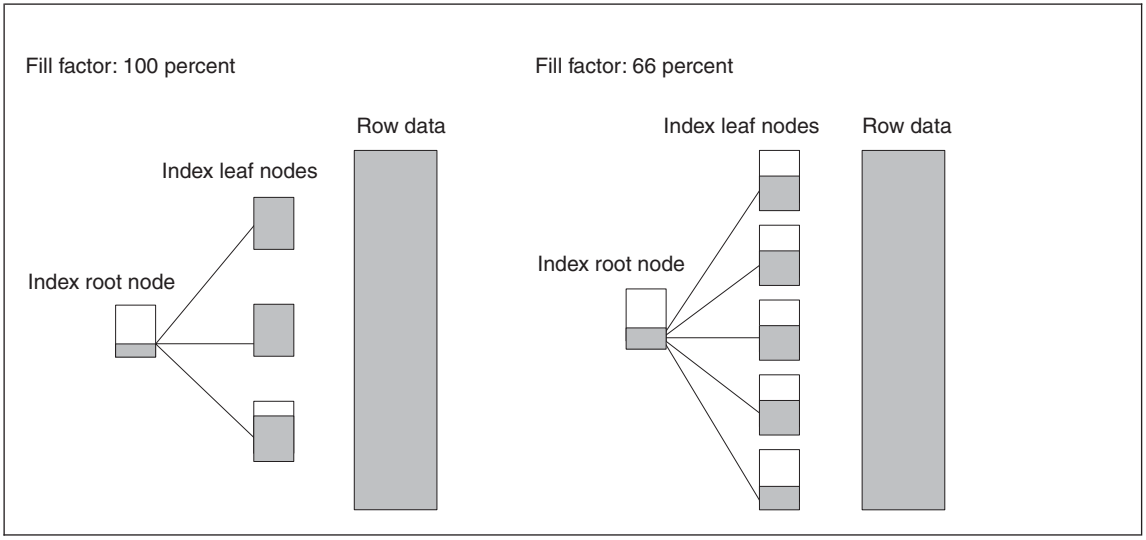


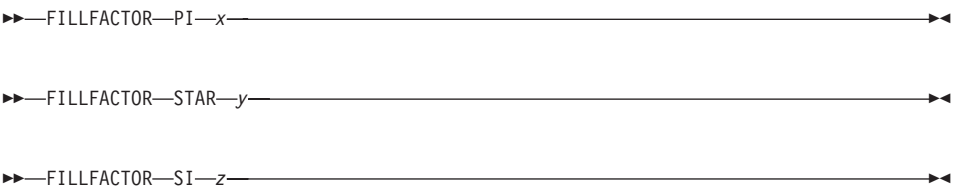
Figure 32. Fill factor and index nodes

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:

```
Elements per index node = (fillfactor/100) x (8172/[keysize+6])
```

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 97.. 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

Note: 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 a time-cyclic database

The goal in designing a time-cyclic database is to segment the indexes like the data so that the index segments can be rolled off quickly. To create a time-cyclic database that will be easy to maintain with the least amount of disruption to users, you segment the data by a time period value such as a day, week or month.

This example illustrates how to create multiple segments in a time-cyclic database. The example includes a simple star schema with a STAR index and local TARGET indexes on foreign keys. The indexes are segmented in the same way as the data.

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.
- The Sales table has local TARGET indexes on each foreign key (perkey, prodkey, mktkey). Local indexes provide the following advantages:
  - Because the Red Brick Warehouse knows that the local index segment corresponds exactly to the data segment, the index segment can be rolled off quickly when you roll off the data segment. For details, see "Rolling off and reusing data and index segments" on page 295.
  - Local indexes improve the performance of TARGETjoin and TARGETscan operations. For details, see the *Query Performance Guide*.
- 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.

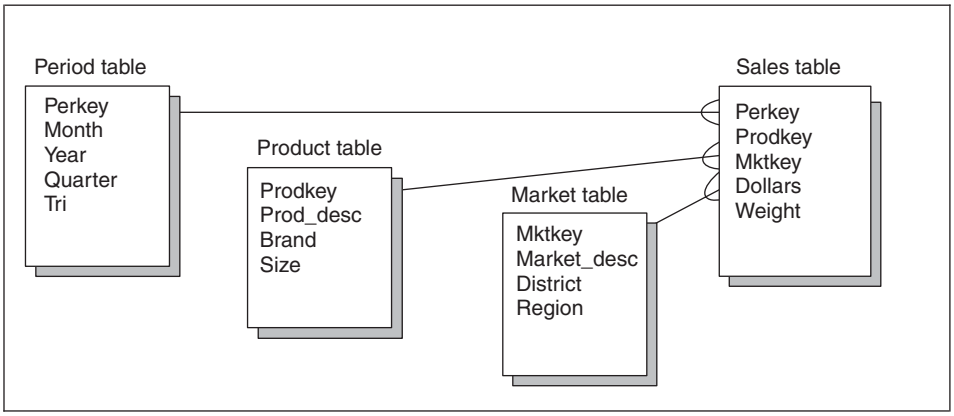


Figure 33. Schema example

The following sections describe how to create the segments, tables, and indexes for this time-cyclic database example. For details on how to maintain this time-cyclic database example, see page 294.

Creating the data and index segments

You must create the segments before you create the tables and indexes that use them. Figure 34, and Figure 35 show the CREATE SEGMENT statements using the path names in Windows NT or Windows 2000. For UNIX, the path names use forward slashes (instead of back slashes as in Windows), as the following example shows:

UNIX Only

```
create segment s_1q00
   storage '/disk1/s1' maxsize 200,
   storage '/disk1/s2' maxsize 200;
```

End of UNIX Only

The last segments in the table and each index (s_max, star_max, target_per_max, target_prod_max, and target_mkt_max) are empty segments that are placeholders at the high end of the range. When a segment is attached, its range overlaps an existing segment range and the existing segment is searched to verify that it does not contain data in the new range. An empty segment at the high end can be searched more quickly than a segment that contains actual data. While such a segment is not necessary, its

existence allows a new segment to be attached more quickly.

Windows Only

Figure 34 shows how to create the data segments for the Sales table on Windows.

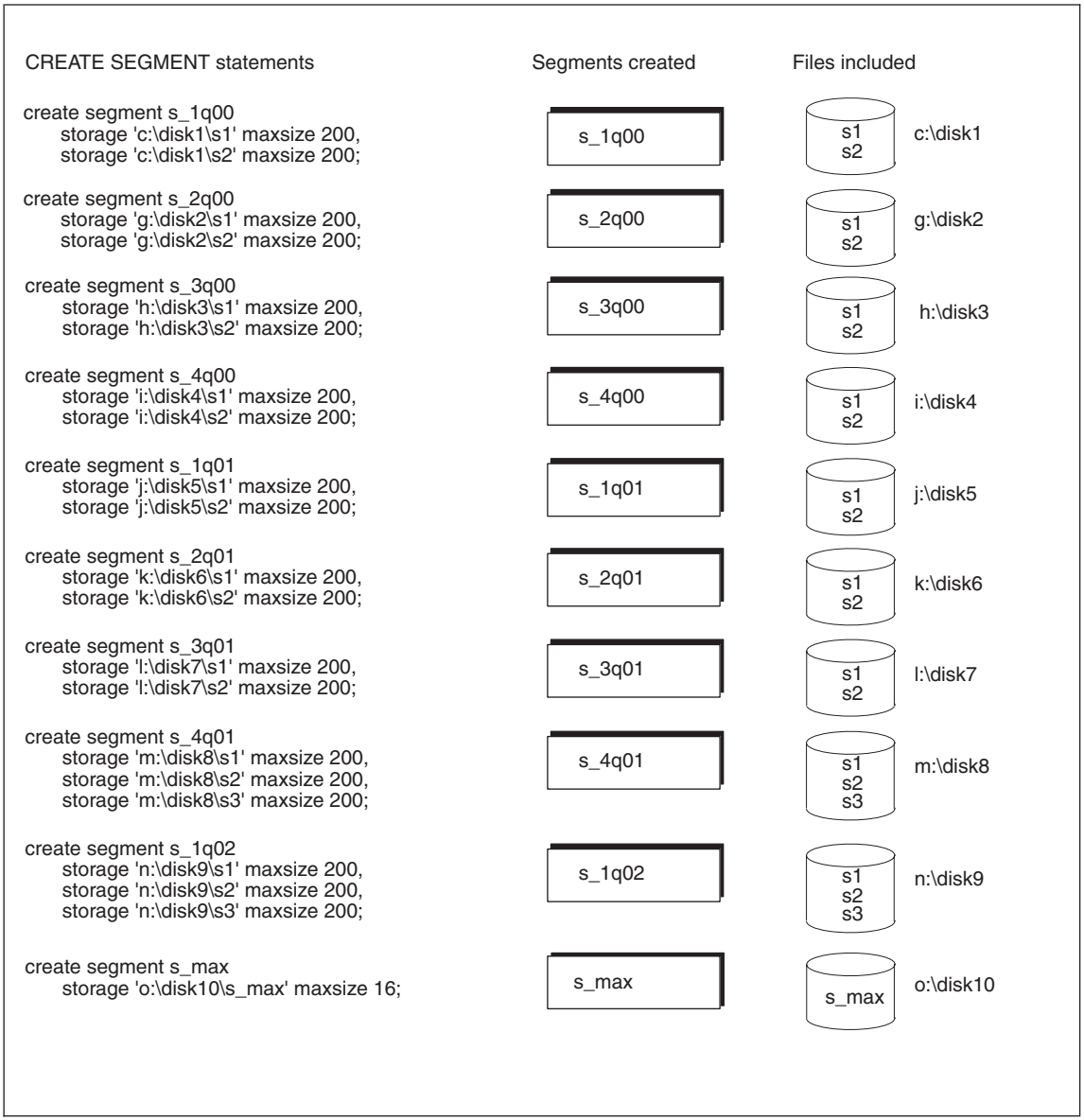


Figure 34. Sales table segments on Windows

Figure 35 shows how to create the index segments for the Sales_star index.

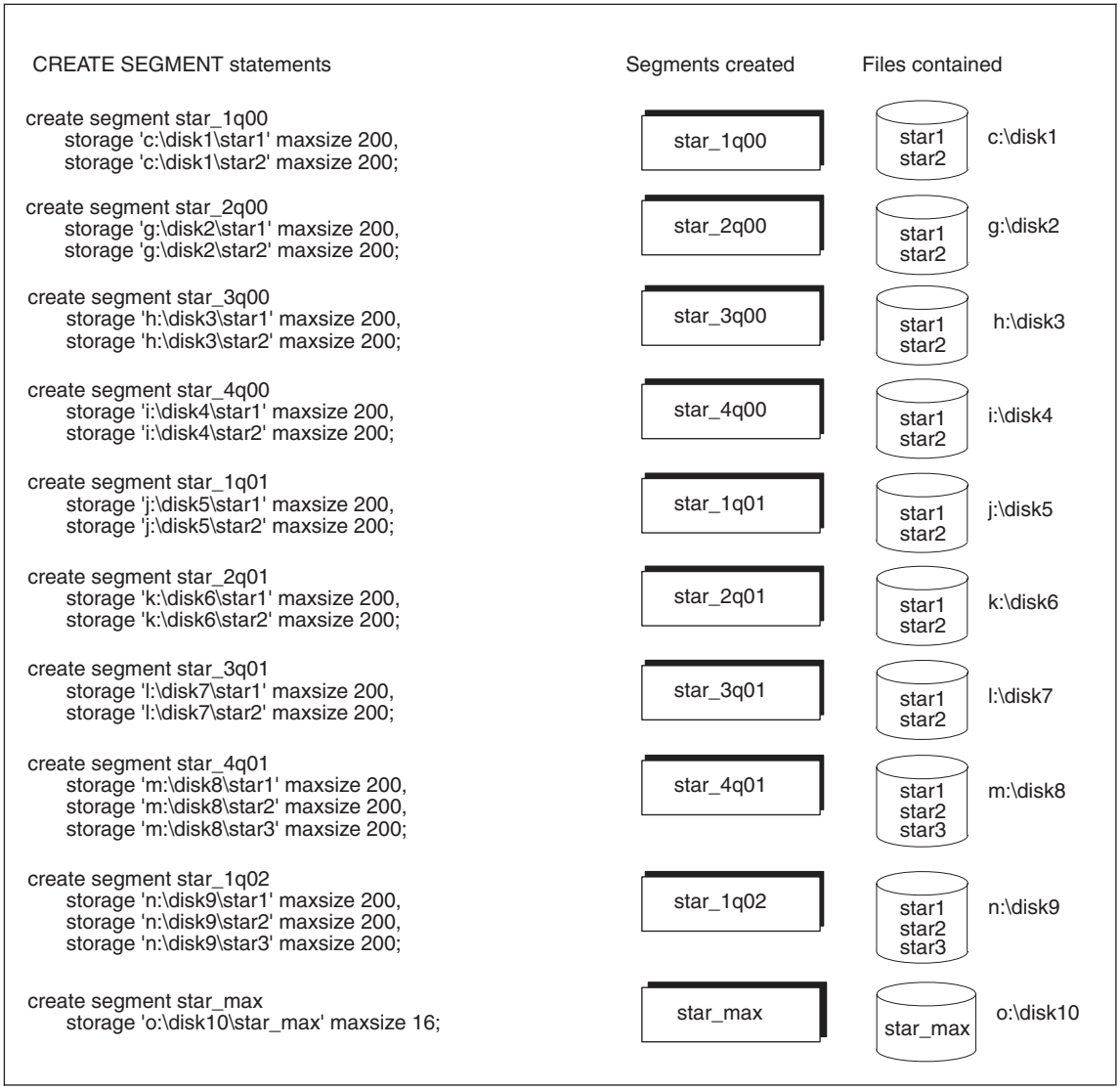


Figure 35. Sales_star index segments on Windows

Figure 36 shows how to create the index segments for one of the TARGET indexes, the Sales_target_mktkey index on the foreign key column mktkey.

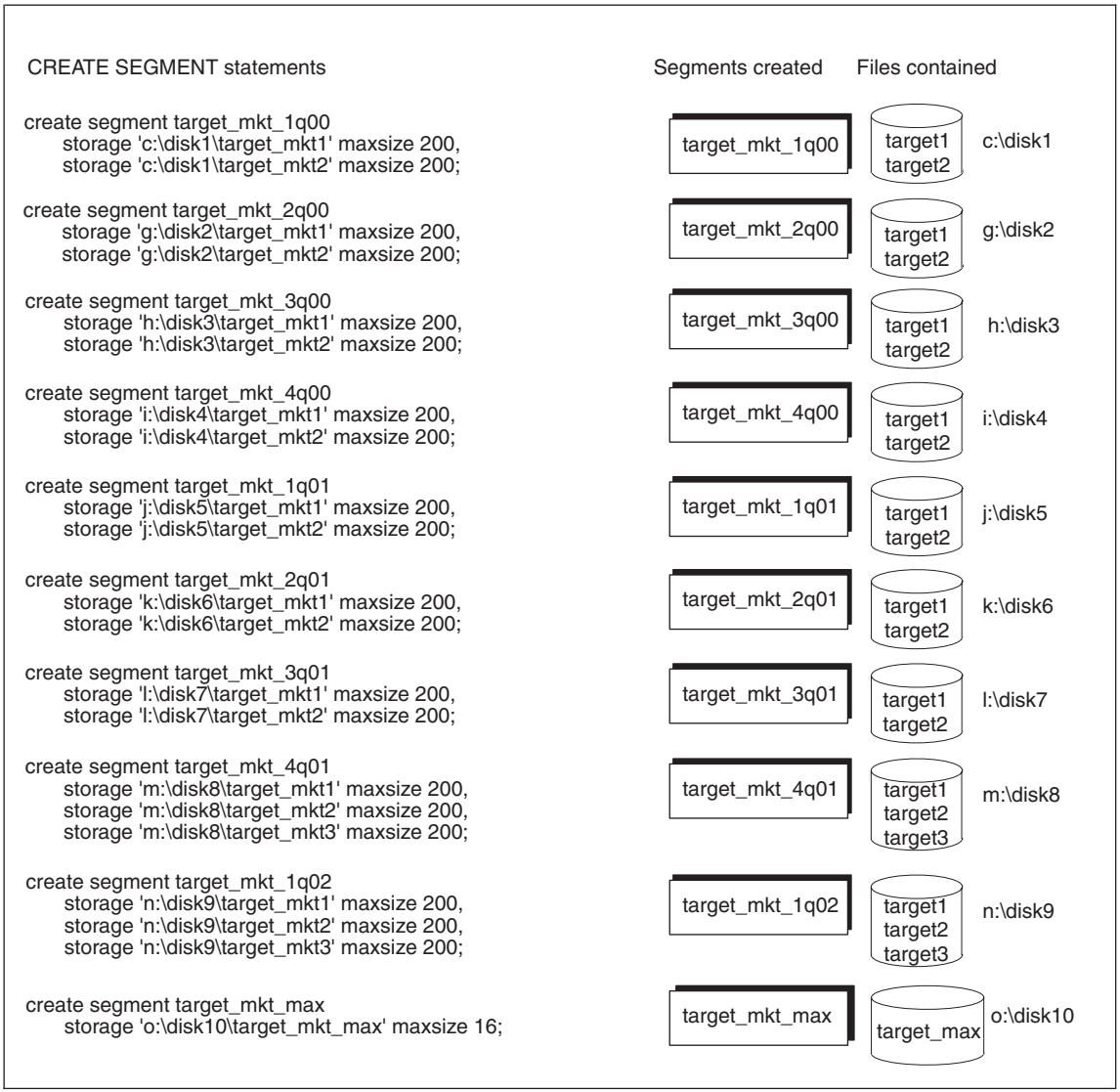


Figure 36. Creating Sales_target_mktkey index segments on Windows

The following CREATE SEGMENT statements define the other two local target indexes:

```
create segment target_per_1q00
   storage 'c:\disk1\target_per1' maxsize 200,
   storage 'c:\disk1\target_per2' maxsize 200;

create segment target_per_2q00
   storage 'g:\disk2\target_per1' maxsize 200,
   storage 'g:\disk2\target_per2' maxsize 200;
```

```
create segment target_per_3q00
   storage 'h:\disk3\target_per1' maxsize 200,
   storage 'h:\disk3\target_per2' maxsize 200;

create segment target_per_4q00
   storage 'i:\disk4\target_per1' maxsize 200,
   storage 'i:\disk4\target_per2' maxsize 200;

create segment target_per_1q01
   storage 'j:\disk5\target_per1' maxsize 200,
   storage 'j:\disk5\target_per2' maxsize 200;

create segment target_per_2q01
   storage 'k:\disk6\target_per1' maxsize 200,
   storage 'k:\disk6\target_per2' maxsize 200;

create segment target_per_3q01
   storage 'l:\disk7\target_per1' maxsize 200,
   storage 'l:\disk7\target_per2' maxsize 200;

create segment target_per_4q01
   storage 'm:\disk8\target_per1' maxsize 200,
   storage 'm:\disk8\target_per2' maxsize 200,
   storage 'm:\disk8\target_per3' maxsize 200;

create segment target_per_1q02
   storage 'n:\disk9\target_per1' maxsize 200,
   storage 'n:\disk9\target_per2' maxsize 200,
   storage 'n:\disk9\target_per3' maxsize 200;

create segment target_per_max
   storage 'o:\disk10\target_per_max' maxsize 16;

create segment target_prod_1q00
   storage 'c:\disk1\target_prod1' maxsize 200,
   storage 'c:\disk1\target_prod2' maxsize 200;

create segment target_prod_2q00
   storage 'g:\disk2\target_prod1' maxsize 200,
   storage 'g:\disk2\target_prod2' maxsize 200;

create segment target_prod_3q00
   storage 'h:\disk3\target_prod1' maxsize 200,
   storage 'h:\disk3\target_prod2' maxsize 200;

create segment target_prod_4q00
   storage 'i:\disk4\target_prod1' maxsize 200,
   storage 'i:\disk4\target_prod2' maxsize 200;

create segment target_prod_1q01
   storage 'j:\disk5\target_prod1' maxsize 200,
   storage 'j:\disk5\target_prod2' maxsize 200;

create segment target_prod_2q01
   storage 'k:\disk6\target_prod1' maxsize 200,
```

```
   storage 'k:\disk6\target_prod2' maxsize 200;

create segment target_prod_3q01
   storage 'l:\disk7\target_prod1' maxsize 200,
   storage 'l:\disk7\target_prod2' maxsize 200;

create segment target_prod_4q01
   storage 'm:\disk8\target_prod1' maxsize 200,
   storage 'm:\disk8\target_prod2' maxsize 200,
   storage 'm:\disk8\target_prod3' maxsize 200;

create segment target_prod_1q02
   storage 'n:\disk9\target_prod1' maxsize 200,
   storage 'n:\disk9\target_prod2' maxsize 200,
   storage 'n:\disk9\target_prod3' maxsize 200;

create segment target_prod_max
   storage 'o:\disk10\target_prod_max' maxsize 16;
```

End of Windows Only

Creating 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 (5),
   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_1q00, s_2q00, s_3q00, s_4q00, s_1q01, s_2q01,
      s_3q01, s_4q01, s_1q02, s_max)
```

```
segment by values of (perkey) ranges (
   min: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':DATE'2001-07-01',
   DATE'2001-07-01':DATE'2001-10-01',
   DATE'2001-10-01':DATE'2002-01-01',
   DATE'2002-01-01':DATE'2002-04-01',
   DATE'2002-04-01': max);
```

Creating the STAR index

Suppose you want to create a STAR index covering all of the foreign keys on the Sales table and the STAR index is to be segmented *exactly* like the data; that is, the index entries corresponding to each row of data in the s_1q00 data segment are in the star_1q00 index segment, the index entries corresponding to each row of data in the s_2q00 data segment are in the star_2q00 index segment, and so on.

You can accomplish this STAR index segmentation in one of two ways:

- Segment by range of row IDs of the leading dimension table (Period) of the STAR index
- Segment like the fact table because the leading column Perkey of the STAR index is the segmenting column of the fact table Sales

Tip: Segmenting the STAR INDEX like the fact table is easier than segmenting by row IDs in the leading dimension table Period. Procedures for both ways are below.

Index entries in a STAR index are not the values of the indexed data but the row IDs of the dimension (referenced) table containing the data. For details of this row ID structure and how it relates to the fact and dimension table, see "Segmenting a STAR index" on page 135.

To segment the STAR index by range of row IDs of the leading dimension table:

1. Load the leading dimension table before you create the STAR index segmented like the data.

   The leading dimension table must be loaded before you create the STAR index because you must specify the row IDs in the dimension table to define the segment ranges in the CREATE STAR INDEX statement.
2. Find the boundaries for the STAR index segment ranges

   Query the RBW_ROWNUM pseudocolumn and the primary key column (Date) of the leading dimension table Period. 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 on page 162.

```
select rbw_rownum, date
   from period
   where date = '04-01-00'
      or date = '07-01-00'
      or date = '10-01-00'
      or date = '01-01-01'
      or date = '04-01-01'
      or date = '07-01-01'
      or date = '10-01-01'
      or date = '01-01-02'
      or date = '04-01-02';
```

This query produces the following rows:

```
RBW_ROWNUM  DATE
        90 2000-04-01
       181 2000-07-01
       273 2000-10-01
       365 2001-01-01
       455 2001-04-01
       546 2001-07-01
       638 2001-10-01
       730 2002-01-01
       821 2002-04-01
```

Tip: The ranges in the CREATE STAR INDEX statement see row numbers in the referenced table Period, not row numbers in the Sales table. For the complete syntax of the CREATE INDEX statement, see the "*SQL Reference Guide*."

As the Period table is typically the smallest table in the database, it is notd usually segmented. Therefore, you do not need to query the RBW_SEGNAME pseudocolumn.

3. 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_1q00, star_2q00, star_3q00, star_4q00,
      star_1q01, star_2q01, star_3q01, star_4q01,
       star_1q02, 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)
```

To segment the STAR index like the fact table:

1. Load the leading dimension table before you create the STAR index segmented like the data

The leading dimension table must be loaded before you create the STAR index because the create index operation still obtains the or row numbers or row IDs from the dimension table to define the segment boundaries, even though you do not need to specify them in the CREATE STAR INDEX statement.

The STAR index is still physically segmented by row IDs of the dimension table, but it is also logically segmented by the value of the segmenting column of the fact table.

2. Query the leading dimension table to ensure that the rows have been loaded in the order that you expect.

   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 on page 162.

```
select date from period
   where date = '04-01-00'
      or date = '07-01-00'
      or date = '10-01-00'
      or date = '01-01-01'
      or date = '04-01-01'
      or date = '07-01-01'
      or date = '10-01-01'
      or date = '01-01-02'
      or date = '04-01-02';
```

   This query produces the following rows:

```
DATE
90 2000-04-01
181 2000-07-01
273 2000-10-01
365 2001-01-01
455 2001-04-01
546 2001-07-01
638 2001-10-01
730 2002-01-01
821 2002-04-01
```

3. Specify the SEGMENT LIKE DATA clause in the CREATE STAR INDEX statement

```
create star index sales_star
      on sales (perkey, prodkey, mktkey)
      in (star_1q00, star_2q00, star_3q00, star_4q00,
          star_1q01, star_2q01, star_3q01, star_4q01,
          star_1q02, star_max)
      segment like data
```

Creating the local TARGET indexes

Suppose you want to create TARGET indexes on each foreign key in the Sales table and that the TARGET indexes are segmented *exactly* like the data. For example, the index entries corresponding to each row of data in the s_1q00 data segment are in the target_1q00 index segment, the index entries

corresponding to each row of data in the s_2q00 data segment are in the target_2q00 index segment, and so on. To accomplish this segmentation, use the SEGMENT LOCAL clause in the CREATE INDEX statement:

```
create target index sales_target_mktkey
      on sales (mktkey)
      in (target_mkt_1q00, target_mkt_2q00, target_mkt_3q00,
         target_mkt_4q00, target_mkt_1q01, target_mkt_2q01,
         target_mkt_3q01, target_mkt_4q01, target_mkt_1q02,
         target_mkt_max)
      segment local
```

You must use the SEGMENT LOCAL clause instead of the SEGMENT LIKE DATA clause because the segmenting column of the table is not the column of this TARGET index. A local TARGET index is easier to maintain in a time-cyclic database than a non-local TARGET index. For details, see "Rolling off and reusing data and index segments" on page 295.

The following table illustrates how sales data is mapped to the segments of the table, STAR index, and one of the target indexes.

Table 5. Mapping sales data to segments of a table

Sales data for these ranges	Mapped to these table segments	Mapped to these store index segments	Mapped to these promo index segments
min:DATE'2000-04-01' 20,000,000 rows	s_1q00	star_1q00	target_1q00
DATE'2000-04-01':DATE'2000-07-01' 30,000,000 rows	s_2q00	star_2q00	target_2q00
DATE'2000-07-01':DATE'2000-10-01' 40,000,000 rows	s_3q00	star_3q00	target_3q00
DATE'2000-10-01':DATE'2001-01-01' 50,000,000 rows	s_4q00	star_4q00	target_4q00
DATE'2001-01-01':DATE'2001-04-01' 60,000,000 rows	s_1q01	star_1q01	target_1q01
DATE'2001-04-01':DATE'2001-07-01' 50,000,000 rows	s_2q01	star_2q01	target_2q01
DATE'2001-07-01':DATE'2001-04-10' 60,000,000 rows	s_3q01	star_3q01	target_3q01
DATE'2001-10-01':DATE'2002-01-01' 70,000,000 rows	s_4q01	star_4q01	target_4q01
DATE'2002-01-01':DATE'2002-04-01' 40,000,000 rows	s_1q02	star_1q02	target_1q02
DATE'2002-04-01':max 10,000,000 rows	s_max	star_max	target_max

Each segment includes the lower end of its range but excludes the upper end. For example, the segment s_2q00 includes the data for 2000-04-01 but does not include data for 2000-07-01. Similarly, the STAR index segment star_2q00 and the TARGET index segment target_2q00 include the index entries for 2000-04-01 data but do not include index entries for 2000-07-01 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,
   weight smallint 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.

Figure 37 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. The STAR and TARGET indexes for the Sales table are segmented like the data, as shown in the table on page 166.

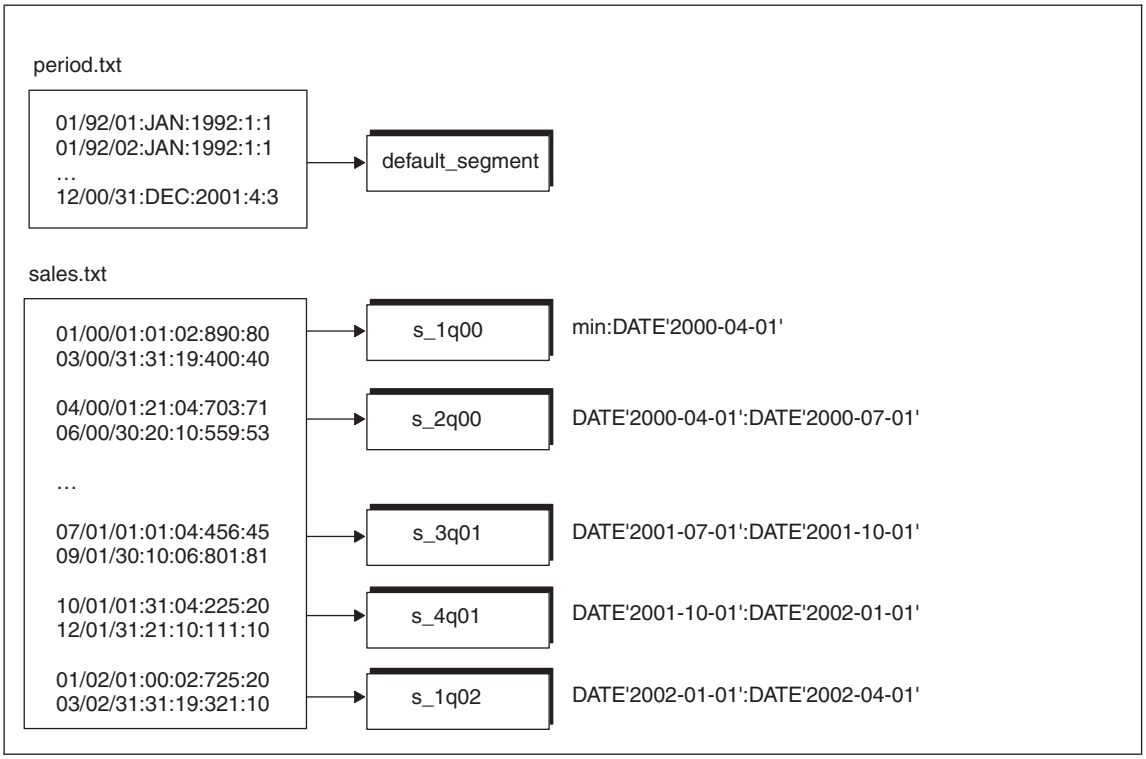


Figure 37. Data format for period and sales tables

When the next quarter (2Q02) arrives, it will be time to roll off the segment containing the oldest quarter (1Q00) and add data for this latest quarter. For details, see "Maintaining a time-cyclic database" on page 294.

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, see the *IBM Red Brick 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.

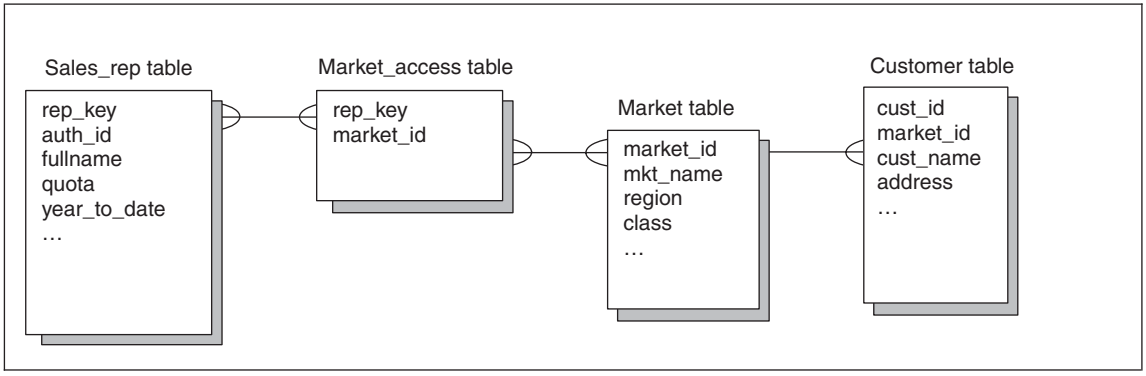


Figure 38. View example

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 IBM Red Brick Warehouse 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, see the *SQL Reference Guide*.

Availability and scope

The way in which a macro is defined determines its availability and scope. IBM Red Brick Warehouse 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 IBM Red Brick Warehouse. For information about macro use by a particular tool, see specific IBM 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, see the *SQL Self–Study Guide*.

Chapter 6. Working with a versioned database

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

© Copyright IBM Corp. 1996, 2004

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, see 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 185.

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, see the *IBM Red Brick 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, see "Creating the version log" on page 187.

- 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.

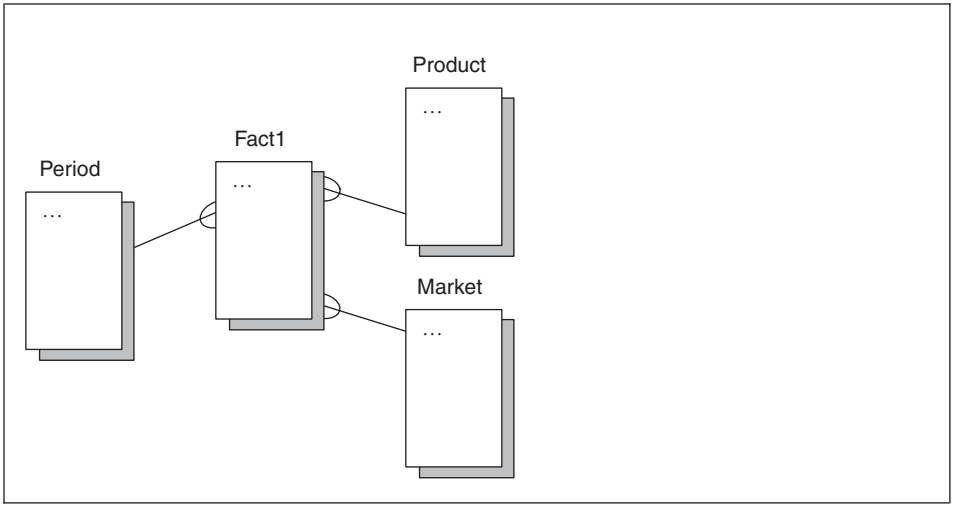


Figure 39. Simple star schema

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 176. You can also freeze the version available to users, as described in “Understanding frozen versions” on page 183. 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 185.

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, see "Vacuum cleaner daemon" on page 9.

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 IBM Red Brick Warehouse uses 8 kilobytes of disk space. The following figure shows a database with five blocks of data.

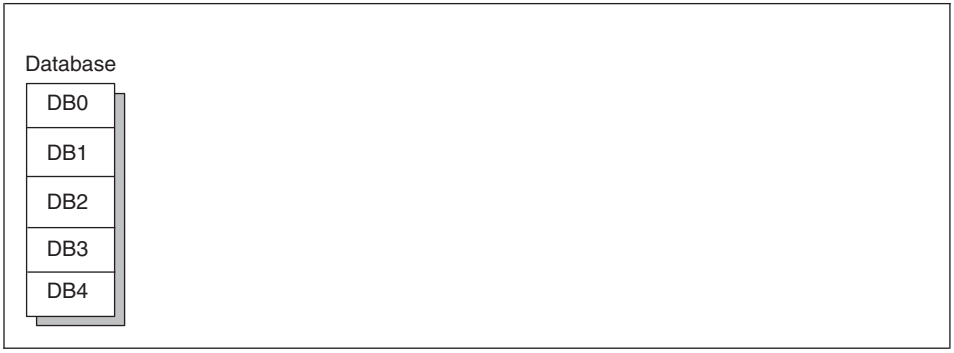


Figure 40. Database blocks

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.

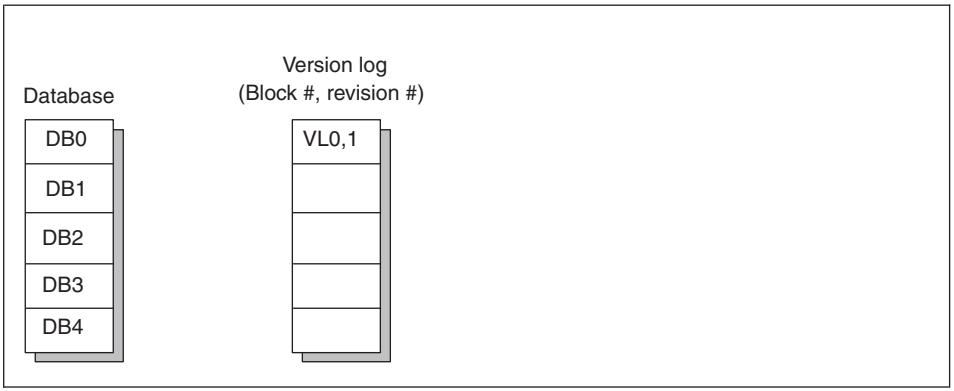


Figure 41. Version log after first transaction

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.

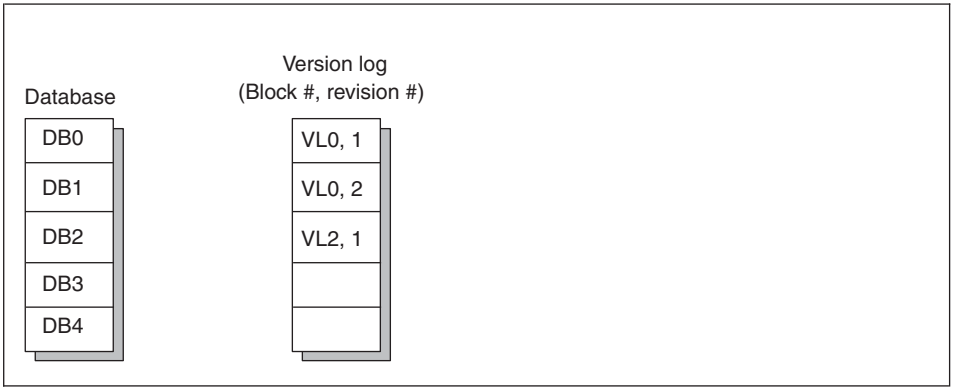


Figure 42. Version log after second transaction

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, see "Controlling the vacuum cleaner" on page 192.

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.

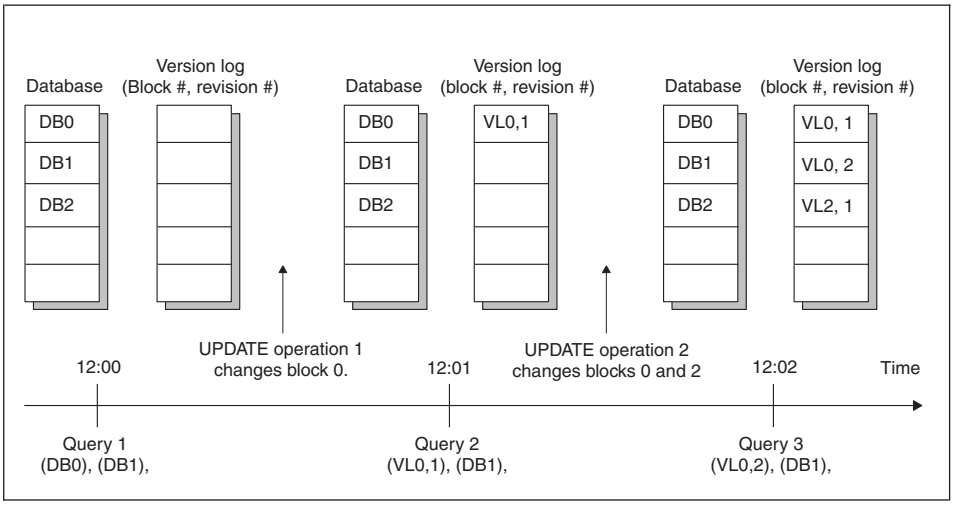


Figure 43. Query behavior without frozen versions

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, see “Controlling frozen versions” on page 189.

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, IBM 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. IBM recommends that you set up a realistic test database and measure it directly. For information on determining the size of your version log, see “Monitoring the version log” on page 191.

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 checkpoint 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 191.

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 188.

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, see "Dropping the version log and adding space" on page 188. For information on configuration, see "Creating the version log" on page 187.

To monitor the space being used by the version log, see 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, see "Monitoring the version log."

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, see Chapter 8, "Managing database activity in an enterprise," on page 225.

Backup and recovery

The version log should be empty before you perform a checkpoint backup, a partial or full restore operation, or when you copy or move a versioned database. For online backups, the version log does not need to be empty.

With versioning for recoverability, perform a checkpoint backup after loading the data. With versioning for concurrency, perform a checkpoint backup after completing the following procedure. For these backups, follow the appropriate backup strategy to ensure a successful restore. For more information on TMU guidelines and backup strategies, see the *Table Management Utility Reference Guide*.

To prepare for checkpoint backup when using versioning for concurrency:

1. Generate the list of PSUs to back up.

   ```
   select used, physical_location from rbw_storage;
   ```

   This query returns the list of PSUs and their sizes.

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 256..
3. Clean the version log.

   ```
   alter database clean version log;
   ```

   For more information, see “Cleaning the version log” on page 188.
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 IBM Customer Support. For instructions on dropping the version log, see “Dropping the version log and adding space” on page 188.

For details on backup and restore operations, see the *Table Management Utility Reference Guide*.

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 IBM Red Brick Warehouse. 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," on page 343.
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

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 theRISQL 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 Only

After IBM Red Brick Warehouse 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.

© Copyright IBM Corp. 1996, 2004

- 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
...
```

End of UNIX Only

Windows Only

After IBM Red Brick Warehouse 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 IBM Red Brick Warehouse 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.

End of Windows Only

For more information about the environment variables, see Chapter 2, "Key concepts," on page 17.

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, see 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 198, "Granting database object privileges" on page 200, and "Granting access with role-based security" on page 201.

To drop database users from a database, use the REVOKE CONNECT statement.

For information about the GRANT, REVOKE, and CREATE ROLE statements, see 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, see "Administering password security" on page 214.

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 IBM Red Brick Warehouse 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
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

For information about using role-based security to break down the tasks of the system roles and recombine them in new roles, see "Granting access with role-based security" on page 201.

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, see 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
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.

For a complete discussion of the GRANT and REVOKE statements, see 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 201.
- Object privileges, as defined in "Granting database object privileges" on page 200.
- 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, see the *IBM Red Brick 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, see "Monitoring database activity with dynamic statistic tables" on page 227..
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.

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.

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, see 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, see 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, see "Granting database object privileges" on page 200.

For complete syntax of the GRANT privilege statement, see 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.

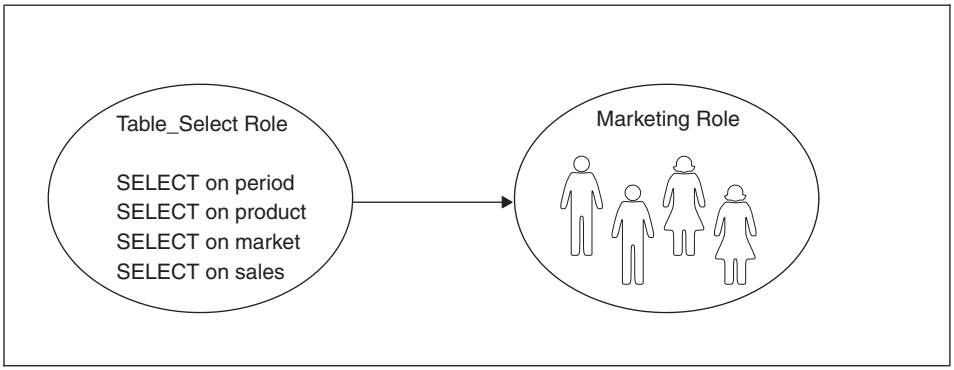


Figure 44. Indirect granting of table_select role

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 212.

For complete syntax of the GRANT authorization and role statement, see 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.

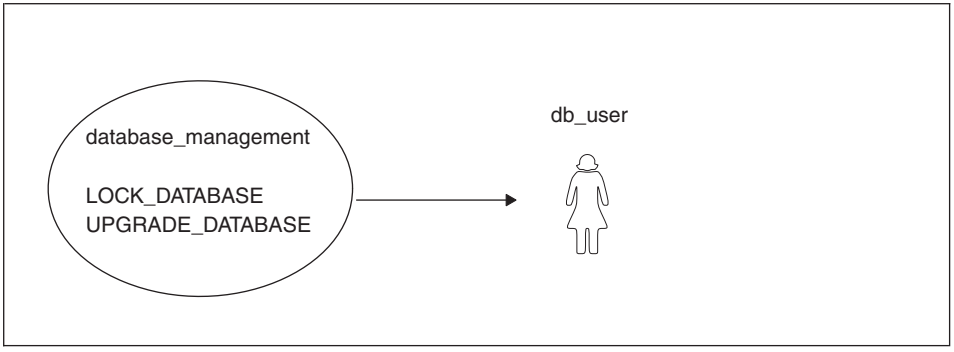


Figure 45. Indirect membership in role

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.

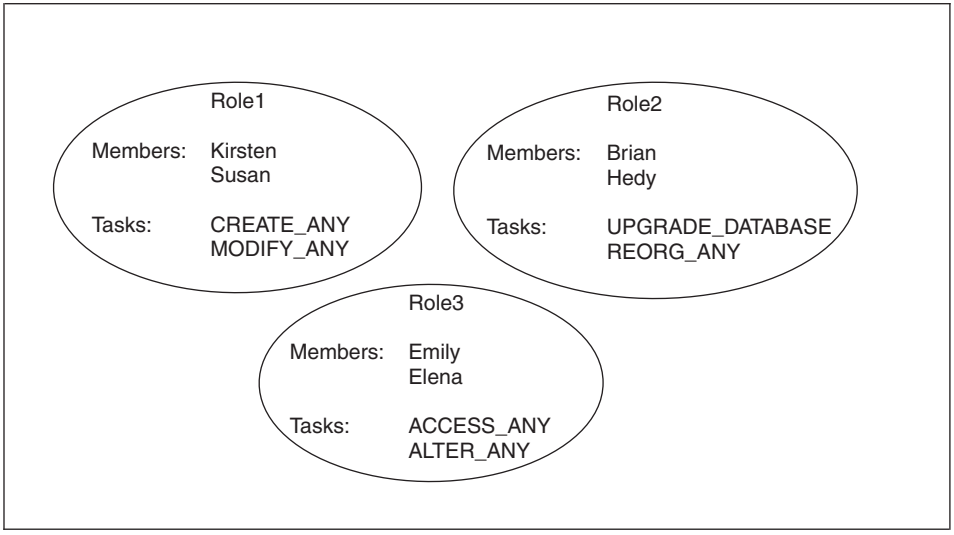


Figure 46. Role members and task authorizations

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.

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, see 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, see "Roles" on page 335. For complete syntax of the DROP ROLE statement, see 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 ...
CHRIS          N    N    R    R    R    N    N    N    N    N    N
CODY           N    N    N    N    N    I    I    I    I    I    I
DATABASE_M     N    N    N    N    N    N    N    N    N    N    N
JOE            N    N    N    N    N    N    N    N    N    N    N
JOHN           N    N    N    N    N    N    N    N    N    N    N
JUDY           N    N    R    R    R    N    N    N    N    N    N
MARIA          N    N    N    N    N    N    N    N    N    N    N
MARKETING      N    N    N    N    N    R    R    R    R    R    R
NASI           N    N    N    N    N    I    I    I    I    I    I
OBJECT_MAN     N    N    N    N    N    Y    Y    Y    Y    Y    Y
SECURITY_M     N    N    Y    Y    Y    N    N    N    N    N    N
SUDHIR         N    N    N    N    N    I    I    I    I    I    I
SYSTEM         Y    N    Y    Y    Y    Y    Y    Y    Y    Y    Y
TABLE_SELE     N    N    N    N    N    N    N    N    N    N    N
```

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, see the *SQL Reference Guide*.

The following table lists the password parameters and describes their functions.

PASSWORD parameter **Description**

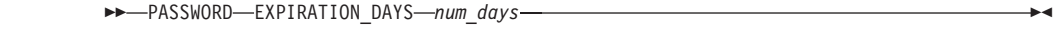
EXPIRATION_DAYS	Maximum number of days passwords exist. Forces users to change their passwords regularly.
EXPIRATION_WARNING_DAYS	The number of days prior to password expiration that the user receives a warning message.
RESTRICT_PREVIOUS	Minimum number of password changes that must occur before a password 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.
ALLOW_LOGNAME	Allow password to include the user's login name.
CHANGE_INITIAL	The user must change the initial password when the user logs in for the first time.
COMPLEX_NUMERIC	Changed password must differ from previous passwords by more than the value of a single numeric.
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, see "Limiting frequency of password changes" on page 218.
- 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:

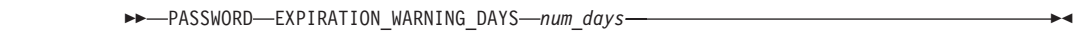
```
select grantee, expired
from rbw_userauth
where grantee = 'user_name' ;
```

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:

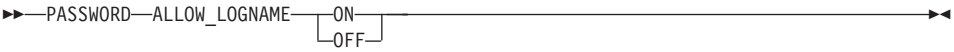
Assume the `rbw.config` file contains the following entries:

```
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.

Restricting use of login name in password

You can restrict users from using their login name as part of their passwords by setting the PASSWORD ALLOW_LOGNAME parameter in the `rbw.config` file. Set this parameter with the following syntax:



Requiring user to change initial password at first login

You can force users to change their initial passwords when they first log in by setting the PASSWORD CHANGE_INITIAL parameter in the `rbw.config` file. Set this parameter with the following syntax:

Change this parameter to ON if you want users to change their initial password right away.

Enforcing differences from previous passwords

You can prevent users from changing only the numeric portion of previous passwords by setting the PASSWORD COMPLEX_NUMERIC and PASSWORD RESTRICT_PREVIOUS parameters.

The PASSWORD COMPLEX_NUMERIC parameter determines if changed passwords must differ from previous passwords by more than the value of one or more numerics. Set this parameter in the `rbw.config` file with the following syntax:

You must set this parameter in combination with the PASSWORD RESTRICT_PREVIOUS parameter to ensure that users change more than just one or more numerics to another numeric in their previous passwords. If you do not specify the RESTRICT_PREVIOUS parameter, it defaults to a value of 0, which indicates no restrictions for re-creating old passwords.

The PASSWORD COMPLEX_NUMERIC 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.

Examples:

Suppose the rbw.config file contains the following entry:

```
PASSWORD ALLOW_LOGNAME OFF
```

Users must not use their login name in their passwords. Therefore, the following GRANT CONNECT statement returns an error message when issued by owen:

```
grant connect to owen with 'pw2owen1' ;
** ERROR ** (1043) New password for user owen must not contain the login name.
```

Suppose the rbw.config file contains the following entries:

```
PASSWORD RESTRICT_PREVIOUS 4
PASSWORD COMPLEX_NUMERIC ON
```

Suppose also that user1 and user2 execute the following GRANT CONNECT statements to create valid passwords:

```
grant connect to user1 with 'sample1' ;
grant connect to user2 with 'other1samp2' ;
```

When users change their passwords, the passwords must differ by more than any numeric replaced with a different numeric. Therefore, the following GRANT CONNECT statements return error messages when issued by user1 and user2, respectively, because only the numeric portion of the previous passwords was changed to form the new passwords:

```
grant connect to user1 with 'sample3' ;
grant connect to user2 with 'other2samp4' ;
```

If the DBA executes the above GRANT statements, they complete successfully.

Enforcing password complexity and length

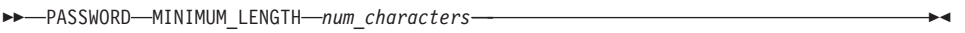
You can force users to create complex and secure passwords by setting any combination of the PASSWORD MINIMUM_LENGTH parameter and the complexity parameters.

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 1 to 128. The default is 1.

Note: Database passwords should not exceed eight characters.

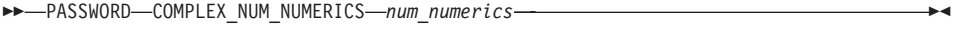
Syntax: COMPLEX_NUM_ALPHA

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.

Note: Database passwords should not use the semicolon character (;).

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 8
```

When users create new passwords, the passwords must have at least 8 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 'db12fu$%' ;
grant connect to james with 'sq67%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

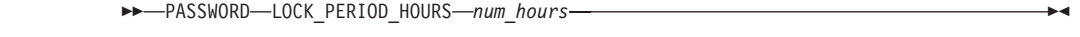
range of 0 to 128. The default is 0, which indicates no restriction on the number of incorrect entries.

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

This chapter describes the tasks involved in monitoring, controlling, and tracking resource use of IBM Red Brick Warehouse. 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 Warehouse 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, see 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.

© Copyright IBM Corp. 1996, 2004

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, see Chapter 7, "Providing database access and security," on page 195.

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 IBM Red Brick Warehouse with the installation script, the script asks whether you want to create the administration database. IBM 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.

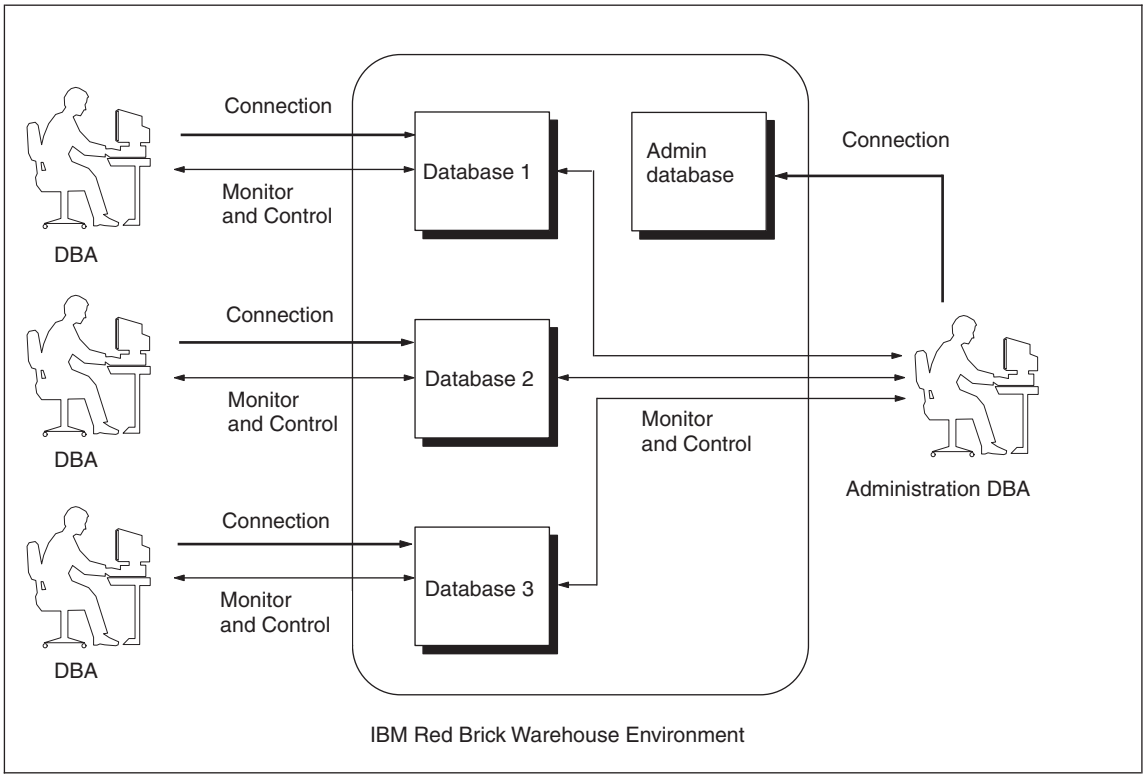


Figure 47. Role of administration database

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, see 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, see "Administration daemon process" on page 232.

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

The database server increments the cache read statistic each time a block is found in the local buffer cache when the session performs a read or write operation. For the case when a session writes data, the database server must read the data on the block in the cache before it can do the write. Therefore, the cache read statistic is also incremented for a write operation.

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.

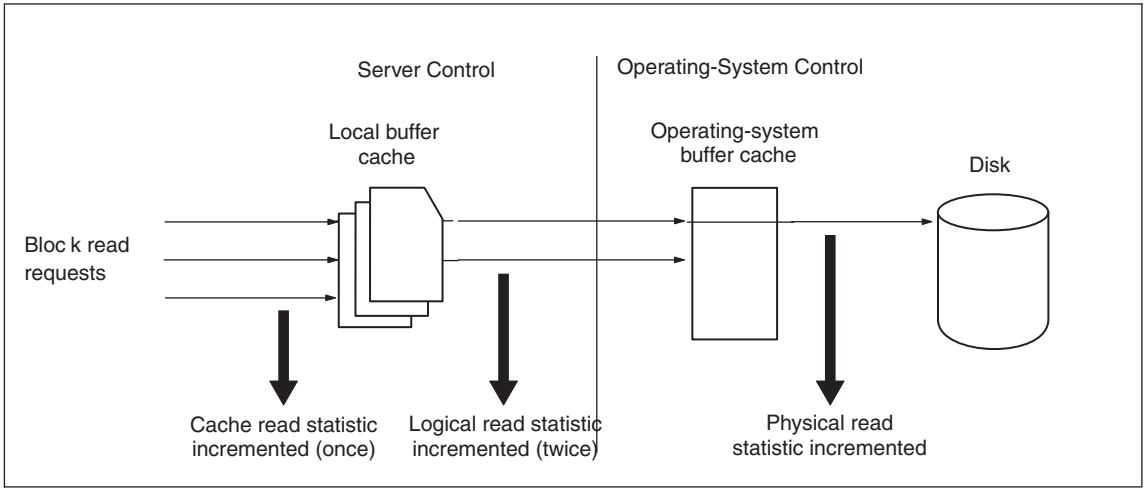


Figure 48. How read statistics are generated

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, see 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 Only

Your platform must have the UNIX **renice** command in order to support user priorities. You must specify the full path name of the `renice` executable file with the ADMIN RENICE_COMMAND configuration parameter. The full path name includes the directory name, but not the name of the executable.

End of UNIX Only

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, see "Monitoring and controlling a database server on UNIX" on page 330 or "Monitoring and controlling a database server on Windows" on page 332.

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.

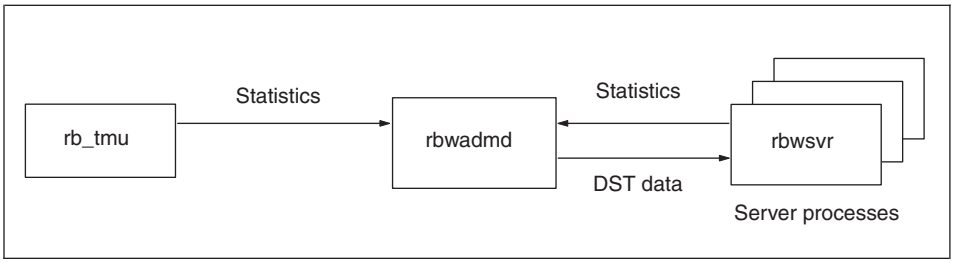


Figure 49. Data flow between rbwadmd and other database processes

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.

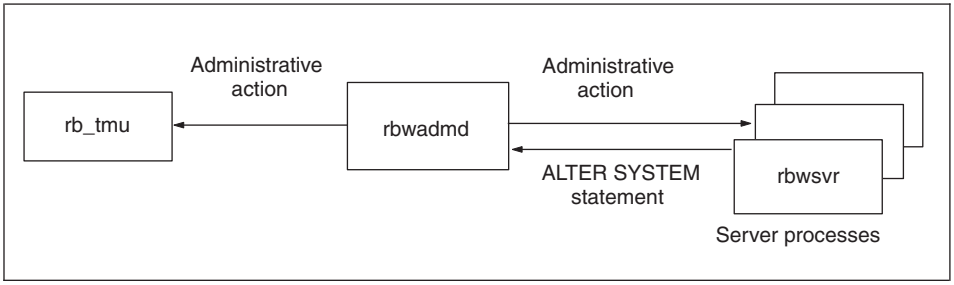


Figure 50. ALTER SYSTEM flow of control

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 Warehouse 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 IBM Red Brick Warehouse 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 Warehouse 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 Warehouse 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.

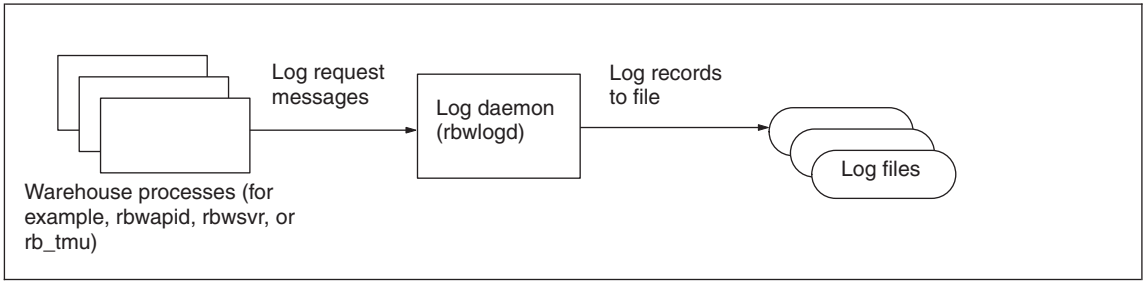


Figure 51. Log daemon

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, see "Log files" on page 241.

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, see "Event log messages" on page 239.

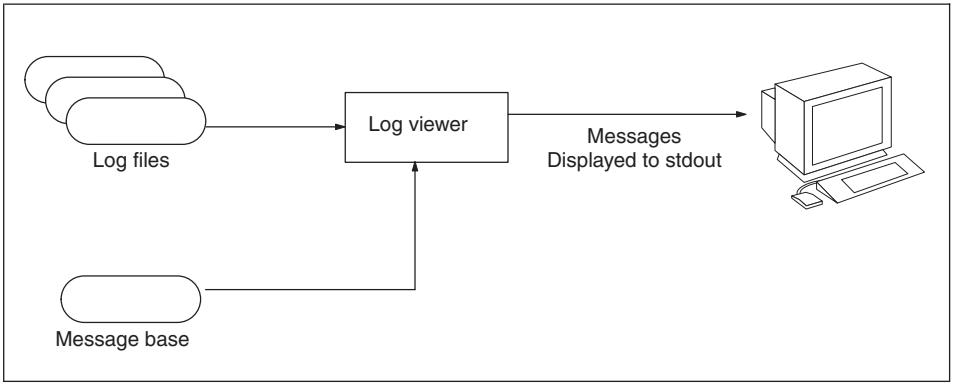


Figure 52. The log viewer

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 Only

```
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]  ...]]
```

End of UNIX Only

Windows Only

```
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]  ...]]
```

End of Windows Only

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 either the active log file (*-a*) or another specified log file (*logfile*). As new records are written to the file, they are displayed to stdout. (Similar to the UNIX `tail` command.) This option must be preceded by *-a* or followed by *logfile*.
-f	Specifies continuous display of either the active log file (*-a*) or another specified log file (*logfile*). 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.) This option must be preceded by *-a* or followed by *logfile*.
-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).
-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, see "Event category" on page 240.
-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, see "Event severity" on page 239.

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, see "Log files" on page 241.

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 Only

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.
```

End of UNIX Only

Windows Only

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.
```

End of Windows Only

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, see "Setting the log severity filter level" on page 243.

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 Warehouse 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.

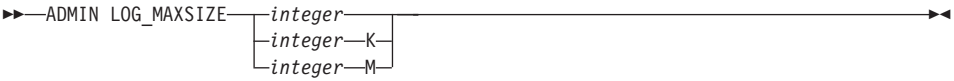
Specifying the location of log files

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 path name or a relative path name. Relative path names 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

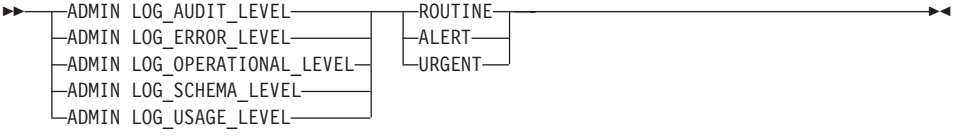
If you specify K or M, this suffix must immediately follow the numeric value with no spaces.

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 xALTER 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, see "Accounting" on page 248.

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, see 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, see the *IBM Red Brick 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.
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.

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

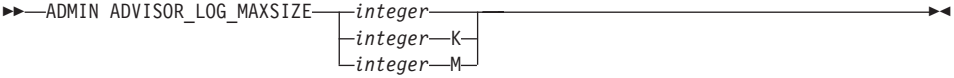
The ADMIN ADVISOR_LOG_DIRECTORY `rbw.config` file parameter specifies the directory that stores the log files.

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 path name. Relative path names 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, see 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.

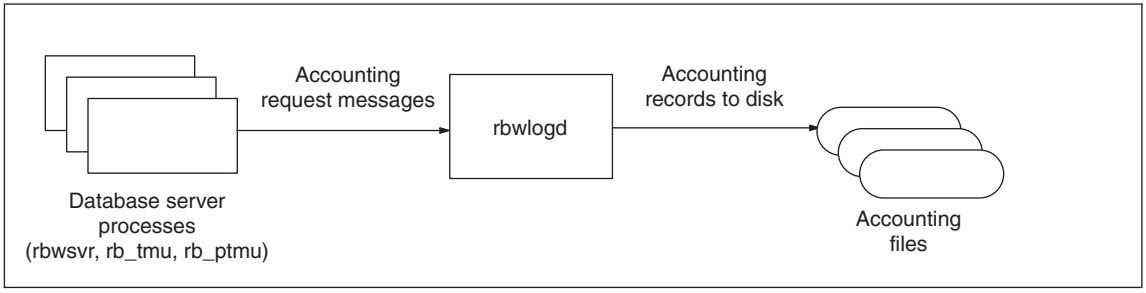


Figure 53. Accounting process on UNIX

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 user name
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
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

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, see 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 path name or a relative path name. Relative path names 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, see 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

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, see the *Table Management Utility Reference Guide*.

This chapter includes the following sections:

- Locking Tables and Databases
- Obtaining Information on Tables and Indexes
- Monitoring Growth of Tables and Indexes
- How Space is Allocated to a Segment
- Altering Segments
- Maintaining a Time-Cyclic Database
- 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

© Copyright IBM Corp. 1996, 2004

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 191.

For a complete description of the LOCK statement, see the *SQL Reference Guide*.

READER PRIORITY option

The order in which locks are granted is controlled by the READER_PRIORITY option in the `rbw.config` file. By default, this 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.

Although this behavior is appropriate for most decision-support applications, 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. And when queries are executed, they must acquire a database-level lock on the system tables, which is not released until the individual tables are locked.

For applications that support activities such as DDL operations and loads while the database is online, setting READER_PRIORITY to off disables query-priority locking. Operations get locks in the order for which they are applied, with no priority given to read locks for queries. Unless you routinely encounter problems with delayed DDL operations that must be performed

while the database is online, IBM recommends that you leave OPTION READER_PRIORITY set to ON, giving queries priority over operations that write to the database.

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, IBM Red Brick Warehouse provides a read-only (RO) lock and a write-blocking (WB) lock.

For databases with versioning, Red Brick 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.
RD	Read-data	Read transaction that does not allow any write transactions to the table.
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 "READER PRIORITY option" on page 256.
WS	Write-schema	Only used when READER_PRIORITY is set to OFF. See "READER PRIORITY option" on page 256.

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, see "Maintaining referential integrity with ON DELETE" on page 141.

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: IBM 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, see 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, see "Types of table locks" on page 257.

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, see 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:

- Obtain segment statistics for tables, including row counts
- Check for damage to the table storage data structure
- Repair damage to the table storage data structure

You can use the CHECK INDEX statement to obtain configuration and size information and to check an index for corruption.

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, see "Using CHECK TABLE with the VERBOSE option" on page 147.

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.
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 on page 143.
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 the example on page 269.
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 142.
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 on page 144..

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
```

The following example shows the contents of the output files that the CHECK TABLE statement with the VERBOSE option produces for the Sales table.

```
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.
```

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.

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, see "Freespace field" on page 280.

For a description of how to use the CHECK TABLE segment statistics to determine the effectiveness of the VARCHAR column fill factor, see "Using CHECK TABLE with the VERBOSE option" on page 147.

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: IBM 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 IBM Customer Support.

CHECK INDEX size and configuration information

The CHECK INDEX output is slightly different for each type of index:

- B-TREE index

  For details, see "Size and space information for B-TREE index" on page 265 and "Configuration information for B-TREE indexes" on page 267.

- STAR index

  The CHECK INDEX summary output file provides configuration information for each table referenced in the STAR index. For details, see "Configuration information for STAR indexes" on page 269. "Size and space information for STAR indexes" on page 268 describes similar space information as a B-TREE index.

- TARGET index

  Each TARGET index is internally comprised of three B-TREE indexes, as "CHECK INDEX output for TARGET index" on page 270 describes. "Size

and space information for TARGET indexes" on page 272 describes the statistical information that CHECK INDEX displays for TARGET indexes.

Size and space information for B-TREE index

This example shows the CHECK INDEX statement with the VERBOSE option for the BTREE_SALES_IDX index on the Sales table from the Aroma database.

```
RISQL> check index BTREE_SALES_IDX
directory '/qa/local/virginia' verbose;>
INFORMATION

Index: 22 Segment: 58 is ok
Report File: /qa/local/virginia/chk_0_58_rep.20020604.074848.22361

Index: 22 Segment: 59 is ok
Report File: /qa/local/virginia/chk_0_59_rep.20020604.074848.22362

Index validation succeeded
Summary File:
/qa/local/virginia/chk_0_Summary_rep.20020604.074848.17412
```

The CHECK INDEX statement produces an output file for each segment and a summary file. The name of each output file for each segment starts with zero and the segment number, as the following filenames shows:

```
chk_0_58_rep.20020604.074848.22361
chk_0_59_rep.20020604.074848.22362
```

The following example shows the contents of the CHECK INDEX output file for the first segment of the B-TREE index.

```
114 nala % more
chk_0_58_rep.20020604.074848.22361
Checking Index Segment

Index name: BTREE_SALES_IDX
Index revision number: NULL
------------------------------

Validating index...

BTree index of type: IX_COMPOSITE
Keysize:  8
Unique:   0
Nulls:    0
Number of variable length cols: 0
Index is stored in 2 segments.
Entries per page: 583
Fill factor: 1.000000
Offset to children: 4664
Nulls are not allowed in this index.

Composite types (2):
0) Type: IX_INT4
1) Type: IX_INT4
```

```
Validating 0th segment
-----------------------------------------------
Validating segment LOCAL_BTREE1...
Root block  : 3
Index depth : 1

Average entries in leaf nodes  : 576.283333
Average entries in inner nodes : 60.000000

Total leaf entries             : 34577
Total Inner entries            : 60

Total non-null rows indexed    : 34577
Total leaf nodes               : 60
Total inner nodes              : 1

Segment is ok
```

When you specify the VERBOSE option, the CHECK INDEX statement generates the index size and space information as shown in the previous example. If you specify the DESCRIBE option, CHECK INDEX does not display the following four lines of information:

```
Total leaf entries             : 34577
Total Inner entries            : 60

Total leaf nodes               : 60
Total inner nodes              : 1
```

The CHECK INDEX size and space information correlate to the information in the RBW_SEGMENTS and RBW_STORAGE system tables as follows:

- **Total leaf entries** value 34577 equals the number of rows in the corresponding data segment for this index segment (ROWCOUNT value in RBW_SEGMENTS).
- The values in **Total leaf nodes** and **Total inner nodes is the total number of 8-kilobyte blocks.**

  The following calculation shows the amount of space that segment LOCAL_BTREE1 occupies:

```
Space used = (Total leaf nodes + Total inner nodes) * 8 kilobytes
           = (60 + 1)* 8 kilobytes
           = 488 kilobytes
```

The USED column in the RBW_STORAGE system table indicates how much of the PSU has been allocated so far. The USED value is usually larger than the calculated space from the CHECK INDEX output because the index build adds extra blocks for splits and metadata. In this example, the USED value 496 is larger than the calculated value 488 because the USED value includes an 8-kilobyte block for metadata.

```
RISQL> select substr(seg.name, 1,17) as segname, pseq, used,
>    totalfree, rowcount
>    from rbw_segments seg, rbw_storage sto
>    where tname = 'SALES' and seg.name = sto.segname
>    order by seg.name;
Move this complete output to last CHECK INDEX description
SEGNAME           PSEQ        USED       TOTALFREE   ROWCOUNT
DAILY_DATA1          2          32             992      34577
DAILY_DATA1          1        1024             992      34577
DAILY_DATA2          2          56             968      35364
DAILY_DATA2          1        1024             968      35364
DEFAULT_SEGMENT_2    1        1800         2095344       NULL
LOCAL_BTREE1         1         496             528       NULL
LOCAL_BTREE2         1         504             520       NULL
SALES_STAR_SEG1      1         424             600       NULL
SALES_STAR_SEG2      1         432             592       NULL
SALES_TARGET_IX1     1          56             968       NULL
SALES_TARGET_IX2     1          56             968       NULL
```

Configuration information for B-TREE indexes

The CHECK INDEX statement reports configuration information for the index. You can use the DESCRIBE option to show this configuration information without validating the index. The CHECK INDEX output in the previous example shows the following configuration information:

- Keysize of the B-TREE index is 8 bytes
- The index is stored in two segments
- 583 entries fit on a page (8-kilobyte block), but only an average of 576.28 entries are on each leaf node because the index is not full
- Fill factor is 100%
- NULL values are not allowed for this BTREE_SALES_IDX index
- The index is a composite (multicolumn) index and each column has an INTEGER data type

The following example shows that the CHECK INDEX summary file contains the following configuration information:

- Names of the columns defined in the B-TREE index
- Total number of rows in the table

```
116 nala % more chk_0_Summary_rep.20020604.074848.17412
Checking overall index

Index name: BTREE_SALES_IDX
Index revision number: NULL
-------------------------------------------

   NO.   COLUMN NAME
   ---  ------------------------------
   0     CLASSKEY
   1     PRODKEY
```

```
Total data row(s) in this table         : 69941
Total data row(s) indexed on this table : 69941
```

Size and space information for STAR indexes

This example shows the CHECK INDEX statement with the VERBOSE option for the SALES_STAR_IDX index on the Sales table from the Aroma database.

```
RISQL> check index SALES_STAR_IDX
directory '/qa/local/virginia' verbose;>
INFORMATION


Index: 20  Segment: 31 is ok
Report File: /qa/local/virginia/chk_0_31_rep.20020530.131423.28183

Index: 20  Segment: 37 is ok
Report File: /qa/local/virginia/chk_0_37_rep.20020530.131423.28184

Index validation succeeded
Summary File:
  /qa/local/virginia/chk_0_Summary_rep.20020530.131423.20002
```

Because the SALES_STAR_IDX index consists of two segments, the CHECK INDEX statement produces two output files and a summary file. The name of each output file for the CHECK INDEX statement starts with zero and the segment number, as the following filenames shows:

```
chk_0_31_rep.20020530.131423.28183
chk_0_37_rep.20020530.131423.28184
```

The following example shows the contents of the CHECK INDEX output file for the first segment of the STAR index.

```
120 nala % more chk_0_31_rep.20020530.131423.28183
Checking Index Segment

Index name: SALES_STAR_IDX
Index revision number: NULL
------------------------------

Validating index...

STAR index of type: IX_BYTE_STRING
Keysize: 6
Unique:  1
Nulls:   0
Number of variable length cols: 0
Index is stored in 2 segments.
Entries per page: 681
Fill factor: 1.000000
Offset to children: 4086
Nulls are not allowed in this index.

Validating 0th segment
```

```
-----------------------------------------------
Validating segment SALES_STAR_SEG1...
Root block:  3
Index depth: 1

Average entries in leaf nodes  : 677.980392
Average entries in inner nodes :  51.000000

Total leaf entries             : 34577
Total Inner entries            : 51

Total nun-null rows indexed    : 34577
Total leaf nodes               : 51
Total inner nodes              : 1

Segment is ok
```

When you use the VERBOSE option, CHECK INDEX shows the following size information:

- Total number of index entries (leaf and inner)

  The output above shows that 34577 entries are currently in this SALES_STAR_SEG1 index segment.

- Total number of 8-kilobyte blocks used as leaf nodes and inner nodes

  The following calculation shows the amount of space segment SALES_STAR_SEG1 occupies:

```
Space used = (Total leaf nodes + Total inner nodes) * 8 kilobytes
           = 52 * 8 kilobytes
          = 416 kilobytes
```

This calculated 416 kilobytes is less than the USED value 424 in the RBW_STORAGE system table because the USED value includes an 8-kilobyte block for metadata.

```
RISQL> select substr(seg.name, 1,18) as segname, pseq, maxsize,
>    initsize, extendsize,used
>    from rbw_segments seg, rbw_storage sto
>    where tname = 'SALES'and seg.name = sto.segname
>    order by seg.name;
SEGNAME           PSEQ   MAXSIZE   INITSIZE    EXTENDSIZE  USED
...
SALES_STAR_SEG1   1      1024      104         104          424
SALES_STAR_SEG2   1      1024      104         104          432
```

Configuration information for STAR indexes

The CHECK INDEX output in the previous example of a STAR index shows similar configuration information as the output in the example on page 265 for a B-TREE index. The following example shows that the CHECK INDEX summary file contains the following configuration information for each foreign key constraint defined in the STAR index:

- MAXSEGMENTS

- MAXROWS PER SEGMENT
- Column name

```
122 nala % more chk_0_Summary_rep.20020530.131423.20002
Checking overall index

Index name:SALES_STAR_IDX
Index revision number:NULL
------------------------------------------------------------
MAXROWS values used to build this index:

Constraint: SALES_DATE_FKC
NO.    MAX SEGMENTS    MAX ROWS PER SEGMENT    COLUMN NAME
---   -------------   ---------------------   -----------
0      1               2500                    PERKEY

Constraint: SALES_PRODUCT_FKC
NO.    MAXSEGMENTS     MAX ROWS PER SEGMENT    COLUMN NAME
---   -------------  ----------------------  ----------
0      1               2500                    CLASSKEY
1      1               2500                    PRODKEY

Constraint: SALES_STORE_FKC
NO.    MAX SEGMENTS    MAX ROWS PER SEGMENT    COLUMN NAME
---   -------------  ----------------------  -----------
0      1               2500                    STOREKEY

Constraint: SALES_PROMO_FKC
NO.    MAX SEGMENTS    MAX ROWS PER SEGMENT    COLUMN NAME
---   -------------  ----------------------  -----------
0      1               2500                    PROMOKEY

Total data row(s)in this table         : 69941
Total data row(s) indexed on this table: 69941
```

CHECK INDEX output for TARGET index

This example shows the CHECK INDEX statement with the VERBOSE option for the PROD_PER_SALES_IX index on the Sales table from the Aroma database.

```
RISQL> check index PROD_PER_SALES_IX
directory '/qa/local/virginia' verbose;
INFORMATION

Index: 21 Segment: 38 is ok
Report File: /qa/local/virginia/chk_0_38_rep.20020530.130747.25523

Index: 21 Segment: 40 is ok
Report File: /qa/local/virginia/chk_0_40_rep.20020530.130747.25524

Index validation succeeded
Summary File:
/qa/local/virginia/chk_0_Summary_rep.20020530.130747.20002
```

A TARGET index internally consists of the following B-TREE indexes:

- Domain bit index

  This index contains one entry for each domain value. The key is the domain value.
- RLA (Row List Array) bit index

  This index points to the subblocks that hold the row information. The key is the domain value and a rowID.
- NULL bit index

  This index points to the subblocks that hold the row information for NULL rows.

The following example shows an excerpt of the CHECK INDEX output for the first segment of the TARGET index. This excerpt shows the DOMAIN bit index information which is similar to the CHECK INDEX output for a B-TREE index.

```
106 nala % more chk_0_38_rep.20020530.130747.25523
Checking Index Segment

Index name: PROD_PER_SALES_IX
Index revision number: NULL
------------------------------

Column Name is PROMOKEY
Representation is BVI_HYBRID

Domain Bit Index
-----------------
Validating index...

BTree index of type: IX_INT4
Keysize: 4
Unique:  0
Nulls:   0
Number of variable length cols: 0
Index is stored in 2 segments.
Entries per page: 817
Fill factor: 1.000000
Offset to children: 3268
Nulls are not allowed in this index.

Validating 0th segment
------------------------------------------------
Validating segment SALES_TARGET_IX1...
Root block:  1
Index depth: 0

Average entries in leaf nodes  : 146.000000
Average entries in inner nodes :   0.000000

Total leaf entries             : 146
Total Inner entries            : 0
```

```
Total non-null rows indexed    : 34577
Total leaf nodes               : 1
Total inner nodes              : 0
```

The TARGET INDEX information is presented as follows:

```
Column Name is PROMOKEY
Representation is BVI_HYBRID
```

When you specify the VERBOSE option, the CHECK INDEX statement generates the index size and space information as shown in the previous example. If you specify the DESCRIBE option, CHECK INDEX does not display the following four lines of information:

```
Total leaf entries             : 34577
Total Inner entries            : 0

Total leaf nodes               : 1
Total inner nodes              : 0
```

Size and space information for TARGET indexes

For TARGET indexes, CHECK INDEX displays the size and space statistics for the DOMAIN bit, RLA bit, and NULL bit indexes. The following example shows this statistical information.

```
Statistical Information
-----------------------

Total no. of Domain blocks used : 1

Average space usage in Domain blocks : 18.017578 percent

Total no. of RLA blocks used : 1

Average space usage in RLA blocks : 28.710938 percent

Total no. of NULL blocks used : 0

Total no. of data blocks used : 3
Total no. of data subblocks used : 146
Total data size : 3573

Average space usage in data blocks : 67.903646 percent


Segment Domain Size is 146
Segment Size is 34577
Table Size is 69941
Null Vector Size is 0


Segment is ok
```

```
Statistical Information
-----------------------

Total no. of Domain blocks used : 1

Average space usage in Domain blocks : 18.017578 percent

Total no. of RLA blocks used : 1

Average space usage in RLA blocks : 28.710938 percent

Total no. of NULL blocks used : 0

Total no. of data blocks used : 3
Total no. of data subblocks used : 146
Total data size : 3573

Average space usage in data blocks : 67.903646 percent

Segment Domain Size is 146
Segment Size is 34577
Table Size is 69941
Null Vector Size is 0

Segment is ok
```

Each block is 8 kilobytes. The following calculation shows the amount of space that segment SALES_TARGET_IX1 occupies:

```
Space used = ( Num Domain blocks + Num RLA blocks + Num Null blocks
               + Num data blocks ) * 8 kilobytes
           = ( 1 + 1 + 0 + 3 ) * 8 kilobytes
           = 5 * 8 kilobytes
           = 40 kilobytes
```

This calculated space value 40 from the CHECK INDEX statistics is less than the USED value 56 in the RBW_STORAGE system table for segment SALES_TARGET_IX1 because the USED value contains additional 8-kilobyte blocks used for index building and metadata. When you build TARGET indexes unoptimized, additional blocks are added for index splits.

```
RISQL> select substr(seg.name, 1,17) as segname, pseq, used,
>   totalfree, rowcount
>   from rbw_segments seg, rbw_storage sto
>   where tname = 'SALES' and seg.name = sto.segname order by seg.name;
SEGNAME            PSEQ      USED    TOTALFREE    ROWCOUNT
...
SALES_TARGET_IX1      1        56          968        NULL
SALES_TARGET_IX2      1        56          968        NULL
```

The following table describes the statistics fields that CHECK INDEX generates for a TARGET index.

CHECK TABLE segment statistics field	Description
Total no. of Domain blocks used	Number of 8-kilobyte blocks that reside in the domain B-TREE of the TARGET index.
Average space usage in Domain blocks	Percentage of space used in the domain bit index of the TARGET index.
Total no. of RLA blocks used	Number of 8-kilobyte blocks that reside in the RLA B-TREE of the TARGET index.
Average space usage in RLA blocks	Percentage of space used in the RLA bit index of the TARGET index.
Total no. of NULL blocks used	Number of 8-kilobyte blocks that reside in the NULL B-TREE of the TARGET index.
Average space usage in NULL blocks	If NULLs are allowed, percentage of space used in the NULL bit index of the TARGET index.
Total no. of data blocks used	Total number of 8-kilobyte blocks that contain subblocks.
Total no. of data subblocks used	Each data block is comprised of subblocks which contain the row pointers. The number of subblocks depends on the storage representation of the TARGET index and the number of rows per subblock.
Average space usage in data blocks	Percentage of space used in each data block.
Segment Domain Size is	Number of distinct values for the indexed column.
Segment Size is	Number of rows indexed in this index segment.
Table Size is	Total number of rows in the table.
Null Vector Size is	Number of NULL values in the indexed column.

Configuration information for TARGET indexes

The CHECK INDEX statement reports similar configuration information for a TARGET index as for a B-TREE or STAR index. The CHECK INDEX output in the example on page 270 shows the following configuration information that is unique to the PROD_PER_SALES_IX TARGET index:

- Column name is PROMOKEY
- Representation is `BVI_HYBRID`
- 817 entries fit on a page (8-kilobyte block), but only 146 entries are on the leaf node because the index is not full

The following example shows that the CHECK INDEX summary file contains the total number of domain values (`302`) in the TARGET index.

```
Checking overall index

Index name: PROD_PER_SALES_IX
Index revision number: NULL
-----------------------------------------------------

   NO.   COLUMN NAME
   ---  -----------
   0     PROMOKEY
```

```
Total Domain values in this table        : 302
Total data row(s) in this table          : 6994
1Total data row(s) indexed on this table : 69941
```

Checking for index corruption

When you specify the VALIDATE option, the CHECK INDEX statement compares the key in the table row to the key in the index to ensure that they match.

If you specify the FULL keyword with the VALIDATE option, the CHECK INDEX operation also ensures that each index entry points to a unique row and that every row in the table is pointed to by one index entry.

The CHECK INDEX with the full validate option uses the value in the INDEX_TEMPSPACE_THRESHOLD configuration parameter. For considerations on setting this parameter, see "THRESHOLD value" on page 114.

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 TOTALFRE**E**, 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, see "Considerations for growing tables" on page 123.

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 IBM Red Brick Warehouse. 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. For more information, see "Dynamic adjustment of MAXSIZE" on page 282.

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" later in this section.

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" later in this section.

IBM Red Brick Warehouse 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" later in this section.

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" previously discussed, 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 IBM Red Brick Warehouse 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" discussed previously in this section, 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
```

How space is allocated 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 load or insert operation continues.

PSUs are used sequentially in the order in which they were defined (by sequence number in the PSEQ column of the RBW_STORAGE table). 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 not add space to a previous PSU once the next PSU is in use. However, if rows are deleted at the end of a segment and the space is released with the ALTER SEGMENT RELEASE STORAGE command, you can drop the last empty PSU. You can then change the MAXSIZE value of a previous PSU.

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.

Dynamic adjustment of MAXSIZE

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.

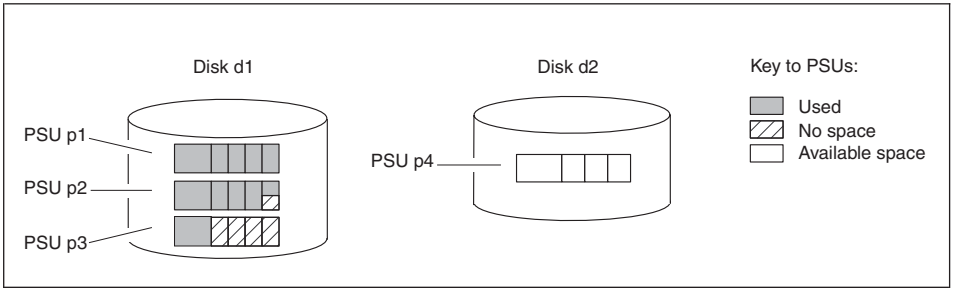


Figure 54. PSU size adjustments

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.

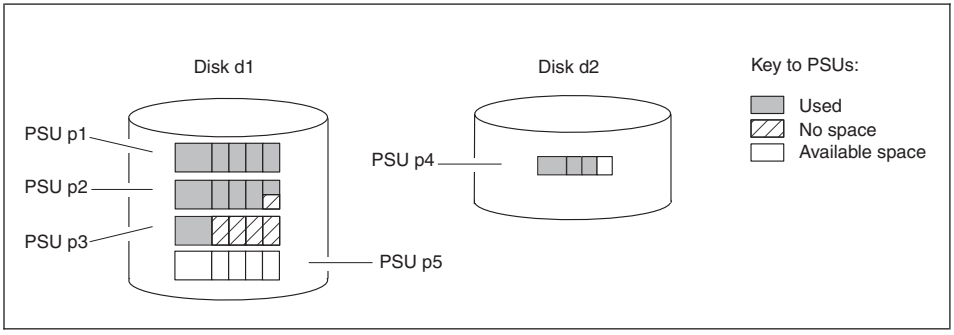


Figure 55. Space use between disks

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)
- Specify a new range for a segment, to change how data is distributed among segments or to move the range values to a nonadjacent segment (RANGE, RANGE MOVE)
- Add a PSU to accommodate growing tables (ADD STORAGE)
- Release space allocated but not used in a segment or drop a PSU from a segment to return storage to the operating system (RELEASE STORAGE, DROP LAST STORAGE)
- Change the maximum size or the extend size of PSUs to control their growth (CHANGE MAXSIZE, CHANGE EXTENDSIZE)
- Change the path of a PSU or migrate an entire segment to a new location as needed (CHANGE PATH, MIGRATE TO)
- 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, see 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, supply a valid range specifier based on the data type of the segmenting column.

To attach a segment to a B-TREE or TARGET index, use one of the following methods:

- For a local index, specify the data segment name in the RANGE LIKE SEGMENT clause. For an example of when to use this clause, see "Attaching a segment to a single-segment table or index" on page 286.
- For a non-local 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 (which you might do to drop the oldest data in a time-cyclic database and reuse the segment for the new data), use one of the following methods:

- Move the range position of the segment with the MOVE keyword. For details, see the procedure on page 298.
- Detach the segment and attach it with a new range specification. For details, see the procedure on page 295.

Attaching a segment to a single-segment table or index

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. For example, suppose you want to attach another segment to the line_items table and attach a corresponding segment to its STAR index. The following CREATE statements place the line_items table in a single default segment and the line_items_star_idx STAR index in another single default segment:

```
create table line_items (
   order_no integer not null,
   line_item integer not null,
   perkey integer not null,
   classkey integer not null,
   prodkey integer not null,
   receive_date date not null,
   quantity integer not null,
   price dec(7,2) not null,
   primary key (order_no, line_item),
   constraint line_order_fkc foreign key (order_no)
      references orders (order_no),
   onstraint line_period_fkc foreign key (perkey)
      references period (perkey),
   constraint line_product_fkc foreign key (classkey, prodkey)
      references product (classkey, prodkey))
   maxrows per segment 2000;

create star index line_items_star_idx on line_items
   (line_order_fkc, line_period_fkc, line_product_fkc);
```

To attach another segment to a single-segment table or index:

1. If the table was defined with the default MAXSEGMENTS value of 1, you must first alter the table to specify a larger value.

   ```
   alter table line_items change maxsegments 2;
   ```

2. Create another segment for the table:

   ```
   create segment line_items_seg2
      storage 'line_items_seg2_psu1'
      maxsize 1024
      initsize 100
      extendsize 100;
   ```

3. Obtain the default segment name for the table:

   ```
   select substr(name, 1,20) as name,
      substr(tname,1,17) as tname,
      substr(iname, 1,22) as iname, NPSUS
   from rbw_segments
   where tname = 'LINE_ITEMS';
   NAME                 TNAME         INAME              NPSUS
   DEFAULT_SEGMENT_19   LINE_ITEMS    NULL                  1
   DEFAULT_SEGMENT_20   LINE_ITEMS    LINE_ITEMS_PK_IDX     1
   DEFAULT_SEGMENT_36   LINE_ITEMS    LINE_ITEMS_STAR_IDX   1
   ```

4. Alter the existing segment to specify the segmenting column

Because the table resides in a single segment and no segmenting column was specified when the table was created, you must define a segmenting column before you can attach additional segments.

```
alter segment default_segment_19 of table line_items
segment by order_no;
```

5. Obtain the range to specify for the new segment:

```
select distinct(order_no) from line_items;
   3600
   ...
   3624
```

Suppose you want the first segment to contain the first 25 order numbers and the second segment to contain the next 25 order numbers.

```
Segment 1 range (3600:3625)
Segment 2 range (3625:3650)
```

A segment includes the starting boundary value but not the ending value. Therefore, the first segment will contain the values `3600` to `3624` and the second segment will contain the values `3625` to `3649.` However, you must specify the values `min` and `max` for the first and last segments, respectively, when you define the segment ranges.

```
seg1       seg2
min:3625   3625:max
```

6. Attach the new segment to the table:

```
alter segment line_items_seg2 attach to table line_items
range (3625:max);
```

The table Line_items must contain no rows in the range of the new segment. Otherwise, you receive an error indicating that the attach operation would require data movement.

7. Rename the default segment of the table to a more meaningful, relevant name. Use a name similar to the newly attached segment:

```
alter segment default_segment_19 of table line_items
   rename line_items_seg1;
```

Warning: Even though renaming a default segment changes its name, it still behaves like a default segment. If you drop the table, this segment will also be dropped.

8. If you also want to segment an existing index on the table, follow these steps:
   a. Create the new index segment:

```
create segment lisi_seg2
   storage 'line_items_star_psu2'
   maxsize 512
   initsize 50
   extendsize 50;
```

b. Attach the new index segment:

```
alter segment lisi_seg2
   attach to index line_items_star_idx range like
   segment line_items_seg2;
```

This example specifies the RANGE LIKE SEGMENT clause to attach the index segment with a range identical to the newly attached table segment.

c. Rename the default segment of the index to a more meaningful, relevant name. Use a name similar to the newly attached index segment:

```
alter segment default_segment_36
   of index line_items_star_idx rename line_items_seg1;
```

Warning: Even though renaming a default segment changes its name, it still behaves like a default segment. If you drop the index, this segment will also be dropped.

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.

To move the range of a segment in a table or index, specify a new range of data values or row numbers for the segment. For an example of the RANGE MOVE clause in the ALTER SEGMENT statement, see "Rolling Off and Reusing Data and Index Segments" later in this section.

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, see "Partial availability of tables and indexes" on page 25.

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, see 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 *IBM Red Brick 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.

Releasing and dropping storage from a segment

If a segment contains unused space at the end, you can return this space to the operating system by using one of the following options in the ALTER SEGMENT statement:

- RELEASE STORAGE

  Use the RELEASE STORAGE option to free space where rows were deleted at the end of a segment.
- DROP LAST STORAGE

  Use the DROP LAST STORAGE option to drop the last empty PSU in a segment.

Important: If rows were deleted in the last PSU but the space is still logically in use, you must first release the space with the RELEASE STORAGE option before you can drop the PSU. For the procedure, see page 289.

For example, suppose that the Sales table consists of 2 segments and that you no longer require the data in the second segment. You want to release the disk space for other applications to use, but you want to keep the segment (at its minimum space allocation) for later use. The following procedure uses this example.

To release and drop storage from a segment:

1. Delete the data in the segment.

```
RISQL> delete from sales where perkey > 415;
** INFORMATION ** (211) Deleted 35270 rows from SALES.
```

2. Run CHECK TABLE to verify the physical integrity of the table.

```
RISQL> check table sales directory'/qa/local/virginia/check_results/
'verbose;
```

   Ensure that the following line appears in the CHECK TABLE execution messages:

```
No inconsistencies were detected.
```

3. Determine if freespace exists in the segment.
   a. Look at the **free blocks** field in the Segment statistics section of the output of the CHECK TABLE statement.

```
Segment statistics:
active rows:94, deleted rows:0, total blocks:135, free blocks:118
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:128034, unused RowIDs:4130, unusable freespace:0
```

      In these statistics, the **free blocks** field shows a value of 118 and the **active rows** field shows that 94 rows still exist in the segment. However, not all of these118 blocks might be freed if any of the 94 active rows are at the end of the segment.

   b. Query the used column of the rbw_storage system table to determine if any space is still logically in use.

```
RISQL> select substr(seg.name, 1,17) as segname, pseq,
         used, totalfree, rowcount
      from rbw_segments seg, rbw_storage sto
      where tname = 'SALES' and seg.name = sto.segname and
         usage = 'TABLE'
      order by seg.name;
SEGNAME      PSEQ   USED   TOTALFREE   ROWCOUNT
DAILY_DATA1     1   1024         992      34577
DAILY_DATA1     2     32         992      34577
DAILY_DATA2     1   1024         984         94
DAILY_DATA2     2     40         984         94
```

      The used column shows that the second segment daily_data2 has 1064 kilobytes (1024 in PSU1 and 40 in PSU2) still in use. The rowcount column shows that 94 rows still exist in the second segment.

4. Release space that was once occupied by deleted rows at end of segment.

```
RISQL> alter segment DAILY_DATA2 of table sales release
> storage;
```

   a. Because the query and CHECK TABLE outputs in step 3 show that the second segment still has 94 rows in it, you might not be able to release much storage because the remaining rows might be at the end of the PSU and the free space is in the middle of the segment. In this case, you might receive one of the following messages after the ALTER SEGMENT RELEASE STORAGE statement:

```
** ERROR ** (6402) Cannot RELEASE STORAGE: No freeable space
   in the segment.
```

This 6402 message is issued when the end of the segment still contains active rows.

```
** WARNING ** (6403) Percentage of blocks freed:  0.74 percent.
```

This 6403 message appears when the release operation was able to free at least one 8-kilobyte block at the end of the segment.

b. If all rows are deleted from the second segment, then no rows exist at the end and you would be able to release more space. For example, after deleting the remaining 94 rows in segment DAILY_DATA2, the following Segment Statistics section in the CHECK TABLE output shows 0 active rows:

```
Segment statistics:
active rows:0, deleted rows:0, total blocks:134, free blocks:118
rows/block:264, storage reclen:29, average reclen:NaN, indirect rows:0
max reclen:29, longest rec:0, min reclen:29, shortest rec:8186
freespace:122760, unused RowIDs:3960, unusable freespace:0
```

However, the USED column in the RBW_STORAGE system table shows that 1064 kilobytes is still logically used:

```
RISQL> select substr(seg.name, 1,17) as segname, pseq,
      used, totalfree, rowcount
   from rbw_segments seg, rbw_storage sto
   where tname = 'SALES' and seg.name = sto.segname and
      usage = 'TABLE'
   order by seg.name;
SEGNAME      PSEQ   USED   TOTALFREE   ROWCOUNT
DAILY_DATA1     1   1024         992      34577
DAILY_DATA1     2     32         992      34577
DAILY_DATA2     1   1024         984          0
DAILY_DATA2     2     40         984          0
```

In this example, the release storage operation frees more storage:

```
RISQL> alter segment DAILY_DATA2 of table sales release storage;
** WARNING ** (6403) Percentage of blocks freed: 98.51 percent.
```

5. Verify the effect of the RELEASE STORAGE operation in step 4b

a. Run CHECK TABLE:

```
RISQL> check table sales
directory'/qa/local/virginia/check_results/' verbose;
```

The following CHECK TABLE output shows that all the data blocks have been released, as indicated by the 0 value in the **free blocks** field and a value of 0 for the **Total data blocks** field in the line just above the segment statistics. The **total blocks** field contains the value 1 for the control block that exists in every segment.

```
19 nala % more chk_11_22_rep.20020127.180113.15692
Checking Table
...
Expecting:0 active rows in segment
Scanning all active blocks. Total data blocks:0...
Segment statistics:
active rows:0, deleted rows:0, total blocks:1, free blocks:0
rows/block:264, storage reclen:29, average reclen:NaN, indirect rows:0
max reclen:29, longest rec:0, min reclen:29, shortest rec:8186
freespace:0, unused RowIDs:0, unusable freespace:0
```

b. The USED column of the RBW_STORAGE system table verifies that the space is no longer logically in use. The USED column shows that the second PSU of segment daily_data2 contains 0 kilobytes and the first PSU contains only 8 kilobytes still in use for the control block.

```
RISQL> select substr(seg.name, 1,17) as segname, pseq,
         used, totalfree, rowcount
      from rbw_segments seg, rbw_storage sto
      where tname = 'SALES' and seg.name = sto.segname and
         usage = 'TABLE'
      order by seg.name;
SEGNAME      PSEQ   USED   TOTALFREE   ROWCOUNT
DAILY_DATA1     1   1024         992      34577
DAILY_DATA1     2     32         992      34577
DAILY_DATA2     1      8        2040          0
DAILY_DATA2     2      0        2040          0
```

6. If step 4 released a large amount of space, you can remove the last PSU from the daily_data2 segment.

```
RISQL> alter segment DAILY_DATA2 of table sales release storage;
** WARNING ** (6403) Percentage of blocks freed: 98.51 percent.
RISQL> alter segment DAILY_DATA2 of table sales drop last storage;
```

The following query shows that only the first PSU remains in the daily_data2 segment and the totalfree column contains less space than in step 5b:

```
RISQL> select substr(seg.name, 1,17) as segname, pseq, used,
> totalfree, rowcount
> from rbw_segments seg, rbw_storage sto
> where tname = 'SALES' and seg.name = sto.segname and
>     usage = 'TABLE'
> order by seg.name;
SEGNAME       PSEQ    USED   TOTALFREE   ROWCOUNT
DAILY_DATA1      1    1024         992      34577
DAILY_DATA1      2      32         992      34577
DAILY_DATA2      1       8        1016          0
```

However, if step 3 did not release much space, you receive the following error:

```
RISQL> alter segment DAILY_DATA2 of table sales release storage;
** WARNING ** (6403) Percentage of blocks freed:  0.74 percent.
RISQL> alter segment DAILY_DATA1 of table sales drop last storage;
```

```
** ERROR ** (6405) Cannot drop PSU
'/qa/local/virginia/aroma/sales_small_psu2'
because it has blocks that are in use or are not yet released.
```

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 270.

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, see the *SQL Reference Guide*.

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 path name 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 path names. 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 path name 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.)

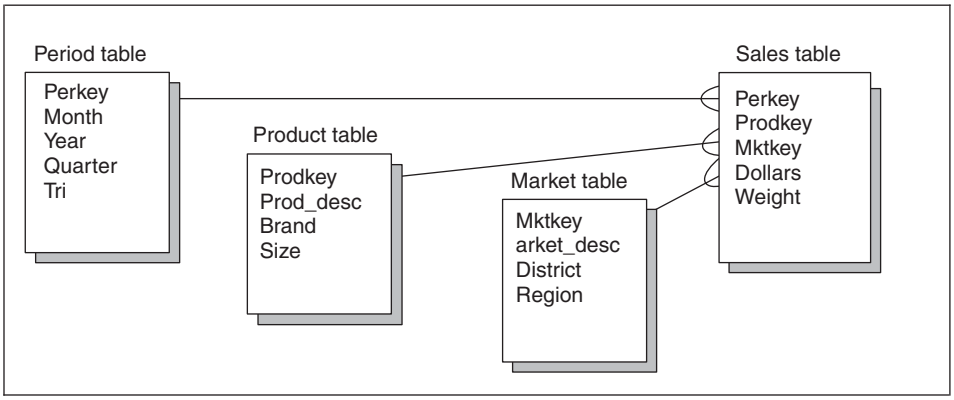
Warning: Use the FORCE INTACT option only when you know the segment is undamaged. If you are uncertain, use the VERIFY option.

Maintaining a time-cyclic database

This section describes 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 example is a continuation of the example in “Creating a time-cyclic database” on page 155. The following figure illustrates the tables used in this example.



Assume the database is created during Q1 2002 and loaded with data for eight full quarters (all of 2000 and 2001) plus data for the current quarter, which is updated weekly. For the CREATE statements used in this example, see "Creating a time-cyclic database" on page 155.

Q2 2002 is about to start. You have two tasks:

- Before Q2 starts, you need to add a new segment to hold Q2 2002 data.
- After Q2 starts, you need to remove the Q1 2000 data.

Tip: When you are ready to delete the data for the oldest quarter, you need to decide whether to save the segment or to drop the segment. If you save the segment, you can reuse it by detaching it from the table and then reattaching it wherever you want to use it. If you drop the segment, you must create another segment if you need one.

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 2002, you can remove the data from the first quarter of 2000. This section shows two procedures to reuse the s_1q00 data and star_1q00 index segments for the data and index entries for the second quarter of 2002:

- Detach a segment and attach it with the new range values for the next time period
- Move the segment range to the new values for the next time period

These procedures only work if you have the data and the indexes segmented identically, as "Creating a time-cyclic database" on page 155 outlines.

To detach a segment and attach it with new range values for next time period:

1. If you want to save the data, use the TMU to unload the segment to a file or tape.
2. Take the segment containing the oldest data offline.

   ```
   RISQL> alter segment s_1q00 of table sales offline;
   ```

3. Use one of the following statements to detach the segment from the table. This action removes all data from the segment, so make sure you do not need the data or save it in step 1 above.)
   - The following statement removes all rows from the table segment and the corresponding entries in the index segments.

     ```
     RISQL> alter segment s_1q00 of table sales detach;
     ```

   - The following statement removes all rows from the table segment and only the corresponding entries in the specified index segment.

     ```
     RISQL> alter segment s_1q00 of table sales detach
     override fullindexcheck on segments (star_1q00);
     ```

     The OVERRIDE FULLINDEXCHECK clause optimizes the DETACH operation when all index entries corresponding to the s_1q00 data segment are stored in the star_1q00 index segment. If your data and indexes are not identically segmented, an error message is issued. Omitting the OVERRIDE FULLINDEXCHECK clause causes the DETACH operation to scan every index segment to ensure that any index entries in other segments of the index are removed.

   The data that resided in the s_1q00 segment is now deleted from the database. The segment is detached from the table but still exists (empty) for reuse. The lower boundary of the next segment, s_2q00, automatically changes to cover the minimum boundary so that it now has the following range:

   ```
   min:DATE'2000-07-01'
   ```

4. Take the index segment corresponding to the oldest data offline.

   ```
   RISQL> alter segment star_1q00 of index sales_star offline;
   ```

5. Detach the index segment from the index.

   ```
   RISQL> alter segment star_1q00 of index sales_star detach;
   ```

   The detach table segment in step 3 removed all corresponding entries from the index segment. Therefore, the index segment is empty and does not need to be invalidated.

6. Rename the old data segment, which will hold data for the second quarter of 2002.

   ```
   RISQL> alter segment s_1q00 rename s_2q02;
   ```

7. Rename the old index segment.

   ```
   RISQL> alter segment star_1q00 rename star_2q02;
   ```

8. Make any other needed changes to the table and index segments. For example, change the maximum size or path of a PSU, or add a new PSU to the segment.
9. If you have not pre loaded the leading dimension table, modify the Period table by adding rows for the new quarter (second quarter of 2002).

   Tip: IBM recommends that you preload the leading dimension table with rows that you expect to use in the future.

   The Period table must contain rows corresponding to the days in the new quarter. Otherwise, new rows inserted in the Sales table would be discarded due to referential integrity failures. The following statements are examples of INSERT statements for the Period table:

```
RISQL> insert into period values
   (DATE'2002-04-01', 'APR', 2002, 2, 1);
...
RISQL> insert into period values
   (DATE'2002-05-01', 'MAY', 2002, 2, 2);
...
RISQL> insert into period values
   (DATE'2002-06-01', 'JUN', 2002, 2, 2);
...
```

   You can also use the TMU to populate the Period table with data for the second quarter of 2002. For an example, see step 6 on page 298 of the procedure ”To move the range to the new time period”.
10. Select from the dimension table to verify that the segmenting values have consecutive row numbers and to obtain the row numbers for the attach.
11. Attach the newly renamed data segment to the table.

```
RISQL> alter segment s_2q02 attach to table sales
   range (DATE'2002-04-01':DATE'2002-07-01');
```

    The range of thes_max segment automatically moves to the following range:

```
DATE'2002-07-01':max
```

    Attaching a segment automatically sets the segment to ONLINE mode.
12. Attach the newly renamed index segment to the table.

```
RISQL> alter segment star_2q02 attach to index sales_star
   range (821:899);
```

    Because the values in a STAR indexes are the row IDs of the referenced table, you must specify row IDs in the RANGE clause of the ALTER SEGMENT statement to attach the index segment.

13. You can now use the TMU or SQL INSERT statements to populate the Sales table with data for the second quarter of 2002 and build the corresponding index entries.

To move the range to the new time period:

1. If you want to save the data, use the TMU to unload the segment to a file or tape.
2. Clear the old data segment to enable the range to move.

   ```
   RISQL> alter segment s_1q00 clear;
   ```

   The CLEAR operation removes the data rows in the segment and all the index entries that reference the data rows. If any precomputed views exist on the Sales table and maintenance is turned on, the CLEAR operation rebuilds them.
3. Rename the old data segment, which will hold data for the second quarter of 2002.

   ```
   RISQL> alter segment s_1q00 rename s_2q02;
   ```
4. Rename the old index segments.

   ```
   RISQL> alter segment star_1q00 rename star_2q02;
   RISQL> alter segment target_mkt_1q00 rename target_mkt_2q02;
   ```
5. Make any other needed changes to the table and index segments. For example, change the maximum size or path of a PSU, or add a new PSU to the segment.
6. If you have not preloaded the leading dimension table, modify the Period table by adding rows for the new quarter (second quarter of 2002).

   Tip: IBM recommends that you preload the leading dimension table with rows that you expect to use in the future.

   The Period table must contain rows corresponding to the days in the new quarter. Otherwise, new rows inserted in the Sales table would be discarded due to referential integrity failures.

   To load the Period table, setup and execute a TMU load operation. A sample TMU control file `period.next.load` and sample data file `period.next.txt` follow:

   ```
   ::::::::::::::
   period.next.load
   ::::::::::::::
   load data
   inputfile '/aroma/period.next.txt'
   append
   discardfile '/discards/period_discards'
   discards 10
   into table period (
   ```

```
perkey  position(2:11) date 'YYYY-MM-DD',
month   position(13:17) character,
year    position(18:28) integer external,
quarter position(30:40) integer external,
tri     position(42:52) integer external);
::::::::::::::
period.next.txt
::::::::::::::
2000-04-01 APR 00000002000 00000000002 00000000001
2000-04-02 APR 00000002000 00000000002 00000000001
2000-04-03 APR 00000002000 00000000002 00000000001
2000-04-04 APR 00000002000 00000000002 00000000001
2000-04-05 APR 00000002000 00000000002 00000000001
2000-04-06 APR 00000002000 00000000002 00000000001
2000-04-07 APR 00000002000 00000000002 00000000001
2000-04-08 APR 00000002000 00000000002 00000000001
2000-04-09 APR 00000002000 00000000002 00000000001
2000-04-10 APR 00000002000 00000000002 00000000001
...
```

The following sample TMU command loads the Period table:

```
rb_tmu /qa/local/virginia/aroma_vjp/period.next.load username password
```

For more information about the TMU, see the *Table Management Utility Reference Guide*.

7. Move the range of the renamed data segment to the table.

```
RISQL> alter segment s_2q02 of table sales
range (DATE'2002-04-01':DATE'2002-07-01') move;
```

The RANGE MOVE clause first performs a detach operation and adjusts the range of the adjacent segment, s_2q00, to the following values:

```
min : DATE'2001-07-01'
```

Then the detached segment is reattached and the range of the adjacent segment, s_max, is adjusted.

```
s_2q02                      s_max
DATE-01-01':DATE-07-01'     DATE-07-01':max
```

When you move the ranges of a table segment, the RANGE MOVE operation automatically moves the ranges of the corresponding local index segments. The following excerpt of a query against RBW_SEGMENTS shows the moved range values of the segments that were created in Figure 34 through Figure 36 on pages 158 through 160:

```
RISQL> select substr(name, 1,17) as segname, npsus,
>    substr(minkey,1,5) as minkey, substr(maxkey,1,5) as maxkey,
>    totalfree, substr(iname, 1,17) as idxname,
>    substr(usage, 1,5) as usage, rowcount
>   from rbw_segments where tname = 'SALES' order by name;
SEGNAME     NPSUS MINKE MAXKE TOTALFREE IDXNAME         USAGE ROWCOUNT
```

```
s_2q02          2 20020 20020      2040 NULL            TABLE       0
s_2q00          2   MIN 20000      1528 NULL            TABLE   16871
s_3q00          2 20000 20001      1504 NULL            TABLE   17551
s_4q00          2 20001 20011      1456 NULL            TABLE   19138
...
star_1q00       1   MIN DEFAU      808 SALES_STAR_IDX   INDEX    NULL
star_2q00       1 DEFAU DEFAU      808 SALES_STAR_IDX   INDEX    NULL
star_3q00       1 DEFAU DEFAU      800 SALES_STAR_IDX   INDEX    NULL
star_4q00       1 DEFAU   MAX      776 SALES_STAR_IDX   INDEX    NULL
...
target_mkt_2q02 1   822   MAX      992 PERKEY_SALES_TIX INDEX    NULL
target_mkt_2q00 1   MIN   181      920 PERKEY_SALES_TIX INDEX    NULL
target_mkt_3q00 1   181   273      912 PERKEY_SALES_TIX INDEX    NULL
target_mkt_4q00 1   273   365      912 PERKEY_SALES_TIX INDEX    NULL
...
```

8. Move the newly renamed STAR index segment to the table.

```
RISQL> alter segment star_2q02 of index sales_star
range (821:899) move;
```

   The star_2q02 index segment is empty because the CLEAR operation in step 2 removed the index entries when the corresponding data rows were removed.

9. You can now use the TMU or SQL INSERT statements to populate the Sales table with data for the second quarter of 2002 and build the corresponding index entries.

Adding a new segment

As the end of Q2 2002 approaches, you want to add another segment between s_3q02 and s_maxto 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 2002, 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_3q02.

UNIX Only

```
create segment s_3q02
   storage '/disk10/s1' maxsize 200,
   storage '/disk10/s2' maxsize 200,
   storage '/disk10/s3' maxsize 200;
```

End of UNIX Only

Windows Only

```
create segment s_3q02
   storage 'c:\disk10\s1' maxsize 200,
   storage 'c:\disk10\s2' maxsize 200,
   storage 'c:\disk10\s3' maxsize 200;
```

End of Windows Only

2. Add the additional Perkey entries to the Period table to cover the additional date ranges for quarter 3 of 2002. Omitting this step causes referential integrity failures when the data for the new quarter is loaded. For example:

```
insert into period values
   (DATE'2002-07-01', 'JULY', 2002, 3, 2);
...
insert into period values
   (DATE'2002-08-01', 'AUG', 2002, 3, 2);
...
insert into period values
   (DATE'2002-09-01', 'SEPT', 2002, 3, 3);
...
```

3. Attach the s_3q02 segment to the Sales table, specifying the date range for quarter 3 of 2002.

```
alter segment s_3q02 attach to table sales
   range (DATE'2002-07-01':DATE'2002-10-01');
```

   The range of the *s_max* segment automatically moves to the range:

```
DATE'2002-10-01':max
```

   Attaching a segment automatically sets the segment to ONLINE mode.

4. Create a new index segment named star_3q02.

UNIX Only

```
create segment star_3q02
   storage '/disk10/star1' maxsize 200,
   storage '/disk10/star2' maxsize 200,
   storage '/disk10/star3' maxsize 200;
```

End of UNIX Only

Windows Only

```
create segment star_3q02
   storage 'c:\disk10\star1' maxsize 200,
   storage 'c:\disk10\star2' maxsize 200,
   storage 'c:\disk10\star3' maxsize 200;
```

End of Windows Only

5. Attach the new index segment to the STAR index.

   You can attach the new segment with one of the following segment specifications:

   ```
   alter segment star_3q02 attach to index sales_star
      range (899:992);

   alter segment star_3q02 attach to index sales_star
      range like segment (s_3q02);
   ```

6. Create new segments for all local TARGET indexes. The following example shows how to create a new index segment named target_mkt_3q02 for one TARGET index.

   UNIX Only

   ```
   create segment target_mkt_3q02
      storage '/disk10/target_mkt1' maxsize 200,
      storage '/disk10/target_mkt2' maxsize 200,
      storage '/disk10/target_mkt3' maxsize 200;
   ```

   End of UNIX Only

   Windows Only

   ```
   create segment target_mkt_3q02
      storage 'c:\disk10\target_mkt1' maxsize 200,
      storage 'c:\disk10\target_mkt2' maxsize 200,
      storage 'c:\disk10\target_mkt3' maxsize 200;
   ```

   End of Windows Only

7. Attach the new local index segment to the TARGET index.

   ```
   alter segment target_mkt_3q02 attach to
      index sales_target_mktkey
      range like segment (s_3q02);
   ```

   Only a TARGET or B-TREE index that is defined as local can be attached with the same segment range as the data.

The following figure illustrates the new segment ranges for the Sales table.

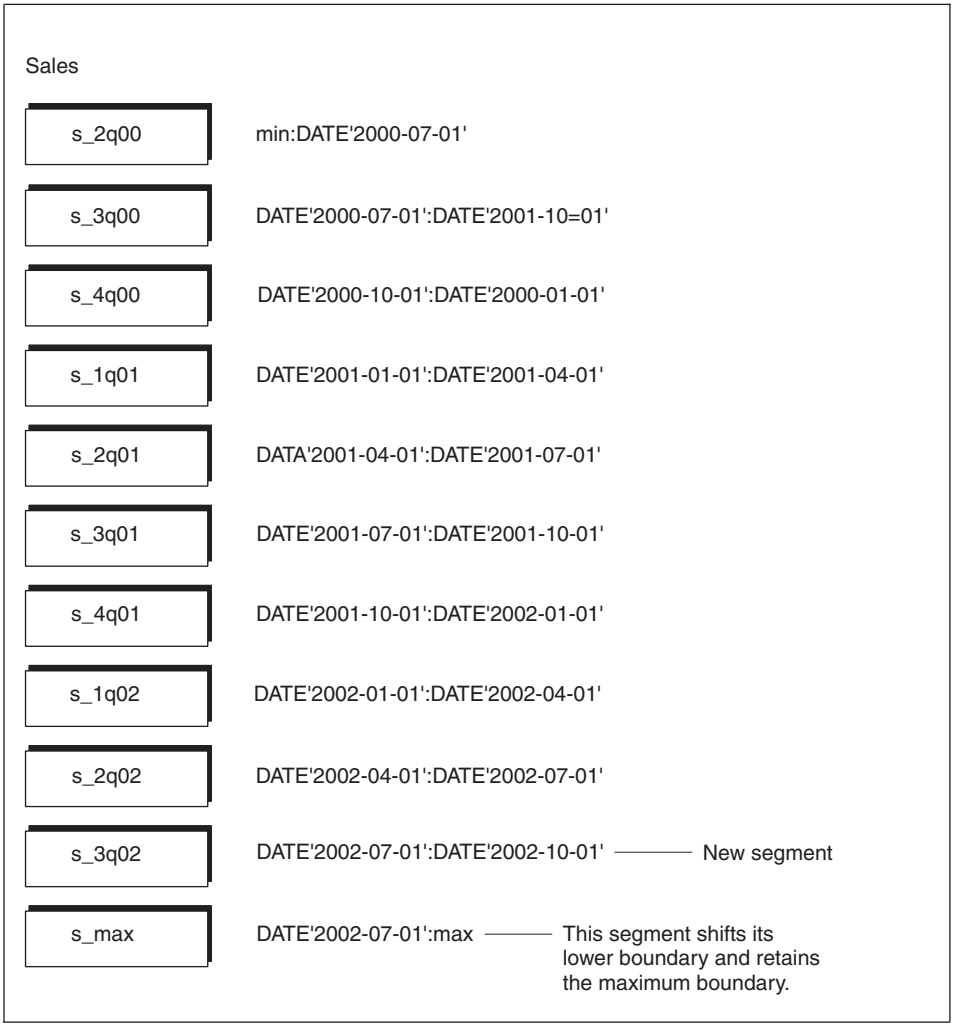


Figure 56. New segment ranges for sales table

Now you can load data into the new segment, using either the standard or the offline load procedure.

Deleting the oldest data

Now you need to remove data for the oldest quarter, the second quarter of 2000, which resides in the s_2q00 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. Set the s_2q00 segment to OFFLINE mode.

   ```
   alter segment s_2q00 of table sales offline;
   ```

3. Use one of these statements to detach the s_2q00 segment.

   ```
   alter segment s_2q00 of table sales detach;

   alter segment s_2q00 of table sales detach
      override fullindexcheck on segments (star_2q00);
   ```

   For a caution about OVERRIDE FULLINDEXCHECK, see "Rolling off and reusing data and index segments" on page 295.

   The data that resided in the s_2q00 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_3q00, automatically change to cover the minimum boundary with a range.

   ```
   min:DATE'2000-10-01'
   ```

4. Detach the star_2q00 index segment.

   ```
   alter segment star_2q00 of index sales_star detach;
   ```

5. You can either save these segments for reuse or drop them with DROP statements.

   ```
   drop segment s_2q00;
   drop segment star2q00;
   ```

The ranges are now as follows.

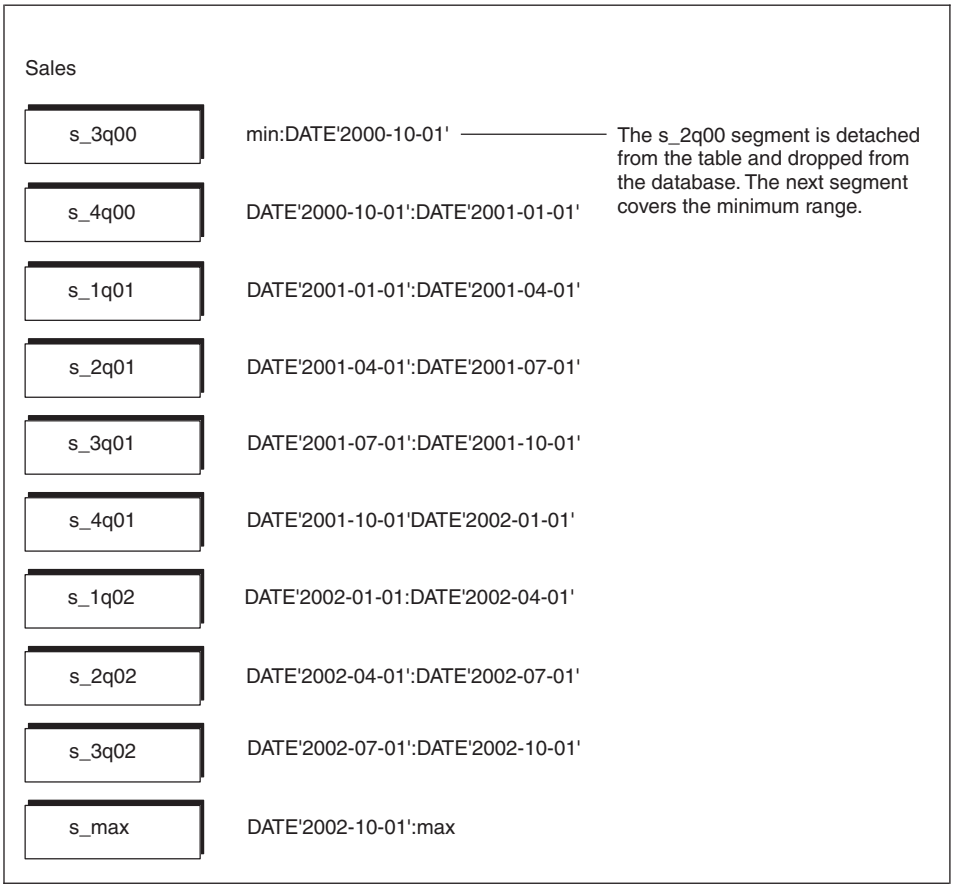


Figure 57. Segments and data ranges for sales table

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, see 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

IBM 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, see the *SQL Reference Guide*.

Note: ALTER TABLE statements are best performed with the IN PLACE USING *working segment* option, which writes the altered table to a predefined segment. If the ALTER TABLE operation fails to complete for some reason, it can be reset or resumed.

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, the column 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, the column's 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.

ALTER TABLE location

When you add or drop a column, you can specify where you want the modified table to be built during the ALTER TABLE operation:

- In one or more named segments (IN *segment_name*).

  This operation uses named segments to build the modified table. The number of segments that you specify must correspond to the number of segments in the original table.

  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, this option requires a larger amount of space than other ALTER TABLE operations.

- In a working segment (IN PLACE USING *working_segment_name*).

  This operation uses a single working segment to build the modified table. A *working segment* is a predefined segment that temporarily holds the altered data during an ALTER TABLE operation.

  This operation copies a table segment into the working segment, alters the data in memory, and then stores the altered table in the original segment. The same working segment is reused for all of the segments in the table. This option is recommended because it gives a high degree of recoverability if the ALTER TABLE operation fails.

- In the existing segments of the table itself (IN PLACE).

  This operation uses the existing segment or segments in the table to build the modified table. If the new row is larger than the original row (that is, the total width of the added columns is larger than the total width of the dropped columns), the segment must be large enough to hold all the new rows.

- In a default segment.

  This operation uses one default segment to build the modified table and only applies to tables that reside in a single named or default segment.

Recommendation: Use the IN PLACE USING *working_segment_name* option for most ALTER TABLE operations, especially those that add columns to large database tables. When you use a working segment, the operation can be resumed with a higher probability of success if it fails to complete for some reason. An IN PLACE operation without a working segment can never be reset and, in some cases, cannot be successfully resumed.

ALTER TABLE space requirements

If you are adding columns to a table, make sure that adequate space exists in both the table segments themselves and any named or working segments that

you create to perform the ALTER TABLE operation. Before you start the operation, run the following query to determine the current size of the table's largest segment:

```
select substr(rbw_segments.tname, 1, 15) as tablename,
  substr(rbw_segments.name, 1, 15) as segmentname,
  sum(rbw_storage.maxsize) as max_kb,
  sum(rbw_storage.used) as total_used_kb
from rbw_segments, rbw_storage
where rbw_segments.tname = 'SALES'
  and rbw_segments.id = rbw_storage.segid
  and rbw_segments.iname is null
group by tablename, segmentname;

TABLENAME       SEGMENTNAME          MAX_KB        TOTAL_USED_KB
SALES           DAILY_DATA1            2048            1056
SALES           DAILY_DATA2            2048            1080
```

Based on the result of this query, you can estimate the size that the altered table will grow to, then create the appropriate segments for the operation to use. In this example, the size of the working segment must be at least 1080 kilobytes to hold the segment with the larger number of used kilobytes.

Alternatively, you can start the ALTER TABLE operation and check the output of error message 693, which indicates the estimated segment size that the operation is likely to require. This approach is easier than trying to estimate the storage requirements yourself, but note that message 693 returns only estimated segment sizes; in some cases, the operation might require more space than the suggested estimate. If message 693 is not displayed, your working or named segments are large enough as they are and the operation will continue.

The following procedure demonstrates the use of message 693, as well as the working segment option for storing altered data.

To add a column using a working segment:

1. Create the working segment.

   ```
   create segment workseg1 storage 'sales_area1' maxsize 2000;
   ```

2. Add the column to the table.

   For example, suppose you want to add a new VARCHAR (75) column to the Sales table. You can use either the Administrator tool or enter the ALTER TABLE statement into a tool such as RISQL.

   - To use the Administrator tool, choose the **Manage > Tables** option in the top menu bar and select the **Alter Table** radio button.
   - To use a tool such as RISQL, enter a statement like this statement:

     ```
     alter table sales add column webid varchar(75) default 'New Sales web
     site identifier' with fillfactor 50 in_place using workseg1;
     ```

3. If necessary, increase the size of an existing table segment.

   If the existing table segment is not large enough to hold the new column, message 693 displays the number of kilobytes required:

   ```
   ** ERROR ** (693) Segment DAILY_DATA1 must be at least 2416Kbytes in size
   for this ALTER TABLE statement.
   ```

   You can increase the size of the second PSU in this segment with the following SQL statement:

   ```
   alter segment daily_data1 of table sales change maxsize 2 to 2416;
   ```

   If you run the ALTER TABLE statement again, it displays the size required for the second segment:

   ```
   ** ERROR ** (693) Segment DAILY_DATA2 must be at least 2472Kbytes in size
   for this ALTER TABLE statement.
   ```

   Increase the second segment size, and the ALTER TABLE statement now runs successfully.

   ```
   alter segment daily_data2 of table sales change maxsize 2 to 2472;
   alter table sales add column webid varchar(75) default 'New Sales web site
   identifier' with fillfactor 50 in_place using workseg1;
   ```

4. If you receive errors other than message 693 when you run the ALTER TABLE statement, see "Recovering from an interrupted ALTER TABLE operation" on page 316.

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, see 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, see "Setting the MAXSEGMENTS and MAXROWS PER SEGMENT options" on page 140.

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, see "Delete operations and cascaded deletes" on page 46.

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. IBM 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, see 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, see "Setting the VARCHAR column fill factor" on page 142.

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.

Important: 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.

  To determine how much work has been done, look in the log for certain messages. For an IN PLACE alter operation, message 514 will appear in the log. For an IN PLACE USING *working segment* alter operation, message 516 and 514 will both appear in the log.

  For example, the following messages indicate that the ALTER TABLE operation has successfully copied the first segment of the Sales table to the working segment and has altered all of the rows from the working segment back to the original segment:

```
516 Finished copy from segment DAILY_DATA1 to segment SALES_AREA1.
514 Finished alter from segment SALES_AREA1 to segment DAILY_DATA1.
```

- 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.

  Recommendation: If you are altering the table with the IN PLACE option, use a working segment to make recovery easier if the alter operation fails.

  To determine if you can resume the ALTER TABLE operation, query the RBW_TABLES catalog with the following SQL statement:

  ```
  select alter_pending from rbw_tables where name = 'table_name'
  ```

  If the value of `alter_pending` is Y, you can resume the alter operation. Otherwise, you must restart the alter from the first segment.
- 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.

Important: If you are altering the table with the IN PLACE option, make sure you have a current database backup before submitting the 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 the process viewer dialog (`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. Using the rb_cm utility for migrating between different versions of IBM Red Brick Warehouse is supported only if the data is transferred in TMU EXTERNAL format. For more information on the rb_cm utility, see 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.

Compatibility

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-based platform cannot be copied or moved to an HP-UX-based platform or vice versa.

With IBM Red Brick Warehouse Version 6.2 and later versions, you can copy a database that was created with a 32-bit version of the Red Brick server to a 64-bit version of the code on the same vendor platform. Versions of the Red Brick server prior to 6.2 did not support this level of compatibility. For

detailed information about migrating databases from 32-bit to 64-bit platforms, see the *Installation and Configuration Guide* and the *Release Notes*.

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.

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 Only

```
select segname, pseq, location from rbw_storage
   where substr(location, 1, 1) = '/';
```

End of UNIX Only

Windows Only

```
select segname, pseq, location from rbw_storage
   where substr(location, 1, 2) = ':'
      and (substr(location, 1, 3) = '\' ;
```

End of Windows Only

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.

Copying 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, see 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.

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.

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, see Appendix B, "Configuration File."

UNIX Only

To determine what other actions you must take, see the following table.

Change	Action required
SERVER SHMEM	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 REMOTE_TMU_LISTENER QUERYPROCS TOTALQUERYPROCS ADMIN ADVISOR_LOGGING READER_PRIORITY	• Stop and restart warehouse daemon. Do not change while warehouse daemon is running. • Use ipcrm to remove prior memory segment or reboot. (This step is necessary only after severe errors occur.)
SERVER_NAME MESSAGE_DIR LOCALE	Do not change.
PERFORMANCE_MONITOR_ MAXOPERATORS PERFORMANCE_MONITOR_ MAXSESSIONS	Stop and restart the performance monitor daemon (rbwpmond).
BAR_XBSA_LIB	Install and configure the storage manager that corresponds to the new library.
BAR_SM_USER	If specified user is not registered to the storage manager, register the new user to the storage manager.

Change	Action required
ARITHABORT COUNT_RESULT CHECK_TABLE_INDEX_DIRECTORY CHECK_REPORT_FILE_PERMISSIONS CROSS_JOIN EXPORT_DEFAULT_PATH EXPORT_DELIMITER EXPORT_MAX_FILE_SIZE FILE_GROUP FIRST_DAYOFWEEK GRANT_TEMP_RESOURCE_TO_ALL GROUP INFO_MESSAGE_LIMIT RBW_LOADINFO_LIMIT IGNORE_OPTICAL_INDEXES OPTICAL_AVAILABILITY ROWCOUNT ROWS_PER_SCAN_TASK ROWS_PER_FETCH_TASK ROWS_PER_JOIN_TASK ROWS_PER_TARGETJOIN_TASK FORCE_SCAN_TASKS FORCE_FETCH_TASKS FORCE_JOIN_TASKS FORCE_TARGETJOIN_TASKS FORCE_HASHJOIN_TASKS FORCE_AGGREGATION_TASKS PARTIONED_PARALLEL_AGGREGATION PARALLEL_SET_OPERATION TARGETJOIN_LOCAL_PREDICATES OLAP_APPROXIMATE_NUMERIC_FAST_ COMPUTATION ALLOW_POSSIBLE_DEADLOCK DEFAULT_DATA_SEGMENT DEFAULT_INDEX_SEGMENT DEFAULT_SEGMENT_SIZE DEFAULT_PSU_EXTEND_SIZE TEMPORARY_DATA_SEGMENT TEMPORARY_INDEX_SEGMENT	No action needed. Current database server processes do not recognize the change, but new processes do.

Change	Action required
OPTION ADVISOR_LOGGING PRECOMPUTED_VIEW_QUERY_REWRITE UNIFORM_PROBABILITY_FOR_ADVISOR USE_INVALID_PRECOMPUTED_VIEWS ADVISOR_MAXIMUM_CANDIDATE_ VIEWS PRECOMPUTED_VIEW_MAINTENANCE PRECOMPUTED_VIEW_MAINTENANCE_ ON_ERROR RANDOM_SAMPLE_MARGIN ADVISOR_SAMPLE_SIZE SEGMENTS IGNORE_PARTIAL_INDEXES PARTIAL_AVAILABILITY PERFORMANCE_MONITOR_COMMANDS_ LIMIT QUERY_MEMORY_LIMIT QUERY_TEMPSPACE parameters RESULT_BUFFER RESULT_BUFFER_FULL_ACTION VERSIONING TRANSACTION_ISOLATION_LEVEL	No action needed. Current database server processes do not recognize the change, but new processes do.
IDLE_TIMEOUT	No action needed. Current sessions do not recognize the change, but new sessions do.
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 XML_MAX_TASKS XML_TEMP_SPACE	No action needed. Current TMU processes do not recognize the change, but new processes do.
BAR_UNIT_SIZE BARTAPE	No action needed. Current backup and restore sessions (TMU processes) do not recognize the change, but new processes do.

Change	Action required
FILLFACTOR parameters INDEX_TEMPSPACE parameters PASSWORD parameters	No action needed. Current database server and TMU processes do not recognize the change, but new processes do.
ADMIN parameters REPORT_INTERVAL Other ADMIN parameters	No action needed. Current database server and TMU processes do not recognize the change, but new processes do. Stop and restart warehouse daemon or use the ALTER SYSTEM statement to restart logging and admin processes.

End of UNIX Only

Windows Only

To determine what other actions you must take, see the following table.

Change	Action required
SERVER MAX_SERVERS CLEANUP_SCRIPT LOGFILE_SIZE REMOTE_TMU_LISTENER QUERYPROCS TOTALQUERYPROCS ADMIN ADVISOR_LOGGING UNIFIED_LOGON MAX_ACTIVE_DATABASES PROCESS_CHECKING_INTERVAL READER_PRIORITY	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.
PERFORMANCE_MONITOR_MAXOPERATORS PERFORMANCE_MONITOR_MAXSESSIONS	Stop and restart the performance monitor daemon (rbwpmond).
BAR_XBSA_LIB	Install and configure the storage manager that corresponds to the new library.

Change	Action required
BAR_SM_USER	If specified user is not registered to the storage manager, register the new user to the storage manager.
IDLE_TIMEOUT	No action needed. Current sessions do not recognize the change, but new sessions do.

Change	Action required
ARITHABORT ALLOW_POSSIBLE_DEADLOCK COUNT_RESULT CHECK_TABLE_INDEX_DIRECTORY CHECK_REPORT_FILE_PERMISSIONS CROSS_JOIN DEFAULT_DATA_SEGMENT DEFAULT_INDEX_SEGMENT DEFAULT_SEGMENT_SIZE DEFAULT_PSU_EXTEND_SIZE UNIFORM_PROBABILITY_FOR_ADVISOR EXPORT_DEFAULT_PATH EXPORT_DELIMITER EXPORT_MAX_FILE_SIZE FIRST_DAYOFWEEK GRANT_TEMP_RESOURCE_TO_ALL GROUP 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 ROWS_PER_TARGETJOIN_TASK FORCE_SCAN_TASKS FORCE_FETCH_TASKS FORCE_JOIN_TASKS FORCE_HASHJOIN_TASKS FORCE_TARGETJOIN_TASKS FORCE_AGGREGATION_TASKS PARTIONED_PARALLEL_AGGREGATION PARALLEL_SET_OPERATION TARGETJOIN_LOCAL_PREDICATES OLAP_APPROXIMATE_NUMERIC_FAST_ COMPUTATION TEMPORARY_DATA_SEGMENT TEMPORARY_INDEX_SEGMENT	No action needed. Current database server threads do not recognize the change, but new ones do.

Change	Action required
OPTION ADVISOR_LOGGING PRECOMPUTED_VIEW_QUERY_REWRITE USE_INVALID_PRECOMPUTED_VIEWS ADVISOR_MAXIMUM_CANDIDATE_ VIEWS PRECOMPUTED_VIEW_MAINTENANCE PRECOMPUTED_VIEW_MAINTENANCE_ON_ERROR RANDOM_SAMPLE_MARGIN ADVISOR_SAMPLE_SIZE SEGMENTS IGNORE_PARTIAL_INDEXES PARTIAL_AVAILABILITY PERFORMANCE_MONITOR_COMMANDS_LIMIT 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 the change, but new ones do.
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 XML_MAX_TASKS XML_TEMP_SPACE	No action needed. Current TMU processes do not recognize the change, but new ones do.
BAR_XBSA_LIB	Install and configure the storage manager that corresponds to the new library.
BAR_SM_USER	If specified user is not registered to the storage manager, register the new user to the storage manager.

Change	Action required
FILLFACTOR parameters INDEX_TEMPSPACE parameters PASSWORD parameters	No action needed. Current database server threads and TMU processes do not recognize the change, but new ones do.
ADMIN parameters REPORT_INTERVAL Other ADMIN parameters	No action needed. Current threads do not recognize the change, 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.

End of Windows Only

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 or the DBA edits the `rbw.config` file. The values displayed apply to the current session. For more information, see Appendix C, "System Tables and Dynamic Statistic Tables."

Monitoring and controlling a database server

IBM Red Brick Warehouse 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."

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 IBM Red Brick Warehouse 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, see the *Installation and Configuration Guide*.

Restarting the log and administration daemons: If the rbwlogd or rbwadmd daemon goes down while the rbwapid daemon is still running, you can restart each daemon separately. You do not have to stop and restart the rbwapid daemon. To restart rbwlogd or rbwadmd:

1. Make sure the RB_CONFIG and RB_HOST environment variables are set correctly in your environment.
2. Enter rbwlogd or rbwadmd on the command line:

   ```
   % $RB_CONFIG/bin/rbwlogd
   % $RB_CONFIG/bin/rbwadmd
   ```

   The system uses the value of RB_HOST as a parameter to these commands when it starts the daemons. This is useful for identifying which warehouse daemon the administration and log daemons belong to when there is more than one warehouse daemon running.

To stop the administration daemon, use the ALTER SYSTEM TERMINATE ADMIN DAEMON statement.

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. IBM Customer Support also uses these files.

For more information on log files, see "Event logging" on page 234.

There are log files associated with the Vista Advisor. For more information about Advisor log files, see "Advisor log files" on page 245.

Monitoring and controlling a database server on Windows

The IBM Red Brick Warehouse 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 the database service.

Database service threads

As long as the service is running, all of the threads should be running. 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."

To stop the Red Brick database service:

1. Use the ALTER SYSTEM QUIESCE statement to prevent new connections or commands from executing.
2. Use the rbwshow.exe script to see if any active connections exist.
3. If no active connections exist, stop the database service using the Control Panel Services.

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.

If a thread does go down, restart the service from the Control Panel. If the IBM Red Brick Warehouse 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.

Log files

Several log files available in the redbrick directory allow you to monitor the Red Brick 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. IBM 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, see "Event logging" on page 234.

There are log files associated with the Vista Advisor. For more information about Advisor log files, see "Advisor log files" on page 245.

Determining version information

If you need to contact the IBM Customer Support, determine the complete version identification information for the IBM Red Brick Warehouse 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 IBM Red Brick Warehouse 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, see 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, see the *IBM Red Brick 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, see "Setting segment and partial availability behavior" on page 138.
- 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 *IBM Red Brick 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.

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
   ```

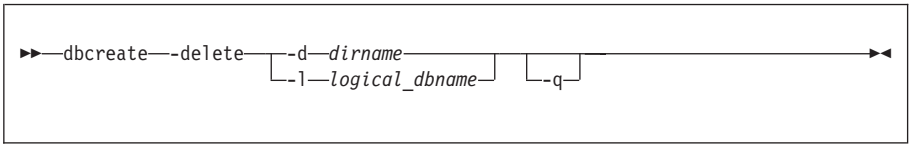
   In the example, pathname is a full directory path specification to the database directory, such as /disk1/db_sales_92.
6. Use operating-system commands (`rm`, `rmdir`) to remove:
   - Any files or directories remaining in the database directory and then remove the directory.
   - Any segment directories 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.

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:



-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.

Managing memory requirements for a database catalog

When a Red Brick process or thread opens the system catalog for a database, it loads the system catalog file RB_DEFAULT_IDX into memory. The system catalog grows over time as you create more database objects (PSUs, segments, tables, views, and so on). As you drop objects, the database frees space within the system catalog. However, if an active object exists in a block between several free blocks, the system catalog removes only the free blocks from the end to the last active block. The free blocks in between the active blocks are not removed, but they can be reused. You can use the rb_syscompact utility to compact the system catalog file RB_DEFAULT_IDX. Compacting a file rearranges blocks to save space when the catalog is loaded into memory.

Operational considerations for compacting a system catalog

To determine the amount of free space currently in the system catalog, use the **show** option of the rb_syscompact utility. For example, from the system prompt:

```
c:\redbrick_dir\bin: rb_syscompact -show username password
(C) Copyright IBM Corp. 1991-2003. All rights reserved.
Version 06.30.0000(0)
Size 581 blocks, Used 571 blocks, Free 480 blocks, 84 percent free.
```

The percent free value is the ratio of the free blocks to used blocks and can help you determine if a large number of free blocks exist. This example shows a very large percentage (84) of free blocks.

Compact the system catalog only if the **show** option shows a significant number of free blocks. After compaction, there is no free space left in the catalog, and if a large number of objects are created again, the catalog will immediately grow and consume more memory during query execution. Do not use the compaction utility too aggressively if you frequently create new database objects.

For example, consider the following scenario:

- The system catalog was just compacted and a user submits a query. Red Brick Warehouse opens the smaller-sized system catalog to run the query.
- Another user executes a large number of DDL statements that cause the system catalog to grow.
- The first user runs another query, causing the database server to reopen the new, enlarged catalog. In turn, a new chunk of memory must be allocated

to accommodate the new size. This additional allocation can cause the process heap to grow, and more memory might be used than if the catalog originally contained some free space.

Removing empty blocks from the system catalog

The rb_syscompact utility removes any empty blocks within the system catalog that result from dropped objects. You must have DBA authorization to execute the rb_syscompact utility.

Important: To run the rb_syscompact utility on Linux platforms, you must be logged into the machine as the Linux user who installed IBM Red Brick Warehouse and your database username must have DBA authorization. For example, if user `redbrick` installed the software, you must log into your Linux machine as redbrick. If user `system` has DBA authorization, you can then run the utility as follows:

```
% rb_syscompact system password
```

To compact the system catalog for a database:

1. Ensure that the following environment variables are set:
   - RB_CONFIG

     The pathname to the directory that contains the `rbw.config` file.
   - RB_HOST

     The logical name used to identify the warehouse daemon process on UNIX or the Red Brick Warehouse service on Windows.
   - RB_PATH (optional)

     The logical database name, as defined in the `rbw.config` file by a pathname to a database directory.

     Note: If this variable is not set, you must specify the database name with the `-d` option of the rb_syscompact utility.
2. Connect to and quiesce the database to prevent other users from connecting to the database. Enter the following SQL statement in a RISQL session (or use the Manage System wizard in the Red Brick Warehouse Administrator tool):

   ```
   alter system quiesce;
   ```

   Alternatively, you can connect to the ADMIN database and specify the database that you want to quiesce by using the `database` option. For example:

   ```
   brick % $RB_CONFIG/bin/risql -d ADMIN username password
   (C) Copyright IBM Corp. 1991-2003. All rights reserved.
   RISQL Entry Tool Version 06.30.0000(0)
   RISQL> alter system quiesce database aroma;
   ```

where aroma is the logical name of the database. If other connections exist when you try to run the rb_syscompact utility, you receive the following error message:

```
** ERROR ** (12720) Cannot perform catalog compaction because other users
are connected to the database.
```

Note: If no other users are connected to the database and you do not quiesce it, the rb_syscompact utility quiesces the database for you and enables logins after compaction completes. However, IBM strongly recommends that you quiesce the database explicitly before using the compaction utility.

3. Compact the system catalog for the quiesced database:

```
c:\redbrick_dir\bin: rb_syscompact username password
(C) Copyright IBM Corp. 1991-2003. All rights reserved.
Version 06.30.0000(0)
Size 581 blocks, Used 571 blocks, Free 480 blocks, 84 percent free.
Final size 101 blocks, Used 91 blocks.
```

The rb_syscompact utility displays either a completion or failure message and then records a message in the log file when it performs compaction. The above completion message shows a difference of 10 blocks between the Size value 101 and Used value 91 even when 0 blocks are free. Red Brick Warehouse keeps 10 blocks on reserve for future growth.

4. Display the amount of free space in the system catalog after compaction:

```
c:\redbrick_dir\bin: rb_syscompact -show username password
(C) Copyright IBM Corp. 1991-2003. All rights reserved.
Version 06.30.0000(0)
Size 101 blocks, Used 91 blocks, Free 0 blocks, 0 percent free.
```

It is also useful to compare the physical size of the catalog file before and after compaction. You can check that the compacted catalog is smaller by running a simple ls command against the RB_DEFAULT_IDX and RB_DEFAULT_IDX.bak files in the database directory:

```
brick% cd $RB_PATH
brick% ls -l RB_DEFAULT_IDX*
-rw------- 1 redbrick   360448 Oct  1 00:17 RB_DEFAULT_IDX
-rw------- 1 redbrick   453216 Oct  1 00:17 RB_DEFAULT_IDX.bak
```

5. Change the state of the quiesced database to active:

```
alter system resume;
```

If you are connected to the ADMIN database, use this command:

```
alter system resume database aroma;
```

where aroma is the logical name of the database.

Recovery considerations for system catalog compaction

The rb_syscompact utility verifies that the system catalog does not have any obvious problems before it starts compaction. If this verification fails, you receive the following error message:

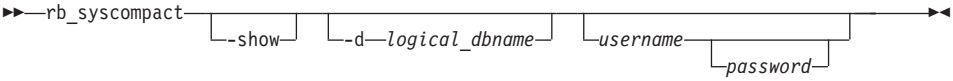
```
** ERROR ** (12710) Catalog sanity check failed. Compaction aborted.
```

The system catalog is one of the most important files for each Red Brick Warehouse database. Because loss of this file would render the database useless, the rb_syscompact utility makes a copy of the current file before making any changes. The utility creates the backup file `RB_DEFAULT_IDX.bak` in the same directory as the original file.

If the rb_syscompact utility finds that the system catalog is corrupt during compaction, it restores the catalog file `RB_DEFAULT_IDX` from the backup file `RB_DEFAULT_IDX.bak`. However, if the compaction fails because of a system or power failure, or because you cancel the utility or issue the `kill` command, then you must manually rename the backup file `RB_DEFAULT_IDX.bak` to `RB_DEFAULT_IDX`.

Important: If you repeatedly compact the catalog, the backup file is overwritten every time, which can make recovery difficult.

rb_syscompact syntax



-show	Displays the size statistics of the system catalog. When you use the **show** option, the catalog is not compacted and the database does not need to be quiesced first.
-d *logical_dbname*	Specifies the logical name of the database to be compacted, as defined in the `rbw.config` file. If you do not specify this option, you must set the RB_PATH environment variable to indicate the database name.
username, password	Specifies the database username and password.

For examples, see “Removing empty blocks from the system catalog” on page 340.

Appendix A. Example: Building a database

This appendix illustrates how to build a database using Aroma, the sample database included with IBM Red Brick Warehouse. 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 IBM Red Brick Warehouse. 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.

© Copyright IBM Corp. 1996, 2004

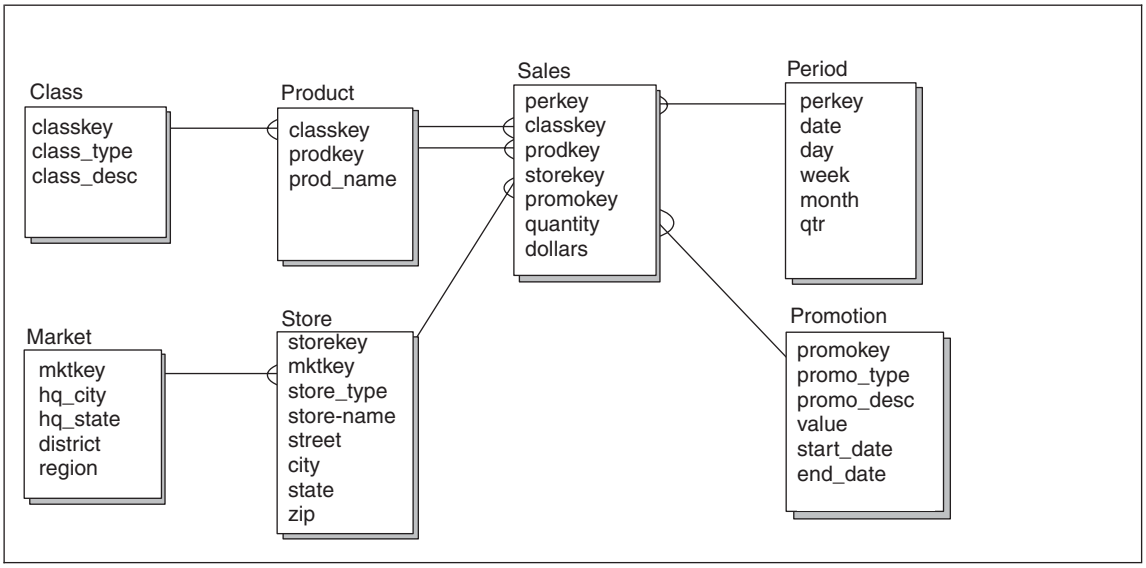


Figure 58. Schema of Aroma database

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 IBM Red Brick Warehouse. 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 IBM Red Brick Warehouse 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 IBM Red Brick Warehouse. You must be logged in as redbrick for the administrative tasks involved in creating a database.

UNIX Only

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.

End of UNIX Only

Windows Only

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 redbrickuser'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`.

End of Windows Only

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.

To create the database directory (Unix only):

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
   ```

To create the database directory (Windows only):

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.

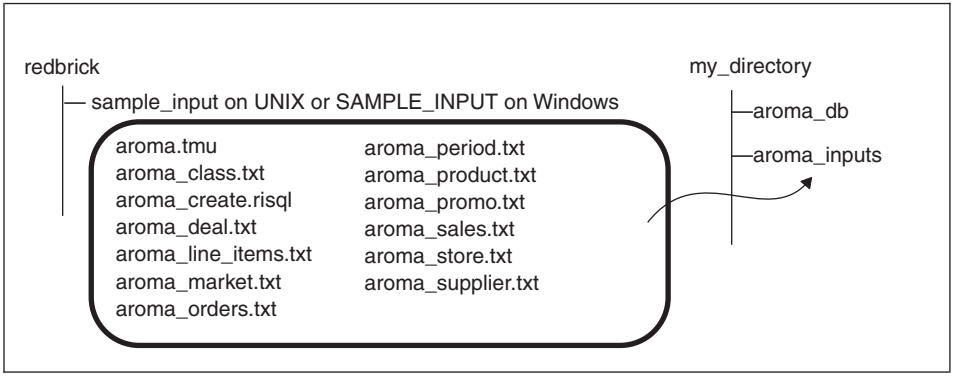


Figure 59. Sample database

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 Only

```
$ 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
```

End of UNIX Only

Windows Only

```
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
```

End of Windows Only

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, see Chapter 4, "Planning a Database Implementation," and Chapter 5, "Creating a database," on page 125.

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 IBM Corp. 1991-2002. All rights reserved.
   RISQL Reporter Version 06.20.0000(0)
   Please type password: xxxxxx, where xxxxxx is your 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,1,11)as NAME, type, substr(creator,1,7) as CREATOR, id,
maxsegments, maxrows_per_seg, maxsize_rows, intact from rbw_tables where id > 0;

NAME        TYPE     CREATOR    ID  MAXSEGMENTS MAXROWS_PER MAXSIZE_ROW INTA
DEAL        TABLE    SYSTEM      6          1        NULL    38797016    Y
ORDERS      TABLE    SYSTEM      9          1        2000    24641348    Y
SALES       TABLE    SYSTEM     11          2       60000      134640    Y
PERIOD      TABLE    SYSTEM      8          1        2500    69205488    Y
SUPPLIER    TABLE    SYSTEM      7          1        NULL    17563514    Y
STORE       TABLE    SYSTEM      2          1        2500    18087798    Y
PROMOTION   TABLE    SYSTEM      5          1        2500    35127028    Y
PRODUCT     TABLE    SYSTEM      4          1        2500    35127028    Y
LINE_ITEMS  TABLE    SYSTEM     10          1        2000    61079086    Y
CLASS       TABLE    SYSTEM      3          1        NULL    27000626    Y
MARKET      TABLE    SYSTEM      1          1        NULL    24641348    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,1,19) as NAME, substr(tname,1,10) as TNAME, type,
 substr(cname,1,11) as COLUMN_NAME, substr(creator,1,7) 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 path name.
- 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 IBM Corp. 1991-2003.
All rights reserved.
Version 06.20.0000(0)
** 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
```

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 IBM Red Brick Warehouse 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 IBM Red Brick Warehouse, 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 322. For descriptions of each file element, refer to "Description of file elements" on page 377.

```
##########################################################################
#
#(C) Copyright IBM Corp. 1991-2003. All rights reserved.
#
```

© Copyright IBM Corp. 1996, 2004

UNIX Only

```
# This file contains various parameters for the Red Brick Warehouse
# Daemon (rbwapid), Server (rbwsvr), and Table Management Utilities
# (rb_tmu and rb_ptmu)
```

End of UNIX Only

Windows Only

```
# This file contains various parameters for the Red Brick Warehouse Service.
# (rbwapid, rbwadmd, rbwlogd, rbwsvr threads), and Table Management Utility
# (rb_tmu AND rb_ptmu)
```

End of Windows Only

```
#
# The notation {ON | OFF} means use either ON, or OFF, etc.
# The default is the first option shown
#
```

UNIX Only

```
# 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 <IPC_key>

# The following is used to specify the communication port for connections
# to the Red Brick Warehouse rbwapid daemon.
<RB_HOST_NAME> SERVER <host>:<port_number>
```

End of UNIX Only

```
# The following value controls the maximum number of concurrent Red Brick
# Warehouse Server sessions that can be started by the rbwapid daemon.
RBWAPI MAX_SERVERS 10

# Daemon Start-up user exit script for cleaning up spill files.
```

UNIX Only

```
RBWAPI CLEANUP_SCRIPT <redbrick_dir>/bin/rb_sample.cleanup
```

End of UNIX Only

Windows Only

```
RBWAPI CLEANUP_SCRIPT <redbrick_dir>\RBW\BIN\rbclean.bat

# Unified Logon for Windows NT
#RBWAPI UNIFIED_LOGON {OFF | ON}

# On NT, the following value gives the stack size for the server threads
# ZERO gives NT default, other values specified as 100K, 1M, 20 etc.
#RBWAPI STACK_SIZE 0
```

End of Windows Only

UNIX Only

```
# The following value can be used to specify the location of the Red Brick
# Warehouse server image
RBWAPI SERVER_NAME <redbrick_dir>/bin/rbwsvr
```

End of UNIX Only

```
# 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)
#RBWAPI PROCESS_CHECKING_INTERVAL 10

# The maximum number of active databases
#RBWAPI MAX_ACTIVE_DATABASES 30

# Remote TMU Listener {ON | OFF}
# Default always ON unless specified OFF
RBWAPI REMOTE_TMU_LISTENER ON

# Message base details (location and language)
```

Windows Only

```
NLS_LOCALE MESSAGE_DIR C:\RedBrick\RBW\MESSAGES
```

End of Windows Only

UNIX Only

```
NLS_LOCALE MESSAGE_DIR <redbrick_dir>/messages
```

End of UNIX Only

```
NLS_LOCALE LOCALE Enlish_UnitedStates.US-ASCII@Binary

# First day of the week. 1=Sunday, 2=Monday, ... 7=Saturday
# Use this to override the default first day of the week.
#NLS_LOCALE FIRST_DAYOFWEEK <1-7>

# 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
#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 {OFF | ON_WITH_CORR_SUB | ON}

# 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}

# Allow pkfk joins in dimension preplans for starjoin
#OPTION PREPLANS_HAVE_ONLY_PKFK_JOINS {OFF | ON}

# Set limit on the number of advisor candidate views to process
#OPTION ADVISOR_MAXIMUM_CANDIDATE_VIEWS 20

# 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 or with high recoverability
#OPTION TMU_VERSIONING {OFF | ON | RECOVER}

# Defaults for TMU periodic commit
#OPTION TMU_COMMIT_RECORD_INTERVAL { <ROWCOUNT> | OFF }
#OPTION TMU_COMMIT_TIME_INTERVAL { <MINUTES> | OFF }

# EXPORT default directory
```

Windows Only

```
#OPTION EXPORT_DEFAULT_PATH c:\temp
```

End of Windows Only

UNIX Only

```
#OPTION EXPORT_DEFAULT_PATH /tmp
```

End of UNIX Only

```
# 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

# Specifies the maximum amount of memory that will be allocated to
# TMU load or reorg job.
#TUNE TMU_MAX_BUFFERS <number>
# Specifies the fraction of the amount of buffers specified by
```

```
# TMU_MAX_BUFFERS that is allotted to conversion tasks.
#TUNE TMU_CONVERSION_BUFFER_PERCENT <0 to 100>

# Specifies the fraction of the amount of buffers specified by
# TMU_MAX_BUFFERS that is allotted to loader output task.
#TUNE TMU_OUTPUT_BUFFER_PERCENT <0 to 100>

# Specifies the fraction of the amount of buffers specified by
# TMU_MAX_BUFFERS that is allotted to index tasks.
#TUNE TMU_INDEX_BUFFER_PERCENT <0 to 100>

# Tuning parameter for parallel query
#TUNE ROWS_PER_SCAN_TASK 2147483647
#TUNE ROWS_PER_FETCH_TASK 2147483647
#TUNE ROWS_PER_JOIN_TASK 2147483647
#TUNE ROWS_PER_TARGETJOIN_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>}
#TUNE FORCE_TARGETJOIN_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 {OFF | ON}

# Enable parallelism for queries that contain UNION, INTERSECT, and EXCEPT
#TUNE PARALLEL_SET_OPERATION {OFF | ON}

# 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
```

Windows Only

```
#TUNE INDEX_TEMPSPACE_DIRECTORY c:\temp
```

End of Windows Only

UNIX Only

```
#TUNE INDEX_TEMPSPACE_DIRECTORY /tmp
```

End of UNIX Only

```
# 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
```

Windows Only

```
#TUNE QUERY_TEMPSPACE_DIRECTORY c:\temp
```

End of Windows Only

UNIX Only

```
#TUNE QUERY_TEMPSPACE_DIRECTORY /tmp
```

End of UNIX Only

```
# 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 SAMPLE_MARGIN { <zero_or_positive float> | ON | OFF }

# Sample size (percentile) to be used by advisor for candidate view analysis.
#TUNE ADVISOR_SAMPLE_SIZE <0.0 to 100.0>

# Process local (non-join) predicates in TargetJoin, if possible
#OPTION TARGETJOIN_LOCAL_PREDICATES {OFF | ON}

# Allow single column B-Tree indexes to be considered for TARGET join plans
#OPTION ALLOW_SINGLE_COLUMN_BTREE_IN_TARGETJOIN {OFF | ON}

# Use fast incremental computation for olap functions on approximate datatypes
#OPTION OLAP_APPROXIMATE_NUMERIC_FAST_COMPUTATION {ON | OFF}

# Enable/disable mmapping of dimension tables/indexes for queries
#OPTION QUERY_MMAP {ON | OFF}

# Maximum amount of query mmap memory
#TUNE QUERY_MMAP_LIMIT 5M

########################################################
#
#  XML section:
#
########################################################

# Directory to place temporary XML spill files.
# If left unset, defaults to directory set by QUERY_TEMPSPACE_DIRECTORY.
#OPTION XML_TEMP_SPACE <temporary_spill_space_for_xml_parsing>
# To turn off nested xml input semantics, default is ON.
# If off, repeating elements in the same process unit will give
# error #9019 as in 6.20.
#OPTION NESTED_XML_INPUT {ON | OFF}
# The number of XML parallel parsing tasks to be spawned. Default is 1 (serial)
#TUNE XML_MAX_TASKS <no_xml_parsing_tasks>

# To tune memory allocated to *each* of the parallel row mapper tasks.
# Default is 1M. range is 64k to 256M.
# If nested iterations causes consumption of memory more than this
```

```
# then error #9052 would result and parsing would stop.
#TUNE NESTED_XML_MEMORY_LIMIT <nnn>K|M

############################################################
#
#  BACKUP/RESTORE section:
#
############################################################
# size of a bar unit. Default value is 256M.
#TUNE BAR_UNIT_SIZE 256M
```

UNIX Only

```
# logical tape device name mappings. There is no default
# value. This entry is required if TAPE device is used
# for backu/restore.
#BARTAPE dev1 /dev/rmt/0n
```

End of UNIX Only

```
# The following two parameters config the user name
# and the XBSA library path of a XBSA storage manager
# There are no default values. These two entries are required
# if a XBSA storage manager is used for backup/restore.
#OPTION BAR_SM_USER  redbrick
```

Windows Only

```
#OPTION BAR_XBSA_LIB c:\progra~1\bin\libbsa.ddl
```

End of Windows Only

UNIX Only

```
#OPTION BAR_XBSA_LIB /local/lib/xbsa.lib
```

End of UNIX Only

```
############################################################
#
#  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
```

UNIX Only

```
#ADMIN   RENICE_COMMAND  <full_path_of_a_renice_command>
```

End of UNIX Only

```
# 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 1
# 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
# Allow login name to be part of password
#PASSWORD ALLOW_LOGNAME ON
# User must change the initial password during first login session
#PASSWORD CHANGE_INITIAL OFF
# Whether changes to the numerals in password need to be complex
#PASSWORD COMPLEX_NUMERIC OFF

##########################################################
#
```

```
# Query Performance Monitor section
#
#########################################################
# maximum number of sessions that can be concurrently monitored
# OPTION PERFORMANCE_MONITOR_MAXSESSIONS  20
# maximum number of commands that can be stored in the performance
# monitor DST tables
# OPTION PERFORMANCE_MONITOR_COMMANDS_LIMIT  100
# maximum number of operators within a query that can be monitored
# OPTION PERFORMANCE_MONITOR_MAXOPERATORS  100
```

Windows Only

```
#########################################################
#
#  NETWORK section:  Add additional entries as services
#                     are created
#
#########################################################

<RB_HOST_NAME> SERVER <host>:<port_number>
```

End of Windows Only

```
#########################################################
#
#  DATABASE section:  Add additional entries as databases
#                     are created
#
#########################################################
# Logical database name mappings
```

UNIX Only

```
DB AROMA <redbrick_dir>/aroma_dir
DB ADMIN <redbrick_dir>/admin_dir
```

End of UNIX Only

Windows Only

```
DB AROMA <redbrick_dir>\aroma_db
DB ADMIN <redbrick_dir>\admin_db
```

End of Windows Only

Description of file elements

This section provides a brief description of the `rbw.config` file entries present when IBM Red Brick Warehouse 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 322.

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 IBM Red Brick Warehouse 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 IBM Red Brick Warehouse. 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. Default: 10 (base 10 integer) For information on how to calculate the IPC key range using the values of SHMEM and MAX_SERVERS, refer to the *Installation and Configuration Guide*

File element	Description
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. For more information on the sample scripts, see "Removing temporary files" on page 119.
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
RBWAPI STACK_SIZE (Windows)	Specifies the stack size for the server threads. Default: 0
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.
RBWAPI REMOTE_TMU_LISTENER	When set to ON, enables the server to respond to requests from the remote TMU. For more information on this parameter, see the *Table Management Utility Reference Guide*. Default: ON
NLS_LOCALE MESSAGE_DIR	Specifies the directory used for the error message files. Default: ./messages on UNIX or .\messages on Windows

File element	Description
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.
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.
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

File element	Description
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
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 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`

File element	Description
OPTION PREPLANS_HAVE_ONLY_PKFK_JOINS	Specifies whether dimension tables that are not joined across primary-key foreign-key columns are included in STARjoin preliminary plans. Default: OFF
OPTION ADVISOR_MAXIMUM_ CANDIDATE_VIEWS	Specifies the maximum number of candidates that each analysis can generate. Use this setting to control the number of candidates that are evaluated to reduce the elapsed time of the Advisor analysis. Default: 20
OPTION VERSIONING	Starts DML statements as versioning transactions.
OPTION TRANSACTION_ISOLATION_LEVEL	Sets the isolation level when acquiring locks for DML statements.
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`

File element	Description
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`
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

File element	Description
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
OPTION TARGETJOIN_LOCAL_PREDICATES	Specifies whether non-join predicates can be processed as part of a TARGETjoin operation. Default: OFF
OPTION ALLOW_SINGLE_COLUMN_BTREE_ IN_TARGETJOIN	Enables or disables the use of single-column B-TREE indexes in TARGETjoin plans. When this command is set to ON, a TARGETjoin plan can make use of any combination of single-column TARGET indexes, single-column B-TREE indexes, and multi-column B-TREE indexes. By default, TARGETjoin query processing relies on TARGET indexes defined on single-column foreign keys and B-TREE indexes on multi-column foreign keys. Default: OFF
OPTION QUERY_MMAP	Enables or disables memory mapping during query execution for the current session. For more information on this parameter, see the *Query Performance Guide*. Default: ON
OPTION BAR_SM_USER	Specifies the XBSA storage manager user name. This parameter and the BAR_XBSA_LIB parameter are required if an XBSA storage manager is used for backup and restore. For more information on this parameter, see the *Table Management Utility Reference Guide*. Default: none

File element	Description
OPTION BAR_XBSA_LIB	Specifies the X/Open Backup Services API (XBSA) library path of an XBSA storage manager. This parameter and the BAR_SM_USER parameter are required if an XBSA storage manager is used for backup and restore. For more information on this parameter, see the *Table Management Utility Reference Guide*. Default: none
BARTAPE	Maps the logical name to the physical name of a tape device. Required if a tape device is used for backup and restore. Specify multiple entries for multiple devices with one tape device per BARTAPE parameter. Default: not set

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 131071 blocks on 32-bit platforms and from 128 blocks to 8388607 blocks on 64-bit platforms. For more information on this parameter, see the *Table Management Utility Reference Guide*. Default: 128 blocks
TMU_MAX_BUFFERS	Specifies the size of the TMU buffer cache, in 8K blocks. This parameter represents the total memory allocation for all of the processing required to complete a LOAD DATA or REORG operation. When an operation runs in parallel, the buffers are shared among the conversion, output, and index tasks. Default: 10% of the physical memory on the machine, up to a maximum of 256MB for OPTIMIZE OFF loads and 128MB for OPTIMIZE ON loads.
TMU_CONVERSION_BUFFER_PERCENT	Specifies the percentage of the TMU_MAX_BUFFERS memory reserved for conversion tasks. Default: Not set; internal distribution rules apply.

File element	Description
TMU_OUTPUT_BUFFER_PERCENT	Specifies the percentage of the TMU_MAX_BUFFERS memory reserved for output tasks. Default: Not set; internal distribution rules apply.
TMU_INDEX_BUFFER_PERCENT	Specifies the percentage of memory reserved for index tasks. Default: Not set; internal distribution rules apply.
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.) Range of values: integer from 1 to 2^{31}
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.) Range of values: integer from 1 to 2^{31}
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.) Range of values: integer from 1 to 2^{31}
TUNE ROWS_PER_TARGETJOIN_TASK	Specifies the minimum estimated number of rows the TARGETjoin processes before parallel TARGETjoin processes are used. Default: 2,147,483,647 (Do not enter commas.) Range of values: integer from 1 to 2^{31}
TUNE QUERYPROCS	Specifies upper limit on the number of processes used to process a single query. Default: 0 Range of values: integer from 0 to 32,767

File element	Description
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 Range of values: integer from 0 to 32,767
TUNE FORCE_SCAN_TASKS	Specifies the number of parallel processes to use in a relation scan. Default: OFF Range of values: integer from 0 to 32,767
TUNE FORCE_FETCH_TASKS	Specifies the number of parallel processes to use fetching row data for a single query. Default: OFF Range of values: integer from 0 to 32,767
TUNE FORCE_JOIN_TASKS	Specifies the number of parallel processes to use processing joins for a single query. Default: OFF Range of values: integer from 0 to 32,767
TUNE FORCE_TARGETJOIN_TASKS	Specifies the number of parallel processes to use processing TARGETjoins for a single query. Default: OFF Range of values: integer from 0 to 32,767
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 FILE_GROUP	Defines disk groups for purposes of reducing disk seek contention. Default: none Range of values: integer from 0 to 32,767
TUNE FORCE_HASHJOIN_TASKS	Specifies the number of parallel processes to use processing hybrid hash joins. Default: OFF Range of values: integer from 0 to 32,767

File element	Description
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 Range of values: integer from 0 to 32,767
TUNE PARTITIONED_PARALLEL_AGGREGATION	Specifies if parallel processes are used to process aggregation functions. Default: OFF
TUNE PARALLEL_SET_OPERATION	Specifies if parallel processes are used to process queries that contain the set operators UNION, INTERSECT, and EXCEPT. 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
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 OLAP_APPROXIMATE_NUMERIC_FAST_COMPUTATION	When set to ON, improves performance of queries that contain OLAP functions. For more information on this parameter, see the *SQL Reference Guide*. Default: ON

File element	Description
OPTION PERFORMANCE_MONITOR_COMMANDS_LIMIT	Specifies the maximum number of commands that the performance monitor can store in the performance DSTs. Value must be greater than 0. For more information on this parameter, see the *Query Performance Guide*. Default: 100
OPTION PERFORMANCE_MONITOR_MAXOPERATORS	Specifies the maximum number of operators that a monitored query can contain. For more information on this parameter, see the *Query Performance Guide*. Default: 100
OPTION PERFORMANCE_MONITOR_SESSIONS	Specifies the maximum number of concurrent sessions that the performance monitor can monitor. Value must be greater than 0. For more information on this parameter, see the *Query Performance Guide*. Default: 20
OPTION TMU_OPTIMIZE	Turns optimized index-building on or off for the TMU. Default: OFF
OPTION XML_TEMP_SPACE	Specifies the directory in which spill files during parallel parsing of XML files are created. For more information on this parameter, see the *Table Management Utility Reference Guide*. Default: directory specified in QUERY_TEMPSPACE_DIRECTORY
OPTION NESTED_XML_INPUT	Enables or disables nested XML input semantics. Default: On
TUNE NESTED_XML_MEMORY_LIMIT	Specifies the maximum amount of memory that can be allocated to each XML parsing task when nested XML data is being loaded. Range of values: 64 kilobytes to 256 megabytes Default: 1 megabyte

File element	Description
TUNE XML_MAX_TASKS	Specifies maximum number of parallel tasks to allocate for parsing of XML input files. For more information on this parameter, see the *Table Management Utility Reference Guide*. Default: 1
TUNE INDEX_TEMPSPACE_THRESHOLD	Specifies the file size (in kilobytes or megabytes) at which spill files for index building are created. Default size: 10 megabyte Default units: kilobytes Maximum value: 2 gigabytes
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
TUNE QUERY_TEMPSPACE_MAXSPILLSIZE	Specifies maximum size in kilobytes (K), megabytes (M), or gigabytes (G) to which a file for query processing can grow. Default: 1 gigabyte
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

File element	Description
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)
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

File element	Description
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
TUNE 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`
TUNE QUERY_MMAP_LIMIT	Specifies the maximum amount of process address space, or program memory, that can be used to memory-map dimension tables and indexes during query execution. Default: 5M
TUNE BAR_UNIT_SIZE	Specifies the maximum size of a backup file, in kilobytes (K), megabytes (M), or gigabytes (G). For more information on this parameter, see the *Table Management Utility Reference Guide*. Default: `256M`

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 path name 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 path name 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 (DROP) or kept (KEEP) when the table or index associated with them is dropped. (Default segments are always dropped.) Default: KEEP

File element	Description
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
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.

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>/logson UNIX or <%RB_CONFIG%>\logs on Windows

File element	Description
ADMIN ACCT_MAXSIZE	Sets maximum accounting file size. The minimum value is 10,240 kilobytes. Default: limited only by available disk space (Do not enter commas.)
ADMIN ACCT_LEVEL	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
ADMIN LOG_USAGE_LEVEL	Sets the minimum severity level that is logged for usage events. Default: ALERT

File element	Description
ADMIN REPORT_INTERVAL	Sets the maximum interval (in minutes) between dynamic system table refreshes. Default: 1 minute
ADMIN RENICE_COMMAND (UNIX)	Specifies the full path name of a UNIX **renice** executable file that changes user priorities. The full path name includes the directory, but not the name of the executable. For example, if the command resides in `/usr/ucb/renice`, specify `/usr/ucb` in this parameter. 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.)

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

File element	Description
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
PASSWORD ALLOW_LOGNAME	Specifies whether the password can include the user's login name. Default: ON
PASSWORD CHANGE_INITIAL	Specifies whether the user must change the initial password when the user logs in for the first time. Default: OFF
PASSWORD COMPLEX_NUMERIC	Specifies whether the changed password must differ from previous passwords by more than the value of a single numeric. Default: OFF

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)	x			x	x	rbwapid
RB_HOST MAPFILE (UNIX)	x			x	x	rbwapid
RB_HOST SERVER	x					rbwapid
RBWAPI parameters						
MAX_SERVERS	x					rbwapid
CLEANUP_SCRIPT	x					rbwapid
UNIFIED_LOGON (Windows)	x					rbwapid
STACK_SIZE (Windows)	x					rbwapid
SERVER_NAME (UNIX)	x					rbwapid
LOGFILE_SIZE	x					rbwapid
PROCESS_CHECKING _INTERVAL	x					rbwapid
MAX_ACTIVE_ DATABASES	x					rbwapid
REMOTE_TMU_ LISTENER	x					rbwapid

Parameters	Set with			Affects		
	rbw.config	SQL SET	TMU command	Server	TMU	Daemon
NLS_LOCALE parameters						
MESSAGE_DIR	x			x	x	all
LOCALE	x			x	x	all
FIRST_DAYOFWEEK	x			x	x	all
RBMON parameter						
RBMON INTERVAL (UNIX)	x					servermon
TUNE parameters						
TMU_BUFFERS	x		x		x	
ROWS_PER_SCAN_ TASK	x	x		x		
ROWS_PER_FETCH_ TASK	x	x		x		
ROWS_PER_JOIN_ TASK	x	x		x		
ROWS_PER_ TARGETJOIN_ TASK	x					
QUERYPROCS	x	x		x		
TOTALQUERYPROCS	x			x		rbwapid
FORCE_SCAN_TASK	x	x		x		
FORCE_FETCH_TASK	x	x		x		
FORCE_JOIN_TASK	x	x		x		
GROUP	x			x		rbwapid
FILE_GROUP	x			x		rbwapid
FORCE_HASHJOIN_ TASK	x	x		x		
PARALLEL_ HASHJOIN	x	x		x		
FORCE_ AGGREGATION_ TASKS	x	x		x		
PARTITIONED_ PARALLEL_ AGGREGATION	x	x		x		
PARALLEL_SET_ OPERATION	x					
RESULT_BUFFER	x	x		x		
RESULT_BUFFER_ FULL_ACTION	x	x		x		
FILLFACTOR SI, PI, STAR	x			x	x	

Parameters	Set with			Affects		
	rbw.config	SQL SET	TMU command	Server	TMU	Daemon
FILLFACTOR VARCHAR	x			x	x	
INDEX_TEMPSPACE_ THRESHOLD, MAXSPILLSIZE, DIRECTORY	x	x		x		
QUERY_MEMORY_ LIMIT, QUERY_TEMPSPACE_ MAXSPILLSIZE, DIRECTORY	x	x		x		
TMU_ROW_ MESSAGES	x		x		x	
TMU_SERIAL_MODE	x		x		x	
TMU_CONVERSION _ TASKS	x		x		x	
TMU_INDEX_TASKS	x		x		x	
TMU_INPUT_TASKS	x		x		x	
TMU_MAX_TASKS	x		x		x	
TMU_MMAP_LIMIT	x		x		x	
SAMPLE_ MARGIN	x	x		x		
ADVISOR_SAMPLE_ SIZE	x	x		x		
XML_MAX_TASKS	x		x		x	
NESTED_XML_ MEMORY_LIMIT	x		x		x	
QUERY_MMAP_ LIMIT	x	x		x		
BAR_UNIT_SIZE	x	x	x		x	
OPTION parameters						
AUTOROWGEN	x		x		x	
ARITHABORT	x	x		x		
ALLOW_POSSIBLE_ DEADLOCK	x	x		x		
READER_PRIORITY	x			x		
CROSS_JOIN	x	x		x		
COUNT_RESULT	x	x		x		
CHECK_REPORT_ FILE_PERMISSIONS	x			x		

Parameters	Set with			Affects		
	rbw.config	SQL SET	TMU command	Server	TMU	Daemon
CHECK_TABLE_ INDEX_ DIRECTORY	x	x		x		
GRANT_TEMP_ RESOURCE_TO_ALL	x			x		
ADVISOR_LOGGING	x	x		x		
UNIFORM_ PROBABILITY_ FOR_ ADVISOR	x	x		x		
PRECOMPUTED_ VIEW_QUERY_ REWRITE	x	x		x		
USE_INVALID_ PRECOMPUTED_ VIEWS	x	x		x		
PREPLANS_HAVE_ ONLY_PKFK_JOINS	x	x		x		
ADVISOR_ MAXIMUM_ CANDIDATE_VIEWS	x	x		x		
VERSIONING	x	x		x	x	
TRANSACTION_ ISOLATION_LEVEL	x	x		x	x	
TMU_VERSIONING	x					
TMU_COMMIT_ RECORD_INTERVAL	x		x		x	
TMU_COMMIT_ TIME_ INTERVAL	x		x		x	
EXPORT_DEFAULT_ PATH	x	x		x		
EXPORT_DELIMITER	x			x		
EXPORT_MAX_FILE_ SIZE	x	x		x		
IDLE_TIMEOUT	x			x		
PRECOMPUTED_ VIEW_ MAINTENANCE	x	x		x	x	
PRECOMPUTED_ VIEW_ MAINTENANCE_ ON_ERROR	x	x		x	x	
TMU_OPTIMIZE	x				x	

Parameters	Set with			Affects		
	rbw.config	SQL SET	TMU command	Server	TMU	Daemon
TARGETJOIN_ LOCAL_ PREDICATES	x	x		x		
ALLOW_SINGLE_ COLUMN_BTREE_ IN_TARGETJOIN	x	x		x		
OLAP_ APPROXIMATE_ NUMERIC_FAST_ COMPUTATION	x	x		x		
XML_TEMP_SPACE	x		x		x	
NESTED_XML_INPUT	x		x		x	
QUERY_MMAP	x	x		x		
BAR_XBSA_LIB	x				x	
BAR_SM_USER	x				x	
PERFORMANCE_ MONITOR_ COMMANDS	x			x		rbwpmond
PERFORMANCE_ MONITOR_ MAXOPERATORS	x			x		rbwpmond
PERFORMANCE_ MONITOR_ MAXSESSIONS	x			x		rbwpmond
SEGMENT parameters						
DEFAULT_DATA_ SEGMENT	x	x		x		
DEFAULT_INDEX_ SEGMENT	x	x		x		
TEMPORARY_DATA_ SEGMENT	x	x		x		
TEMPORARY_INDEX_ SEGMENT	x	x		x		
SEGMENTS (DROP, KEEP)	x	x		x		
IGNORE_PARTIAL_ INDEXES	x	x		x		
PARTIAL_ AVAILABILITY	x	x		x		
IGNORE_OPTICAL_ INDEXES	x	x		x		
OPTICAL_ AVAILABILITY	x	x		x		
DEFAULT_ SEGMENT_SIZE	x	x		x		
DEFAULT_PSU_ EXTENDSIZE	x	x		x		

Parameters	Set with			Affects		
	rbw.config	SQL SET	TMU command	Server	TMU	Daemon
DEFAULT parameters						
ROWCOUNT	x	x		x		
INFO_MESSAGE_ LIMIT	x			x		
RBW_LOADINFO_ LIMIT	x				x	
ADMIN parameters						
* On UNIX, you can set these parameters with an SQL ALTER SYSTEM statement.						
ACCOUNTING *	x					rbwlogd
ACCT_DIRECTORY	x					rbwlogd
ACCT_MAXSIZE *	x					rbwlogd
ACCT_LEVEL*	x					rbwlogd
LOGGING*	x					rbwlogd
LOG_AUDIT_LEVEL *	x					rbwlogd
LOG_DIRECTORY	x					rbwlogd
LOG_ERROR_LEVEL *	x					rbwlogd
LOG_MAXSIZE *	x					rbwlogd
LOG_OPERATIONAL_ LEVEL *	x					rbwlogd
LOG_SCHEMA_ LEVEL *	x					rbwlogd
LOG_USAGE_LEVEL*	x					rbwlogd
REPORT_INTERVAL	x	x		x	x	
RENICE_COMMAND (UNIX)	x			x		
ADVISOR_ LOGGING *	x			x		rbwlogd
ADVISOR_LOG_ DIRECTORY	x			x		rbwlogd
ADVISOR_LOG_ MAXSIZE	x			x		rbwlogd
PASSWORD parameters						
EXPIRATION_DAYS	x			x	x	
EXPIRATION_ WARNING_ DAYS	x			x	x	
CHANGE_MINIMUM_ DAYS	x			x		
RESTRICT_PREVIOUS	x					
COMPLEX_NUM_ ALPHA	x			x		

Parameters	Set with			Affects		
	rbw.config	SQL SET	TMU command	Server	TMU	Daemon
COMPLEX_NUM_ NUMERICS	x			x	x	
COMPLEX_NUM_ PUNCTUATION	x			x	x	
MINIMUM_LENGTH	x			x	x	
LOCK_FAILED_ ATTEMPTS	x			x	x	
LOCK_PERIOD_ HOURS	x			x	x	
ALLOW_LOGNAME	x			x		
CHANGE_INITIAL	x			x		
COMPLEX_NUMERIC	x			x		
Other parameters						
DB logical database names	x			x	x	all
LOCK (WAIT, NO WAIT)		x		x		
			x		x	
LOCK (table, database)		x		x	x	
ORDER_BY_ASC_ NULL, ORDER_BY_ DESC_NULL		x		x		

* On UNIX, you can set these parameters with an SQL ALTER SYSTEM statement.

Appendix C. System tables and dynamic statistic tables

This appendix describes the system catalog, the system tables, and the dynamic statistic tables (DSTs) for IBM Red Brick Warehouse. 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

IBM Red Brick Warehouse 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.

© Copyright IBM Corp. 1996, 2004

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_OPTIMIZER_OPTIONS	Lists the current values of any optimizer options you have set for tables in the database. This system table also stores default STARJOIN and TARGETJOIN THRESHOLD values if you have not changed them with SET commands. For information about specific optimizer options, see the *Query Performance Guide* and the *SQL Reference Guide*.
RBW_OPTIONS	Displays current settings for database parameters that can be tuned.
RBW_PRECOMPVIEW_CANDIDATES	Advisor system table for candidate views (See the *IBM Red Brick Vista User's Guide*.)
RBW_PRECOMPVIEW_UTILIZATION	Advisor system table for views that are defined in the database (Refer to the *IBM Red Brick 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.
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_VIEW_REFERENCES	Describes the tables that a view references.
RBW_VIEWTEXT	Describes the text of all views in RBW_VIEWS.

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.

Column name	Column type	Column description
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	VARCHAR(1022)	Default for column.
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.

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.
LOCAL	CHAR(1)	Flag indicating whether index is a local index where each index segment points to row data in a corresponding table segment. (Y or N).
CNAME	CHAR(128)	Indexed column name; first column name on multicolumn 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.

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.

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.
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.

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.
CATEGORY	SMALLINT	Syntax category for macro. Values less than 256 can be defined by IBM Red Brick Warehouse; NULL if not set with the CREATE MACRO statement. For information about categories that IBM Red Brick Warehouse 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.

RBW_OPTIMIZER_OPTIONS table

The RBW_OPTIMIZER_OPTIONS table lists the current values of any optimizer options you have set for tables in the database. This system table also stores default STARJOIN and TARGETJOIN THRESHOLD values if you have not changed them with SET commands. For information about specific optimizer options, see the *Query Performance Guide* and the *SQL Reference Guide*.

Column name	Column type	Column description
TNAME	CHAR(128)	Name of the target table for the optimization. NULL indicates either a default value or a value that applies to all tables.
OPTIMIZATION_NAME	CHAR(128)	Name of the optimizer option.
VALUE	CHAR(1024)	Current setting of the option for the specified table.
SEQ	INTEGER	Sequence number of the optimization text, for settings that do not fit into one row of the table.

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, see the *IBM Red Brick Vista User's Guide*.

RBW_PRECOMPVIEW_UTILIZATION table

For a description of the RBW_PRECOMPVIEW_UTILIZATION Advisor system table, see the *IBM Red Brick 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.
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 _FKEYn_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.

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.
NCOLS	INTEGER	Number of columns used to segment the data or index.
MINKEY	VARCHAR(1024)	Minimum key value in the segment. For a STAR index, this key value is of the form *segname rownum* where *segname* identifies a segment attached to the referenced table and *rownum* identifies a row within that segment.
MAXKEY	VARCHAR(1024)	Maximum key value in the segment. For a STAR index, this key value is of the form *segname rownum* where *segname* identifies a segment attached to the referenced table and *rownum* identifies a row within that segment.
ID	INTEGER	Segment ID.
TOTALFREE	INTEGER	Kilobytes of unused space in segment. Refer to “TOTALFREE column” on page 278.

Column name	Column type	Column description
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.
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. Possible values: • RBW_SEGID of table segment • LOCAL_ID of corresponding data segment if segment is part of a local index • NULL for STAR index segments, non-local index segments or unattached segments
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.
DEFAULT_SEG	CHAR(1)	Flag indicating whether segment is a default segment (Y), or a named segment (N).

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 276, "How space is allocated to a segment" on page 281, and "Changing PSU sizes" on page 293.
INITSIZE	INTEGER	Kilobytes of initial allocated size of PSU. Refer to "Changing PSU sizes" on page 293..
EXTENDSIZE	INTEGER	Kilobytes of increment size to use when extending PSU. Refer to "Changing PSU location" on page 293.
USED	INTEGER	Kilobytes of current area in use in PSU. Refer to "USED column" on page 277.
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.

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).
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.
ALTER_PENDING	CHAR (1)	Indicates whether an ADD or DROP alter operation is in progress on table columns or constraints (Y or N).

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 418.

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.
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).
DROP_ANY	CHAR(1)	Whether GRANTEE can drop objects created by any user (Y, N, R, or I).

Column name	Column type	Column description
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).
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).

Column name	Column type	Column description
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.

RBW_VIEW_REFERENCES

The RBW_VIEW_REFERENCES table contains for each view, the names of all the tables in the view query. When users display information stored in the RBW_VIEW_REFERENCES 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
VIEW_NAME	CHAR(128)	View table name.
REF_NAME	CHAR(128)	Name of the table that the view references. If the view is a precomputed view, the name of the aggregate table associated with the view is not listed here.
REF_TYPE	CHAR(8)	Type of referenced table: TABLE, VIEW, SYNONYM, or SYSTEM.

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 227.

The Query Performance Monitor stores statistics for queries and other SQL commands in the following performance DSTs:

- DST_PERFORMANCE_COMMANDS
- DST_PERFORMANCE_OPSTATS
- DST_PERFORMANCE_IOSTATS

For more information on the performance DSTs, refer to the *Query Performance Guide*.

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)	Host name where the session is running.
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.

Column name	Column type	Description
CACHE_READS	INTEGER	Number of times that a block was found in local buffer cache (avoiding a logical read request). This number is increased each time the session performs a read or write operation because the database server must read the data on the block in the cache before it can do the write.
CACHE_WRITES	INTEGER	Number of times that a block was found in local buffer cache (avoiding a logical write request).
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.
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.

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.
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.

Column name	Column type	Description
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.
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.

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)	Host name where the session is running.
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.
NODE_NAME	CHAR(128)	Host name where the session is running.
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

Column name	Column type	Description
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.
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).

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_LOGICAL_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).
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.

Column name	Column type	Description
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.

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.

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_LOGICAL_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).
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).

Column name	Column type	Description
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.

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)
DECIMAL or NUMERIC with precision:	
1-2 3-4 5-9 10-11 12-14 15-16 17-18 19-21 22-23 24-26 27-28 29-31 32-33 34-35 36-38	1 2 4 5 6 7 8 9 10 11 12 13 14 15 16

For definitions and use of data types, refer to the *SQL Reference Guide*.

Appendix D. Configuring federated access to Red Brick data sources

This document explains how to set up and query an IBM Red Brick Warehouse database as a federated data source by using IBM DB2 with the Information Integrator ODBC wrapper.

- The first section explains the setup required on Windows platforms.
- The second section explains the setup required on UNIX and Linux platforms.

The procedures that follow assume that you already have the following products installed:

- IBM DB2 UDB Version 8.1.4 or later.
- IBM DB2 Information Integrator Version 8.1.4 or later. Your Information Integrator installation must include the relational ODBC wrapper.
- IBM Red Brick Warehouse Version 6.20.xC5 or later, including the appropriate ODBC client programs for your platform.

  Note: Users on Solaris 64-bit platforms should install the Red Brick client programs from the `sol` (32-bit) directory, not the `sol64` directory.

Federated data access on Windows platforms

IBM DB2 and Information Integrator support Red Brick data federation on the following Windows operating systems:

- Windows 2000
- Windows NT

To configure federated access to a Red Brick data source:

1. Start the IBM DB2 Command Center by selecting Start, then Programs, then IBM DB2, then Command Line Tools, then Command Center.
2. Select the Interactive tab, then enter and execute the following command:

   ```
   update dbm cfg using federated yes
   ```

   This command enables the federation feature for all databases created with the current installation of DB2. You should see the following response:

   ```
   DB20000I The UPDATE DATABASE MANAGER CONFIGURATION command
   completed successfully.
   ```

   You might also see the following message:

© Copyright IBM Corp. 1996, 2004

```
SQL1362W One or more of the parameters submitted for immediate
modification were not changed dynamically. Client changes will not be
effective until the next time the application is started or the TERMINATE
command has been issued. Server changes will not be effective until the
next DB2START command.
```

If you see this message, stop (**db2stop**) and restart (**db2start**) the DB2 service to make the configuration change take effect.

3. Create and connect to a new DB2 database (or connect to an existing database). For example:

   ```
   create database db2brick
   connect to db2brick
   ```

   where *db2brick* is the name of the new database or:

   ```
   connect to sample
   ```

   where *sample* is the existing sample database installed with DB2.

   The CREATE DATABASE command takes a few minutes to complete.

4. Create an ODBC wrapper for Red Brick:

   ```
   create wrapper brick_wrap library 'db2rcodbc.dll'
   options(db2_fenced 'n')
   ```

   where *brick_wrap* is the name of the wrapper.

5. Create a server for Red Brick that references the wrapper you just created:

   ```
   create server brick_svr type odbc version 3.0 wrapper brick_wrap
   options(node 'aroma_local', password 'y')
   ```

   where *brick_wrap* is the wrapper you created in step 4 and *aroma_local* is the name of a valid *system* DSN (data source name). This DSN must point to the Red Brick database, host machine, and port number that you want to connect to from DB2. (If your DSN was created as a *user* DSN, create a *system* DSN copy of the user DSN before proceeding.)

6. Create a user mapping, which maps your DB2 username to the Red Brick database username:

   ```
   create user mapping for bobr server brick_svr
   options(remote_authid 'SYSTEM', remote_password 'MANAGER')
   ```

   where *bobr* is the DB2 username, *brick_svr* is the server created in step 5, and the *remote_authid* and *remote_password* strings match the username and password for logging into the Red Brick database.

7. Open a Command window and run the db2rbsetup.exe program. Respond to the prompts as shown below:

```
Usage: db2rbsetup options
    Options:
        -db2db <DB2 federated database name to connect to>
        -db2user <DB2 user name to logon to DB2 database>
        -db2pwd <DB2 user password to logon to DB2 database>
        -server <DB2 federated server name for Red Brick>
DB2 federated database name to connect to-> sample
DB2 user name to logon to DB2 database-> bobr
DB2 user password to logon to DB2 database->
DB2 federated server name for Red Brick-> brick_svr
Setting up customization options for Red Brick. Please wait...
Red Brick customization options have been setup successfully.
```

   When you press Return after entering the Red Brick server name, the setup program defines the customization options for Red Brick query processing. These options are critical for optimizing query performance when Red Brick tables are accessed from DB2. If you prefer, you can enter the options directly on the command line instead of using the program interactively.

   The process of defining the options could take some time to complete. To verify that the db2rbsetup.exe program was successful, run the following query:

```
select count(*)
from syscat.serveroptions
where servername = 'BRICK_SVR'
```

   where *BRICK_SVR* is the name of the server created in step 5. This query should return a value of 128 or greater.

   Note: The db2rbsetup.exe program can only be executed once for each server (as defined in step 5).

8. Make sure the Red Brick database defined in the server you created in step 5 is up and running.
9. Create nicknames for all of the Red Brick tables you intend to query from DB2. For example, the following commands create nicknames for the **Sales** and **Period** tables in the **Aroma** database:

```
create nickname aroma_sales for brick_svr.sales
create nickname aroma_period for brick_svr.period
```

   You are now ready to run federated queries that involve those tables, using the nicknames you just defined. For example:

```
select * from aroma_sales, aroma_period
where aroma_sales.perkey = aroma_period.perkey
and aroma_sales.dollars > 600
```

Federated data access on UNIX and Linux platforms

IBM DB2 and Information Integrator support Red Brick data federation on the following operating systems:

- AIX
- HP-UX
- Solaris
- Linux

Before you can configure federated access to Red Brick data sources on UNIX and Linux platforms, you must set up the DB2 environment.

To set up the DB2 environment:

1. Create or update the `db2dj.ini` file in your `$DB2_INSTANCE_HOME/sqllib/cfg` directory with the following information:
   - ODBCINI=*odbcini_path*

     where *odbcini_path* is the full pathname of your `.odbc.ini` file.
   - RB_CONFIG=*config_path*

     where *config_path* is the location of the directory that contains the Red Brick Warehouse configuration file (`rbw.config`).
   - LD_LIBRARY_PATH=*datadirectlib_path*

     where *datadirectlib_path* is the location of the `lib` directory for DataDirect Technologies Connect ODBC.

   For example, your `db2dj.ini` file might look like this:

   ```
   ODBCINI=/home/bobr/.odbc.ini
   RB_CONFIG=/rbw
   LD_LIBRARY_PATH=/rbw/lib
   ```

   Note: On AIX platforms, use LIBPATH instead of LD_LIBRARY_PATH, and on HP-UX 10.20 platforms, use SHLIB_PATH instead of LD_LIBRARY_PATH.
2. Set the LD_LIBRARY_PATH environment variable (or LIBPATH on AIX and SHLIB_PATH on HP-UX 10.20):

   ```
   export LD_LIBRARY_PATH=datadirectlib_path:db2lib_path
   ```

   where *datadirectlib_path* is the same value you entered in step 1 and *db2lib_path* is the location of your DB2 instance library. For example:

   ```
   LD_LIBRARY_PATH=/rbw/lib:/sqllib/lib
   ```
3. Make sure the DB2 instance is not running, then set the DataDirect Technologies Connect ODBC library path:

   ```
   db2stop
   db2set db2libpath=datadirectlib_path
   ```

where *datadirectlib_path* is the same path you entered in steps 1 and 2.

Additional setup required on AIX, HP-UX, and Linux platforms
The following additional environment setup is not required on Solaris platforms.

On AIX platforms, complete the following steps as the root user:

1. Change to the location of the `lib` directory for DataDirect Technologies Connect ODBC:

   ```
   % cd /rbw/lib
   ```

2. Create the following symbolic link:

   ```
   % ln -s libodbc.a odbc.so
   ```

On HP-UX platforms, complete the following steps:

1. Create an `exe` directory under `/usr` and change to the new directory:

   ```
   % mkdir /usr/exe
   % cd /usr/exe
   ```

2. Create the following symbolic link:

   ```
   % ln -s /rbw/lib/libodbcinst.sl libodbcinst.sl
   ```

On Linux platforms, complete the following steps:

1. Change to the `/usr/lib` directory:

   ```
   % cd /usr/lib
   ```

2. Create the following symbolic link:

   ```
   % ln -s /rbw/lib/libodbcinst.sl libodbcinst.sl
   ```

To configure federated access to a Red Brick data source:

1. Start the IBM DB2 Command Line Processor (CLP).
2. Execute the following command:

   ```
   db2 => update dbm cfg using federated yes
   ```

   This command enables the federation feature for all databases created with the current installation of DB2. You should see the following response:

   ```
   DB20000I The UPDATE DATABASE MANAGER CONFIGURATION command
   completed successfully.
   ```

   You might also see the following message:

   ```
   SQL1362W One or more of the parameters submitted for immediate
   modification were not changed dynamically. Client changes will not be
   effective until the next time the application is started or the TERMINATE
   command has been issued. Server changes will not be effective until the
   next DB2START command.
   ```

If you see this message, stop (**db2stop**) and restart (**db2start**) the DB2 server to make the configuration change take effect.

3. Create and connect to a new DB2 database (or connect to an existing database). For example:

```
db2 => create database db2brick
db2 => connect to db2brick
```

where *db2brick* is the name of the new database or:

```
db2 => connect to sample
```

where *sample* is the existing sample database installed with DB2.

The CREATE DATABASE command takes a few minutes to complete.

4. Create an ODBC wrapper for Red Brick. The following example is specific to Solaris and Linux platforms:

```
db2 => create wrapper brick_wrap library 'libdb2rcodbc.so'
options(db2_fenced 'n', module '/rbw/lib/libodbc.so')
```

where *brick_wrap* is the name of the wrapper and the module path specifies the location of the Red Brick *libodbc.so* library.

On HP-UX platforms, use the following command:

```
db2 => create wrapper brick_wrap library 'libdb2rcodbc.sl'
options(db2_fenced 'n', module '/rbw/lib/libodbc.sl')
```

On AIX platforms, use the following command:

```
db2 => create wrapper brick_wrap library 'libdb2rcodbc.a'
options(db2_fenced 'n', module '/rbw/lib/libodbc.a(odbc.so)')
```

5. Create a server for Red Brick that references the wrapper you just created:

```
db2 => create server brick_svr type odbc version 3.0
wrapper brick_wrap options(node 'aroma_local', password 'y')
```

where *brick_wrap* is the wrapper you created in step 4 and *aroma_local* is the name of a valid data source that points to the Red Brick database, host machine, and port number that you want to connect to from DB2.

6. Create a user mapping, which maps your DB2 username to the Red Brick database username:

```
db2 => create user mapping for bobr server brick_svr
options(remote_authid 'SYSTEM', remote_password 'MANAGER')
```

where *bobr* is the DB2 username, *brick_svr* is the server created in step 5, and the *remote_authid* and *remote_password* strings match the username and password for logging into the Red Brick database.

7. Run the `db2rbsetup` program. Respond to the prompts as shown below:

```
db2 => db2rbsetup
Usage: db2rbsetup options
    Options:
        -db2db <DB2 federated database name to connect to>
        -db2user <DB2 user name to logon to DB2 database>
        -db2pwd <DB2 user password to logon to DB2 database>
        -server <DB2 federated server name for Red Brick>
DB2 federated database name to connect to-> sample
DB2 user name to logon to DB2 database-> bobr
DB2 user password to logon to DB2 database->
DB2 federated server name for Red Brick-> brick_svr
Setting up customization options for Red Brick. Please wait...
Red Brick customization options have been setup successfully.
```

When you press Return after entering the Red Brick server name, the setup program defines the customization options for Red Brick query processing. These options are critical for optimizing query performance when Red Brick tables are accessed from DB2.

If you prefer, you can enter the options directly on the command line instead of using the program interactively. The process of defining the options could take some time to complete. To verify that the `db2rbsetup` program was successful, run the following query:

```
db2 => select count(*) from syscat.serveroptions
where servername = 'BRICK_SVR'
```

This query should return a value of 128 or greater.

Note: The `db2rbsetup` program can only be executed once for each server (as defined in step 5).

8. Make sure the Red Brick database defined in the server you created in step 5 is up and running.
9. Create nicknames for all of the Red Brick tables you intend to query from DB2. For example, the following commands create nicknames for the **Sales** and **Period** tables in the **Aroma** database:

```
db2 => create nickname aroma_sales for brick_svr.sales
db2 => create nickname aroma_period for brick_svr.period
```

You are now ready to run federated queries that involve those tables, using the nicknames you just defined. For example:

```
db2 => select * from aroma_sales, aroma_period
where aroma_sales.perkey = aroma_period.perkey
and aroma_sales.dollars > 600
```

Example of an .odbc.ini file

The following text shows the typical contents of an `.odbc.ini` file. The name of the DRIVER file varies across platforms.

```
[ODBC]
InstallDir=/home/bobr/rbw

#This section is to name your ODBC DSNs
#One entry per DSN
#
[ODBC Data Sources]
aroma_local=Red Brick Driver

#This section is to define your ODBC DSNs
#One set of entries per DSN
#
[aroma_local]
DRIVER=/rbw/lib/rbodbcdrv.so
SERVER=windsor:6205
RB_CONFIG=/rbw
DATABASE=AROMA
UID=system
PWD=manager
```

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make

© Copyright IBM Corp. 1996, 2004

improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J35/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© IBM 2003. Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1996-2003. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

AIX	IBM Informix SE
C-ISAM	IBM Informix SQL
Client SDK	i.Financial Services
Cloudscape	InformiXML
Cloudsync	J/Foundation
DB2	MaxConnect
DB2 Universal Database	NUMA-Q
Distributed Relational Database Architecture	Object Translator
Dynamic Connect	OS/2
Foundation.2000	OS/390
IBM	OS/400
IBM Informix	RedBack
IBM Informix 4GL	Red Brick
IBM Informix Connect	SystemBuilder
IBM Informix Data Blade Module	U2
IBM Informix Driver for JDBC	UniData
IBM Informix Dynamic Scalable Architecture(DSA)	UniVerse
IBM Informix Dynamic Server	Wintegrate
IBM Informix Enterprise Gateway Manager (Enterprise Gateway Manager)	
IBM Informix Extended Parallel Server	

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Windows, Windows NT, and Excel are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names used in this publication may be trademarks or service marks of others.

Index

Special characters

.rbretrc file 12, 32
.rbwerr file 32
.rbwrc file 12, 31

A

Access authorizations 44, 195, 223
ACCESS_ADVISOR_INFO task authorization 202
ACCESS_ANY task authorization 202
ACCESS_SYSINFO task authorization 202, 226
Accounting
  changing modes 254
  configuring 251
  files 250
  job mode 248
  LA parameters 249
  overview 248
  process 249
  rbwlogd daemon 249
  record format 249
  sample code for 250
  setting mode 253
  starting 253
  startup state 251
  stopping 253
  switching files 254
  workload mode 249
Accounting file
  location 251, 252
  removing 251
  size 250, 252
  switching 254
ACCOUNTING parameter
  rbw.config file entry 393
  usage 251
ACCT_DIRECTORY parameter
  rbw.config file entry 393
  usage 251, 252
ACCT_LEVEL parameter
  rbw.config file entry 394
  usage 253
ACCT_MAXSIZE parameter
  rbw.config file entry 394
  usage 250, 252
ADD STORAGE clause, ALTER SEGMENT statement 284
ADMIN ADVISOR_LOGGING parameter
  modifying 323, 326
  rbw.config file entry 395
ADMIN ADVISOR_LOGGING parameter *(continued)*
  usage 246
ADMIN database
  described 226
  rbw.config file entry 397
ADMIN parameters
  entries in rbw.config 393
  modifying 326, 330
  summary of 402
Administration daemon
  output 43
Administration database 226
Administration of databases, overview 10
Administrator tool 5, 34
Advisor
  described 25
  log files
    specifying directory for 247
    specifying size of 247
  log files, removing 246
  logging 245
ADVISOR_LOG_DIRECTORY parameter
  rbw.config file entry 395
  usage 247
  use with log files 245
ADVISOR_LOG_MAXSIZE parameter
  rbw.config file entry 395
  usage 247
  use with log files 245
ADVISOR_LOGGING parameter
  modifying 325
ADVISOR_LOGGING, ADMIN parameter
  rbw.config file entry 395
ADVISOR_LOGGING, OPTION parameter
  rbw.config file entry 380
ADVISOR_MAXIMUM_CANDIDATE_VIEWS parameter 325, 329, 381
ADVISOR_SAMPLE_SIZE parameter
  modifying 325, 329
  rbw.config file entry 391
Aggregate maintenance 26
Aggregate tables
  described 25
  example 75
ALLOW_LOGNAME parameter
  rbw.config file entry 396
  usage 219
ALLOW_POSSIBLE_DEADLOCK parameter
  modifying 324, 328

© Copyright IBM Corp. 1996, 2004

ALLOW_POSSIBLE_DEADLOCK parameter *(continued)*
rbw.config file entry 379
syntax and usage 260
ALTER DATABASE statement
CLEAN VERSION LOG clause 188
FREEZE QUERY REVISION clause 184
START CLEANING clause 193
STOP CLEANING clause 193
UNFREEZE QUERY REVISION clause 190
ALTER SEGMENT statement
ADD STORAGE clause 284
ATTACH clause 283
CHANGE EXTENDSIZE clause 284
CHANGE MAXSIZE clause 284
CHANGE PATH clause 284, 293
CLEAR clause 284
DETACH clause 283
DROP LAST STORAGE clause 284
failure 284
FORCE INTACT clause 284, 306
MIGRATE TO clause 284, 293
OFFLINE clause 283
ONLINE clause 307
OVERRIDE FULLINDEXCHECK clause 296
RANGE clause 284
RANGE MOVE clause 284
RELEASE STORAGE clause 284
SEGMENT BY clause 284
usage 283
VERIFY clause
cause of damage 307
damaged segments 305
description 284
determining segment damage 294
ALTER SYSTEM statement
activating a database 231
Advisor logging 247
changing modes 254
clauses
ADVISOR_LOGGING 247
CANCEL USER COMMAND 231
CHANGE ACCOUNTING LEVEL 254
CHANGE LOGGING LEVEL 244
CHANGE USER PRIORITY 232
CLOSE USER SESSION 231
QUIESCE DATABASE 230
RESET STATISTICS 231
RESUME DATABASE 231
START ACCOUNTING 253
STOP ACCOUNTING 253
STOP LOGGING 244
SWITCH ACCOUNTING FILE 254
SWITCH LOGGING FILE 244
SWITCH_ADVISOR_LOG_FILE 247
TERMINATE LOGGING DAEMON 244
ALTER SYSTEM statement *(continued)*
closing a session 231
required authorizations 230
ALTER TABLE statement
adding column 310
adding foreign key 315
cascade deletes 314
changing default value 313
DROP CONSTRAINT clause 315
dropping column 310
fill factor 315
interrupted operations 316
location 311
Default segment 311
IN NAMED segment 311
IN PLACE 311
IN PLACE USING working_segment 311, 312
ON DELETE clause 314
out of space errors 317
renaming column 313
space requirements 311
start and step values for SERIAL columns 316
usage 310
ALTER USER statement 232
ALTER_ANY task authorization 202
ALTER_OWN task authorization 203
ALTER_SYSTEM task authorization 202, 226
ALTER_TABLE_INTO_ANY task authorization 203
ARITHABORT parameter
modifying 324, 328
rbw.config file entry 379
Arithmetic errors, options 379
Aroma database
building, tutorial 343, 363
installation 14
rbw.config file entry 397
ASCII code set 37
Associative tables 61
ATTACH clause, ALTER SEGMENT statement 283
Attaching segments 285
Attributes, influence on schema design 72
Audit events 240
Automatic Row Generation, TMU
and referential integrity 46
AUTOROWGEN parameter
modifying 325, 329
rbw.config file entry 379
Availability of tables and indexes 288

B

Backup operations
cleaning version log 188
during index creation 149
on versioned databases 191
overview 14

BAR_SM_USER parameter 323, 327, 329
BAR_UNIT_SIZE parameter
modifying 325
rbw.config file entry 391
BAR_XBSA_LIB parameter 323, 326, 329
BARTAPE parameter
modifying 325
rbw.config file entry 384
Binary character comparisons 37
Block, 8-kilobyte 276
Blocks of data, in version log 181
Boldface type xiv
Browsing dimension tables 71

C

CASCADE keyword, usage 141
Cascaded delete operations 47, 141
Cases tracked by Technical Support xix
Catalog
compaction 340
recovery 342
removing blocks 340
CATEGORY value, macro 171
Cautions
FORCE INTACT for segments 294
OVERRIDE REFCHECK 142
CHANGE ACCOUNTING LEVEL clause, ALTER SYSTEM statement 254
CHANGE EXTENDSIZE clause, ALTER SEGMENT statement 284
CHANGE MAXSIZE clause, ALTER SEGMENT statement 284
CHANGE PATH clause, ALTER SEGMENT statement 284, 293
CHANGE_INITIAL parameter
rbw.config file entry 396
usage 219
CHANGE_MINIMUM_DAYS parameter 395
Character data 37
CHECK INDEX
DESCRIBE option 267
file permissions 379
obtaining size information 261
VERBOSE option 265, 266, 268, 270, 272
CHECK TABLE
checking table structure 261, 264
file permissions 379
VARCHAR fill factor 146, 147
VERBOSE option 147, 262, 264
CHECK_REPORT_FILE_PERMISSIONS parameter
modifying file 324, 328
rbw.config file entry 379
CHECK_TABLE_INDEX_DIRECTORY parameter
modifying 324, 328
rbw.config file entry 380
CLEAN VERSION LOG clause, ALTER DATABASE statement 188
CLEANUP_SCRIPT parameter
modifying 326
modifying file 323
rbw.config file entry 378
syntax and usage 119
Cleanup, temporary space 119
CLEAR clause, ALTER SEGMENT statement 284
Clearing segments 289
Client tools
locales for 40
RISQL Entry Tool and RISQL Reporter 40
third-party 41
Client/server environment
compatibility 42
different locales in 39
Code set
ASCII 37
conversions 40, 42
defined 37
effect of changing 40
Collation sequence 37
Columns
adding 310
changing data type 314
default values, changing 313
dropping 310
estimating width 96
renaming 313
COMMENT value, macro 171
Compaction, system catalog 340
COMPLEX_NUM_ALPHA parameter
rbw.config file entry 396
usage 221
COMPLEX_NUM_NUMERICS parameter
rbw.config file entry 396
usage 221
COMPLEX_NUM_PUNCTUATION parameter
rbw.config file entry 396
usage 221
COMPLEX_NUMERIC parameter
rbw.config file entry 396
usage 219
Configuration file
described 365
entries in 377
examples 365
modifying 322
parameter summary 397
Configuration parameter
ACCOUNTING 251, 393
ACCT_DIRECTORY 251, 252, 393
ACCT_LEVEL 253, 394
ACCT_MAXSIZE 250, 252, 394

Configuration parameter *(continued)*
ADMIN ADVISOR_LOGGING 246
ADVISOR_LOG_DIRECTORY 245, 247, 395
ADVISOR_LOG_MAXSIZE 245, 247, 395
ADVISOR_SAMPLE_SIZE 391
ALLOW_LOGNAME 396
ALLOW_POSSIBLE_DEADLOCK 379
ARITHABORT 379
AUTOROWGEN 379
BAR_UNIT_SIZE 391
BARTAPE 384
CHANGE_INITIAL 396
CHANGE_MINIMUM_DAYS 395
CHECK_REPORT_FILE_PERMISSIONS 379
CHECK_TABLE_INDEX_DIRECTORY 380
CLEANUP_SCRIPT 378
COMPLEX_NUM_ALPHA 396
COMPLEX_NUM_NUMERICS 396
COMPLEX_NUM_PUNCTUATION 396
COMPLEX_NUMERIC 396
COUNT_RESULT 379
CROSS_JOIN 379
DEFAULT_DATA_SEGMENT 392
DEFAULT_INDEX_SEGMENT 392
DEFAULT_PSU_EXTENDSIZE 393
DEFAULT_SEGMENT_SIZE 393
EXPIRATION_DAYS 395
EXPIRATION_WARNING_DAYS 395
EXPORT_DEFAULT_PATH 382
EXPORT_DELIMITER 382
EXPORT_MAX_FILE_SIZE 382
FILE_GROUP 386
FILLFACTOR 387
FILLFACTOR_VARCHAR 387
FIRST_DAYOFWEEK 379
FORCE_AGGREGATION_TASKS 387
FORCE_FETCH_TASKS 386
FORCE_HASHJOIN_TASKS 386
FORCE_JOIN_TASKS 386
FORCE_SCAN_TASKS 386
FORCE_TARGETJOIN_TASKS 386
GRANT_TEMP_RESOURCE_TO_ALL 380
GROUP 386
IDLE_TIMEOUT 382
IGNORE_OPTICAL_INDEXES 309, 393
IGNORE_PARTIAL_INDEXES 393
INDEX_TEMPSPACE_DIRECTORY 110, 389
INDEX_TEMPSPACE_MAXSPILLSIZE 110, 389
INDEX_TEMPSPACE_THRESHOLD 110, 114, 115, 389
INFO_MESSAGE_LIMIT 392
LOCALE 379
LOCK_FAILED_ATTEMPTS 396
LOCK_PERIOD_HOURS 396
LOG_AUDIT_LEVEL 243, 394

Configuration parameter *(continued)*
LOG_DIRECTORY 242, 394
LOG_ERROR_LEVEL 243, 394
LOG_MAXSIZE 241, 243, 394
LOG_OPERATIONAL_LEVEL 243, 394
LOG_SCHEMA_LEVEL 243, 394
LOG_USAGE_LEVEL 243, 244, 394
LOGFILE_SIZE 378
LOGGING 242, 394
MAPFILE 377
MAX_ACTIVE_DATABASES 378
MAX_SERVERS 377
MESSAGE_DIR 378
MINIMUM_LENGTH 396
NLS_LOCALE LOCALE 38
NLS_LOCALE MESSAGE_DIR 43
OLAP_APPROXIMATE_NUMERIC_ FAST_COMPUTATION 387
OPTICAL_AVAILABILITY 309, 393
OPTION ADVISOR_LOGGING 380
OPTION ADVISOR_MAXIMUM_ CANDIDATE_VIEWS 381
OPTION BAR_XBSA_LIB 384
PARALLEL_HASHJOIN 387
PARALLEL_SET_OPERATION 387
PARTIAL_AVAILABILITY 393
PARTITIONED_PARALLEL_ AGGREGATION 387
PERFORMANCE_MONITOR_ COMMANDS_LIMIT 365, 388
PERFORMANCE_MONITOR_ MAXOPERATORS 388
PERFORMANCE_MONITOR_SESSIONS 388
PRECOMPUTED_VIEW_ MAINTENANCE_ON_ERROR 383
PRECOMPUTED_VIEW_MAINTENANCE 382
PRECOMPUTED_VIEW_QUERY_ REWRITE 380
PROCESS_CHECKING_INTERVAL 378
QUERY_MEMORY_LIMIT 389
QUERY_MMAP 383
QUERY_MMAP_LIMIT 391
QUERY_TEMPSPACE_DIRECTORY 389
QUERY_TEMPSPACE_MAXSPILLSIZE 389
QUERYPROCS 385
RANDOM_SAMPLE_MARGIN 391
RBW_LOADINFO_LIMIT 392
READER_PRIORITY 257, 379
REMOTE_TMU_LISTENER 378
RENICE_COMMAND 232, 395
REPORT_INTERVAL 234, 395
RESTRICT_PREVIOUS 395
RESULT_BUFFER 387
RESULT_BUFFER_FULL_ACTION 387
ROWCOUNT 392
ROWS_PER_FETCH_TASK 385
ROWS_PER_SCAN_TASK 385

Configuration parameter *(continued)*
SERVER 377
SERVER_NAME 378
SHMEM 377
specifying in rbw.config 322
TEMPORARY_DATA_SEGMENT 392
TEMPORARY_INDEX_SEGMENT 392
TMU_BUFFERS 384
TMU_COMMIT_RECORD_INTERVAL 381
TMU_COMMIT_TIME_INTERVAL 382
TMU_CONVERSION_TASKS 390
TMU_INDEX_TASKS 390
TMU_INPUT_TASKS 390
TMU_MAX_TASKS 391
TMU_MMAP_LIMIT 391
TMU_OPTIMIZE 388
TMU_ROW_MESSAGES 390
TMU_SERIAL_MODE 390
TMU_VERSIONING 381
TOTALQUERYPROCS 386
TRANSACTION_ISOLATION_LEVEL 381
UNIFIED_LOGON 378
UNIFORM_PROBABILITY_FOR_ADVISOR 380
USE_INVALIDATE_PRECOMPUTED_ VIEWS 380
VERSIONING 381
XML_MAX_TASKS 389
XML_TEMP_SPACE 388
CONNECT system role, described 44, 198
Connections limit
changing 323
rbw.config file entry 377
CONSTRAINT keyword 141
Constraints, weakly selective 85
Contact information xxi
Controlling database activity 230
Copy management utility 225, 318
Costs, of version log 178
COUNT function 379
COUNT_RESULT parameter
modifying 324, 328
rbw.config file entry 379
CREATE INDEX statement, usage 149
CREATE MACRO statement, usage 170
CREATE ROLE statement, usage 204
CREATE SEGMENT statement, usage 132
CREATE TABLE statement
constraint names 141
examples 349
usage 140
CREATE VIEW statement, usage 168
CREATE_ANY task authorization 202
CREATE_OWN task authorization 203
Creating a database, Aroma tutorial 343, 363
CROSS_JOIN parameter
modifying 324, 328
CROSS_JOIN parameter *(continued)*
rbw.config file entry 379
Cross-reference tables 61
CURRENT_USER, SQL variable 170
Customer Support xviii

D

Data
aggregation 75
loading 17
organizing 79
referential integrity during delete 47, 141
Data referential integrity during insert and load 46
Data source name definitions 33
Data sources, accessing with DB2 435
Data types
changing column 314
sizes of 433
Database administrator
DBA account 44
role 13
Database catalog
memory requirements 339
Database directory, defined 26
Database server
described 3
installation 31
limits 14
maximum number of 377
monitoring on UNIX 330
monitoring on Windows 332
on Windows 3
software components 2
stopping on Windows 332
Databases
access to 195, 223
activating 231
adding new 129
adding users 197
backup and restore with versioning 191
catalog
compaction 339
compaction recovery 342
memory requirement 339
removing blocks 340
changing passwords 198, 348
controlling activity 230
copying 294, 318
creating 11
example, Aroma 343, 363
procedure 125, 173
default segment location 133
deleting 11, 336
design 11
dropping database objects 334, 336

Databases *(continued)*
freezing 189
growth patterns 123
implementing 11, 19
limits and maximum sizes 14
loading data 360
locking 255, 261
logical names 27
defining 129
rbw.config file entry 397
maintaining 13
monitoring activity 227
moving 294, 318
organizing data into tables 79
password security 214, 223
quiesce 230
sizing example 101
system account and password 130
table and index sizes, estimating 95
tuning 13
unfreezing 190
when to lock 258
DB2 Information Integrator 435
db2rbsetup program 437
db2rbsetup.exe program 437
DBA system role
described 44, 198
granting 199, 207
revoking 199, 210
dbcreate utility 125
deleting databases 336
example 347
dbsize utility
directory location 96
estimating B-TREE indexes 99
estimating segmented TARGET indexes 100
estimating STAR indexes 99
estimating tables 96
estimating TARGET indexes 100
DDL operations 257
Deadlocks 260
Decision-support databases 56
DEFAULT parameters
entries in rbw.config 392
summary of 402
DEFAULT_DATA_SEGMENT parameter
modifying 324, 328
rbw.config file entry 392
syntax and usage 138
with multiple databases 133
DEFAULT_INDEX_SEGMENT parameter
modifying 324, 328
rbw.config file entry 392
syntax and usage 138
with multiple databases 133
DEFAULT_PSU_EXTENDSIZE parameter
described 92
modifying 324, 328
rbw.config file entries 393
DEFAULT_SEGMENT_SIZE parameter
described 92
modifying 324, 328
rbw.config file entries 393
DELETE operations
locking tables 258
modes, actions 141
versioned 183
Demonstration database, script to install xiv
Dependencies, software xiii
Design of databases 11
DETACH clause, ALTER SEGMENT statement 283
Detaching segments 285
Dimension table
cleaning with versioning 178
defined 58
Directories for temporary space 108, 121
Disk space
allocation 123, 134
offline load requirements 117
organizing with segmented storage 88
reuse 278
table and index sizes, estimating 95
table growth, planning for 123
temporary space requirements, estimating 110, 121
Disk spill files, removing 119
Divide-by-zero errors, options 379
Documentation
list for Red Brick Warehouse xix
Domains, for TARGET indexes
defined 85
specifying size 85
storage representation 86
DROP HIERARCHY statement, usage 335
DROP LAST STORAGE clause, ALTER SEGMENT statement 284
DROP ROLE statement, usage 335
DROP statement 334
DROP_ANY task authorization 202
DROP_OWN task authorization 203
DST_COMMANDS table 424
DST_DATABASES table
column names 426
version log space 191
DST_LOCKS table 257, 427
DST_SESSIONS table 428
DST_USERS table 431
Dynamic statistic tables
and RBW_TABLES table 227
DST_COMMANDS 424
DST_DATABASES 426

Dynamic statistic tables *(continued)*
DST_LOCKS 427
DST_SESSIONS 428
DST_USERS 431
overview 227
refresh interval 234
statistic collection interval 233

E

Environment variables xiv
ODBCINI 33
RB_CONFIG 34
RB_DSN 34
RB_EXE 34
RB_HOME 34
RB_HOST 34
RB_NLS_LOCALE 40, 41
RB_PATH 34
RISQL Entry Tool 196
user accounts 195
Error events 240
Event logging
categories 239, 240
changing severity 244
configuring 242
log daemon 235
log files 241
logging subsystem 235
message templates 236
messages 239
overview 235
severity levels 239
starting 244
startup state 242
USAGE ROUTINE 241
EXPAND statement 172
EXPIRATION_DAYS parameter
rbw.config file entry 395
usage 216, 218
EXPIRATION_WARNING_DAYS parameter
rbw.config file entry 395
usage 217
Expiration, passwords 216
Expired user accounts 216
EXPORT task authorization 202
EXPORT_DEFAULT_PATH parameter
modifying 324, 328
rbw.config file entry 382
EXPORT_DELIMITER parameter
modifying 324, 328
rbw.config file entry 382
EXPORT_MAX_FILE_SIZE parameter
modifying 324, 328
rbw.config file entry 382
Extended star schema 69
EXTENDSIZE value
RBW_TABLES table 293

F

Fact table 58, 72
Fact-to-fact joins
rules for STARjoin 80
Federated access to data sources 435
FILE_GROUP parameter
modifying 324, 328
rbw.config file entry 386
File-system-full messages 275
Fill factor, VARCHAR
changing 315
setting 142
Fill factors, index
changing 154
described 97
effect on key size 154
setting 151
FILLFACTOR parameters
modifying 148, 155, 326, 330
rbw.config file entries 387
usage 153
FILLFACTOR_VARCHAR parameter, rbw.config file entry 387
First login, passwords 219
FIRST_DAYOFWEEK parameter
modifying 324, 328
rbw.config file entry 379
FOR DELETE option 258
FORCE INTACT clause, ALTER SEGMENT statement 284, 294, 306
FORCE TARGETJOIN_TASK parameter
modifying 324
FORCE_AGGREGATION_TASKS parameter
modifying 324, 328
rbw.config file entry 387
FORCE_FETCH_TASKS parameter
modifying 324, 328
rbw.config file entry 386
FORCE_HASHJOIN_TASKS parameter
modifying 324, 328
rbw.config file entry 386
FORCE_JOIN_TASKS parameter
modifying 324, 328
rbw.config file entry 386
FORCE_SCAN_TASKS parameter
modifying 324, 328
rbw.config file entry 386
FORCE_TARGETJOIN_TASKS parameter
rbw.config file entry 386
FOREIGN KEY REFERENCES clause, relating tables 123

Foreign keys
adding and dropping 315
defined 59
multicolumn 62
FREEZE QUERY REVISION clause, ALTER DATABASE statement 184
Frozen versions
controlling 189
defined 183
operations allowed in 184
operations not allowed in 185
overriding 190

G

GRANT authorization and role command
examples 208
usage 199, 207
GRANT CONNECT statement 197
GRANT privilege statement
examples 206
usage 200
GRANT_OWN task authorization 203
GRANT_TABLE task authorization 202
GRANT_TEMP_RESOURCE_TO_ALL parameter
modifying 324
rbw.config file entry 380
GROUP parameter
modifying 324, 328
rbw.config file entry 386
GUI tool, Administrator 34

H

Hash join 87
Hierarchies, dropping 335
High recoverability versioning
described 185
with frozen versions 190

I

IDLE_TIMEOUT parameter
modifying 325, 327
rbw.config file entry 382
IGNORE_OPTICAL_INDEXES parameter
modifying 324, 328
rbw.config file entry 393
syntax and usage 309
IGNORE_PARTIAL_INDEXES parameter
modifying 325, 329
rbw.config file entry 393
IGNORE_QUIESCE task authorization 202
Immediate family, defined 47
Incremental loading 176
INDEX_TEMPSPACE parameters
current values 119
described 109
determining values for 109
INDEX_TEMPSPACE parameters *(continued)*
DIRECTORY 110
location 150
rbw.config file entries 389
use of 110
MAXSPILLSIZE 110
determining size 115
for offline load 117
rbw.config file entries 389
modifying 326, 330
resetting 113
setting 108
THRESHOLD 110
determining value 114
rbw.config file entries 114, 115, 389
use in offline load operations 117
Indexes
B-TREE
attaching segments 285
defined 19
example 84
local 22, 133
size estimates 99
creation guidelines 149
dropping 334
estimating sizes 97
growth, monitoring 275
local
creating 166
defined 22, 133
description 87
sizing 100
overview 19
parallel 150
partial availability 25
segmented TARGET
size estimates 100
segmenting column 287
selection with optical segments 309
STAR
attaching segments 285
creating 81
defined 19
effect of adding rows 123
examples 82
fill factor, changing 155
invalid, cause 123
multiple 19
segmenting 135, 137
size estimates 99
size without MAXROWS PER SEGMENT 123
use of MAXROWS PER SEGMENT option 140
use of MAXSEGMENTS option 140
with simple star schema 59

Indexes *(continued)*
TARGET
attaching segments 285
choosing domain size 85
defined 20
domains, defined 85
domains, storage 86
example 85
hybrid 86
local 22, 133
overview 85
performance 85
selectivity 85
size estimates 100
temporary space requirements 118
verifying creation 151
when to create 79
Indirect role membership
defined 201
example 209, 212
usage 207
INFO_MESSAGE_LIMIT parameter
modifying 324, 328
rbw.config file entry 392
Information Integrator ODBC wrapper 435
Initialization files 12, 31
INITSIZE value, RBW_TABLES table 293
Installation, defining locale during 38
Installing database server 10, 31
INTACT column values 306
Invalid STAR indexes 123
Isolation level 260

J

Java database connectivity 5
JDBC API 5
JDBC Driver 5
Job accounting 248
Joins
between fact tables, indexes for 80
foreign key references 19
relating tables 123

K

Keywords in syntax diagrams xvii

L

LA accounting parameters 249
Language
defined 36
effect of changing 40, 43
nontranslated text 39
Lexical character comparisons 38
Limits, database server 14
Linguistic character comparisons 38
Listener thread 9
LOAD DATA statement examples, Aroma database 355, 361
Load window 175
Loading data
examples, Aroma database 355, 363
incrementally 176
into versioned databases 179
overview 13, 17
time-cyclic data, example 156, 168
while database is online 257
with high recoverability 185
with periodic commit 176
LOCALE parameter
modifying 323, 326
rbw.config file entry 379
Locales
client tool 40
client/server environment 39
code set component 37
components of 36
default 38, 41
defining during installation 38
language component 36
overriding server 39
sort component 37
specifying different for databases 127
system table references 39
territory component 36
LOCK_DATABASE task authorization 202
LOCK_FAILED_ATTEMPTS parameter 396
usage 222
LOCK_PERIOD_HOURS parameter
rbw.config file entry 396
usage 223
Lock-out period, specifying 223
Locked user accounts 222
Locking
READ option 256
types of table locks 257
wait behavior, changing 259
when to lock tables and databases 255, 261
WRITE option 256
LOG _MAXSIZE configuration parameter 241
Log files
Advisor 246
naming conventions 241, 245
removing 242
specifying location 242
specifying size 243
switching 244
types on UNIX 331
types on Windows 333
Log viewer
syntax 236
usage 238

LOG_AUDIT_LEVEL configuration parameter
rbw.config file entry 394
usage 243
LOG_DIRECTORY configuration parameter
rbw.config file entry 394
usage 242
LOG_ERROR_LEVEL configuration parameter
rbw.config file entry 394
usage 243
LOG_MAXSIZE configuration parameter
rbw.config file entry 394
usage 243
LOG_OPERATIONAL_LEVEL configuration parameter
rbw.config file entry 394
usage 243
LOG_SCHEMA_LEVEL configuration parameter
rbw.config file entry 394
usage 243
LOG_USAGE_LEVEL configuration parameter
rbw.config file entry 394
usage 243, 244
logdview executable 236
LOGFILE_SIZE parameter
modifying 323, 326
rbw.config file entry 378
LOGGING configuration parameter
rbw.config file entry 394
usage 242
Logging queries 244
Logical database names
defining 129
described 27
rbw.config file entries 397
Logical I/O statistics 228
Login name, passwords 219

M

Macros
creation and use 170
described 12
dropping 335
examples 172
EXPAND statement 172
resolving references 171
scope 171
Maintaining database 13
Many-to-many relationship
defined by fact table 61
defined by multicolumn foreign key 62
Many-to-one relationship, foreign keys 59
MAPFILE parameter
rbw.config file entry 377
MAX_ACTIVE_DATABASES parameter
modifying 323, 326
rbw.config file entry 378
MAX_SERVERS parameter
modifying 323, 326
rbw.config file entry 377
MAXROWS PER SEGMENT option
and fill factors 143
and VARCHAR fill factor 140
changing value 314
effect on STAR indexes 99
error message 140
forcing REORG operation 140
rows per block 146
unused row numbers 146
usage 140
value in RBW_TABLES 419
MAXSEGMENTS option
changing value 314
effect on STAR indexes 99
forcing REORG operation 140
usage 140
value in RBW_TABLES 419
MAXSIZE value
RBW_STORAGE table 276
RBW_TABLES table 293
MAXSIZE_ROWS value, in RBW_TABLES table 419
MAXSPILLSIZE value
estimating, for queries 121
for INDEX_TEMPSPACE 115
for query temporary slices 121
Membership, roles 201
Memory use, loading and index requirements 110, 121
Memory-mapping parameters 383, 391
Merge phase of optimized index build 110
Message templates 236
MESSAGE_DIR parameter
modifying 323, 326
rbw.config file entry 378
Messages
files 43
translated 43
Microsoft Access, database access with 197
MIGRATE TO clause, ALTER SEGMENT
statement 284, 293
MINIMUM_LENGTH parameter
rbw.config file entry 396
usage 220
MODIFY_ANY task authorization 202
Monitoring
database activity 227
database server processes 330
frozen versions 190
growth of tables and indexes 275
VARCHAR fill factor 146
version log space 191
Multibyte character sets 37
Multiple databases, default segment location 133

Multistar schema 63

N

NLS_LOCALE LOCALE parameter 38
NLS_LOCALE MESSAGE_DIR parameter 43
NLS_LOCALE parameters
  entries in rbw.config 378
  summary of 398
NO ACTION keyword
  described 47
  usage 141
NO WAIT on locks 259
NULL values, in ORDER BY clauses 32

O

Object privileges
  defined 200
  granting 200
  granting to roles 205
  overview 45
ODBC
  client application requirements 197
  driver 5
  initialization files 33
  setting data sources 33
ODBC wrapper, Information Integrator 435
odbc.ini file 33, 442
ODBCINI environment variable 33
OFFLINE clause, ALTER SEGMENT statement 283
Offline load operations, space-planning 117
OFFLINE_LOAD task authorization 203
OLAP_APPROXIMATE_NUMERIC_
  FAST_COMPUTATION parameter
    modifying 324, 328
ONLINE clause, ALTER SEGMENT statement 307
Online transaction-processing databases (OLTP) 55
Operational events 240
Optical segments
  defined 308
  index selection 309
Optical storage support 307, 309
OPTICAL_AVAILABILITY parameter
  modifying 324, 328
  rbw.config file entry 393
  syntax and usage 308
OPTION ADVISOR_LOGGING parameter
  modifying 325, 329
  rbw.config file entry 380
OPTION ADVISOR_MAXIMUM_
  CANDIDATE_VIEWS parameter
    modifying 325
    rbw.config file entry 381
OPTION BAR_XBSA_LIB parameter
  rbw.config file entry 384
OPTION OLAP_APPROXIMATE_NUMERIC_
  FAST_COMPUTATION parameter
    rbw.config file entry 387
OPTION parameters
  entries in rbw.config 379
  summary of 399
OPTION PRECOMPUTED_VIEW_QUERY_REWRITE
  parameter 325, 329
OPTION USE_INVALID_PRECOMPUTED_VIEWS
  parameter
    modifying 329
Out-of-memory errors 114, 150
Out-of-space errors 115
Outboard tables 63
Outrigger tables 63
OVERRIDE FULLINDEXCHECK clause, ALTER
  SEGMENT statement 296
OVERRIDE REFCHECK clause
  caution 47
  usage 142

P

Parallel indexes 150
Parallel processing
  overview 18
Parallel query processing
  number of PSUs 90, 121
PARALLEL_HASHJOIN parameter
  rbw.config file entry 387
PARALLEL_SET_OPERATION parameter
  modifying 324, 328
  rbw.config file entry 387
Parameter 322
PARTIAL_AVAILABILITY parameter
  modifying 325, 329
  rbw.config file entry 288, 393
PARTITIONED_PARALLEL_ AGGREGATION
  parameter
    rbw.config file entry 387
PARTITIONED_PARALLEL_AGGREGATION
  parameter
    modifying 324, 328
PASSWORD parameters
  ALLOW_LOGNAME 219
  CHANGE_INITIAL 219
  CHANGE_MINIMUM_DAYS 218
  COMPLEX_NUM_ALPHA 221
  COMPLEX_NUM_NUMERICS 221
  COMPLEX_NUM_PUNCTUATION 221
  COMPLEX_NUMERIC 219
  described 215
  entries in rbw.config 397
  EXPIRATION_DAYS 216
  EXPIRATION_WARNING_DAYS 217
  LOCK_FAILED_ATTEMPTS 222

PASSWORD parameters *(continued)*
LOCK_PERIOD_HOURS 223
MINIMUM_LENGTH 220
modifying 326, 330
RESTRICT_PREVIOUS 217
summary of 402
Passwords
change at first login 219
change at first login enforcing 219
changes, enforcing 216
changing 198
changing system default 348
complexity and length 221
expiration 216
expiration warning 217
frequency of changes 218
limiting reuse 217
lock-out period 223
locked accounts 222
login name 219
minimum length 220
new users 197
restrictions, standard 198
security 214
security parameters 215
standard restrictions 130
system account, changing 130
use login name 219
Pathnames, relative and full 319
Performance
schema design 58
STAR indexes 81
TARGET indexes 85
PERFORMANCE_MONITOR_ COMMANDS parameter
rbw.config file entry 388
PERFORMANCE_MONITOR_ MAXOPERATORS parameter
rbw.config file entry 388
PERFORMANCE_MONITOR_COMMANDS parameter
modifying 325
PERFORMANCE_MONITOR_COMMANDS_ LIMIT parameter
modifying 255
PERFORMANCE_MONITOR_COMMANDS_LIMIT parameter
modifying 329
PERFORMANCE_MONITOR_MAXOPERATORS parameter
modifying 323, 326
PERFORMANCE_MONITOR_SESSIONS parameter
modifying 323, 326
rbw.config file entry 388
Periodic commit 176
Physical I/O statistics 228
Precomputed views 25
PRECOMPUTED_VIEW_MAINTENANCE parameter
modifying 325, 329
rbw.config file entry 382
PRECOMPUTED_VIEW_MAINTENANCE_ ON_ERROR parameter
modifying 325, 329
rbw.config file entry 383
PRECOMPUTED_VIEW_QUERY_REWRITE parameter
modifying 325, 329
rbw.config file entries 380
Primary key, defined 58
Private macros 171
Process checker daemon 9
PROCESS_CHECKING_INTERVAL parameter
modifying 323, 326
rbw.config file entry 378
Processes
defined 6
monitoring 330
Pseudocolumns 136, 278
PSUs
allocating disk space 134
cannot open 294
changing location 293
changing size 293
damaged 305
extend size 92
for indexes 134
for version log 181
MAXSIZE value 276
moving 293
sequence ID 417
size and growth 91
PUBLIC access 200
Public macro 171
PUBLIC_MACROS task authorization 203

Q

Queries, logging 241, 244
Query sampling 26
QUERY_MEMORY_LIMIT parameter
estimating value of 120
modifying 325, 329
QUERY_MMAP parameter 383
QUERY_MMAP_LIMIT parameter 391
QUERY_TEMPSPACE parameters 108
current values 119
described 120, 121
DIRECTORY
rbw.config file entries 389
use of 110
effect on queries 120
MAXSPILLSIZE
rbw.config file entries 389
value 121

QUERY_TEMPSPACE parameters *(continued)*
modifying 325, 329
QUERY_MEMORY_LIMIT
rbw.config file entries 389
QUERY_TEMPSPACE_DIRECTORY parameter
rbw.config file entry 389
QUERY_TEMPSPACE_MAXSPILLSIZE parameter
rbw.config file entry 389
QUERYPROCS parameter
modifying 323, 326
rbw.config file entry 385
Quiesce, database 230

R

RAID devices, with version log 187
RANDOM_SAMPLE_MARGIN parameter
modifying 325, 329
rbw.config file entry 391
RANGE clause, ALTER SEGMENT statement 284
RANGE LIKE SEGMENT clause
ALTER SEGMENT ATTACH statement 135
RANGE MOVE clause, ALTER SEGMENT statement 284
rb_cm utility 225, 318
RB_CONFIG environment variable
described 34
user accounts 195
rb_creator utility 125, 347
RB_DEFAULT_LOADINFO file 129
rb_deleter utility, deleting databases 336
RB_DSN environment variable 34
RB_EXE environment variable 34
RB_HOME environment variable 34
RB_HOST environment variable
described 34
user accounts 195
RB_NLS_LOCALE environment variable 40, 41
RB_PATH environment variable
described 34
user accounts 196
rb_sample.cleanup script, usage 119
rb_syscompact 339, 340, 342
rbclean.bat template, usage 119
RBMON parameter 398
rbodbc32.ini file 33
RBW_COLUMNS table 407
RBW_CONSTRAINT_COLUMNS table 408
RBW_CONSTRAINTS table 141, 408
RBW_HIERARCHIES table 409
RBW_INDEXCOLUMNS table 409
RBW_INDEXES table 410
RBW_LOADINFO table 410
RBW_LOADINFO_LIMIT parameter
modifying 324, 328
rbw.config file entry 392
RBW_MACROS table 411
RBW_OPTIMIZER_OPTIONS table 406, 412
RBW_OPTIONS table 39, 412
RBW_PRECOMPVIEW_CANDIDATES table 406
RBW_PRECOMPVIEW_COLUMNS table 413
RBW_PRECOMPVIEW_UTILIZATION table 406
RBW_RELATIONSHIPS table 141, 413
RBW_ROLE_MEMBERS table 212, 414
RBW_ROLES table 212, 414
RBW_ROWNUM pseudocolumn 136, 278, 285
RBW_SEGID pseudocolumn 278
RBW_SEGMENTS table 31, 415
RBW_SEGNAME pseudocolumn 136, 278
RBW_STORAGE table 417
RBW_SYNONYMS table 417
RBW_TABAUTH table 212, 418
RBW_TABLES table 418
RBW_USERAUTH table 212, 419
RBW_VIEW_REFERENCES table 422
RBW_VIEWS table 422
RBW_VIEWTEXT table 423
rbw.show script, usage 331
rbw.start script
rbwadmd daemon 331
usage 331
rbw.stop script, usage 331
rbwadmd daemon
details 232
output 43
starting 331
statistic collection interval 233
stopping 331
rbwadmd thread
starting 331
stopping 332
RBWAPI parameters
entries in rbw.config 377
summary of 397
rbwapid daemon
described 7
rbw.config file entries 377
rbwlogd daemon
defined 8
output 43
role in accounting 249
starting 235
stopping 244
rbwlogview executable 236
rbwlsnr thread 9
rbwpchk daemon 9
rbwsvr process 7
rbwvcd daemon 9
READ locking option 256
Read locks 257
Read statistics 228

READER_PRIORITY parameter
modifying 323, 326
rbw.config file entry 379
reducing contention 257
RECOVER option 185, 186
Recovering a damaged segment 305
Recovery procedures 14
Red Brick Warehouse
component overview 2, 5
redbrick
administrative user 11
directory permissions 128
user account 44, 345
user umask setting 128
Referenced tables
defined 58
outboard, outrigger 63
Referencing tables 58
Referential integrity
defined 46
delete behavior, defining 141, 142
delete operations 47
load and insert operations 46
ON DELETE clause 141
RELEASE STORAGE clause, ALTER SEGMENT statement 284
REMOTE_TMU_LISTENER parameter
modifying 323, 326
rbw.config file entry 378
REMOVE DAMAGED SEGMENTS option, ALTER DATABASE CLEAN VERSION LOG statement 188
RENAME clause, ALTER SEGMENT statement 284
Renaming segments 289
RENICE_COMMAND parameter
rbw.config file entry 395
user priorities 232
REORG operations
after ALTER SEGMENT 124
effect of MAXROWS PER SEGMENT option 140
effect of MAXSEGMENTS option 140
REORG_ANY task authorization 203
REPORT_INTERVAL parameter
modifying 326, 330
rbw.config file entry 395
usage 234
Resetting statistics 231
Resource privileges, with ODBC 197
RESOURCE system role
described 44, 198
granting 199, 207
revoking 199, 210
Restore operations
during index creation 149
on versioned databases 191
RESTRICT_PREVIOUS parameter
rbw.config file entry 395
usage 217
Restricted delete operations 47, 141
RESULT_BUFFER parameter
described 121
modifying 325, 329
rbw.config file entry 387
RESULT_BUFFER_FULL_ACTION parameter
described 122
modifying 325, 329
rbw.config file entry 387
Reuse by row 278
Reusing segments 295
REVOKE authorization and role command 210
RISQL Entry Tool and RISQL Reporter
described 4
invoking 348
locales for 40
user accounts 195
ROLE_MANAGEMENT task authorization 203
Role-based security 201
Roles
direct and indirect membership 201
dropping 335
granting 199, 207, 210
membership 207, 210
revoking 199, 210
system 198
system table information 212
user-created 201, 214
Row IDs
description 135
in CREATE STAR INDEX statement 90, 135
Row numbers
description 279
how database server uses 142
in CREATE STAR INDEX statement 90, 135
in row IDs 135
moving range 288
per block 144
RBW_ROWNUM pseudocolumn 285
RBW_SEGNAME pseudocolumn 279
rows per block 142, 147
specifying range 285, 288
unused 143, 146
VARCHAR fill factor effect 143
ROWCOUNT parameter
modifying 324, 328
rbw.config file entry 392
Rows per block
calculation 142
CHECK TABLE output 147, 148
MAXROWS PER SEGMENT 146
number of row numbers 142

Rows per block *(continued)*
typical row size 145
VARCHAR fill factor effect 144, 145, 146
ROWS_PER_FETCH_TASK parameter
modifying 324, 328
rbw.config file entry 385
ROWS_PER_JOIN_TASK parameter
modifying 324, 328
rbw.config file entry 385
ROWS_PER_SCAN_TASK parameter
modifying 324, 328
rbw.config file entry 385
ROWS_PER_TARGETJOIN_TASK parameter
modifying 324, 328
rbw.config file entry 385
Rows, deleting
locking tables 258
referential integrity 46

S

Sampling, query results 26
Schema events 241
Schema locking 256
Scratch space, estimating requirements 109
Secondary dimension tables 63
Security
passwords 214
role-based 201
SEGMENT BY clause
ALTER SEGMENT statement 284
CREATE INDEX statement 90, 91, 135
CREATE TABLE statement 90, 134
SEGMENT LIKE clause
CREATE INDEX statement 91, 135
SEGMENT LOCAL clause
CREATE INDEX statement 91
Segment parameters
entries in rbw.config 393
SEGMENT parameters
summary of 401
Segmented STAR index picture 166
Segmented storage
default location 138
described 20
implementation of 21
planning for use 88, 124
Segments
adding a new segment 300
adding space to 281
altering 283, 294
attaching 285
automatic locking 258
clearing 289
creation guidelines 123, 132, 134
damaged 294
Segments *(continued)*
damaged, recovering 305, 307
default
changing to named 89
defined 21
DEFAULT_SEGMENT_SIZE parameter 92
defined 20
detaching 285
distributing data among 22
dropping 335
intact 294
moving 293
named 20, 88
online, offline 24, 288
synchronizing 288
optical, defined 308
partial availability 25
range, changing 288
range, moving 288
renaming 289
reusing 295
rolling off 295
space available 275, 276
time-cyclic data, example 156, 168
verifying 294
SEGMENTS parameter
modifying 325, 329
rbw.config file entry 392
usage 139
Selectivity, defined 85
SERIAL columns, changing start and step values for 316
SERVER parameter
modifying 323, 326
rbw.config file entry 377
Server process 7
SERVER_NAME parameter
modifying 323, 326
rbw.config file entry 378
Service, warehouse, overview of 6
Session
changing user priority 232
closing 231
SET commands, SQL
DEFAULT SEGMENT STORAGE PATH 138
IGNORE OPTICAL INDEXES 309
OPTICAL AVAILABILITY 309
overview 34
PARTIAL AVAILABILITY 288
REPORT_INTERVAL 234
RESULT BUFFER 121
RESULT BUFFER FULL ACTION 122
SEGMENT 139
specifying 322
TMU COMMIT RECORD INTERVAL 177

SET commands, SQL *(continued)*
TMU COMMIT TIME INTERVAL 177
TMU VERSIONING 186
TRANSACTION ISOLATION LEVEL 260
USE LATEST REVISION 190
VERSIONING 186
Shared memory 10
SHMEM parameter
modifying 323
rbw.config file entry 377
Size estimates
B-TREE indexes 99
example, database sizing 101
segmented TARGET indexes 100
STAR indexes 99
system tables 106
tables and indexes 95
TARGET indexes 100
SKU number 64
Software dependencies xiii
Sort component
defined 37
effect of changing 40
Sort phase of optimized index build 110
SQL statement, canceling 231
Star schemas 78
described 57
design considerations 65
examples 67, 78
performance 58
supported types 78
use of attributes 72
STARjoin
defined 19
START ACCOUNTING clause, ALTER SYSTEM statement 253
Start and step values for SERIAL columns 316
START CLEANING clause, ALTER DATABASE statement 193
Starting, database server 331
Statistics
collection interval 233
platform effect on availability of 230
read and write 228
resetting 231
STOP ACCOUNTING clause, ALTER SYSTEM statement 253
STOP CLEANING clause, ALTER DATABASE statement 193
Stopping, database server 331
Storage devices, optical 307
SuperScan technology 19
SWITCH ACCOUNTING FILE clause, ALTER SYSTEM statement 254
Synchronizing segments 288
Synonyms, dropping 336
Syntax diagrams
keywords in xvii
variables in xvii
System accounts, for new users 195
System catalog
compaction 340
contents 405
System requirements xiii
System roles
described 198
granting 199, 207
overview 44
revoking 199, 210
System tables
described 405, 423
references to locale 39
size estimates 106
System, database user account 130

T

Table Management Utility (TMU) 4
Aroma examples 360, 363
enabling versioning 186
loading with periodic commit 177
TUNE parameters for 392
Tables
access, limiting with views 169
adding rows, effect on STAR index 123
altering 310
attaching segments 285
browsing 71
creating Aroma tables 349
creation guidelines
foreign key order 140
order 140
referential integrity on delete 140
setting MAXROWS PER SEGMENT 140
setting MAXSEGMENTS 140
dropping 336
families, immediate and complete 47
growth 123
growth, monitoring 275
in segments 133
locking 255, 261
multiple fact tables 60
optical storage 308
partial availability 25
segmenting column 287
size estimates 96
sizing 96
system tables, size estimates 106
transient 123
when to lock 258

TARGETjoin
defined 20
local index 23, 133
TARGETJOIN_LOCAL_PREDICATES parameter
modifying 324, 328
Task authorizations
ACCESS_SYSINFO 226
ALTER_SYSTEM 226
defined 201
for managing database activity 226
granting 205
Technical support xviii
TEMP_RESOURCE task authorization 203
Temporary macros
deleting 335
scope 171
Temporary space 121
for creating TARGET indexes 118
for index-building operations 108
for queries 108
for results of a query 108
removing files 119
Temporary tables 96
TEMPORARY_DATA_SEGMENT parameter
modifying 324, 328
rbw.config file entries 392
TEMPORARY_INDEX_SEGMENT parameter
modifying 324, 328
rbw.config file entries 392
Territory
defined 36
effect of changing 40, 43
Thread, defined 6
Time-cyclic data
local indexes for 87
planning for 92
STAR index for 93, 155
Time-cyclic data, example 156, 168
TMU 186
TMU_BUFFERS parameter
modifying 325, 329
rbw.config file entry 384
TMU_COMMIT_RECORD_INTERVAL parameter
modifying 325, 329
rbw.config entry 381
using for periodic commit 177
TMU_COMMIT_TIME_INTERVAL parameter
modifying 325, 329
rbw.config entry 382
using for periodic commit 177
TMU_CONVERSION_TASKS parameter
modifying 325, 329
rbw.config file entry 390
TMU_INDEX_TASKS parameter
modifying 325, 329
TMU_INDEX_TASKS parameter *(continued)*
rbw.config entry 390
TMU_INPUT_TASKS parameter
modifying 325, 329
rbw.config entry 390
TMU_MAX_TASKS parameter
modifying 325, 329
rbw.config entry 391
TMU_MMAP_LIMIT parameter
modifying 325, 329
rbw.config entry 391
TMU_OPTIMIZE parameter
modifying 325, 329
rbw.config file entry 388
TMU_ROW_MESSAGES parameter
modifying 325, 329
rbw.config entry 390
TMU_SERIAL_MODE parameter
modifying 325, 329
rbw.config entry 390
TMU_VERSIONING parameter
enabling versioning 186
modifying 325, 329
rbw.config file entry 381
RECOVER option 186
TOTALFREE column, RBW_SEGMENTS table 278
TOTALQUERYPROCS parameter
modifying 323, 326
rbw.config file entry 386
TRANSACTION_ISOLATION_LEVEL parameter
modifying 325, 329
rbw.config file entry 381
Transaction-processing databases 55
Transactions
defined 45
single statement 45
Transient tables 123
Translated text 39
Trickle feed applications 176
Troubleshooting xix
TUNE parameters
entries in rbw.config 392
OPTICAL_AVAILABILITY 309
summary of 398
Tuning, database 13

U

UNFREEZE QUERY REVISION clause, ALTER DATABASE statement 190
UNIFIED_LOGON parameter
modifying 326
rbw.config file entry 378
UNIFORM_PROBABILITY_FOR_ADVISOR parameter
modifying 325, 328, 329
rbw.config file entry 380

Unloading data
frozen versions 190
overview 13
UPC level 70
UPGRADE_DATABASE task authorization 203
Usage events 241
USAGE ROUTINE event 241, 244
USE_INVALID_PRECOMPUTED_VIEWS parameter
modifying 325, 329
rbw.config file entries 380
USED column, RBW_STORAGE table 277
User access, providing 12
User accounts
adding new users 195, 198
expired 216
locked 222
RISQL Entry Tool 195
User priority, changing 232
USER_MANAGEMENT task authorization 203
User-created roles
creating 204
direct and indirect membership 201
dropping 335
granting 207, 210
managing 201, 214
revoking 210
User, redbrick 11, 26, 44
USER, SQL variable 170
Users
adding to a database 197
changing passwords 198
types of xiii
Utility programs
copy management 225
dbsize 96
rb_cm 225

V

Vacuum cleaner daemon
controlling 192
description 9
manually starting 193
manually stopping 193
VARCHAR columns
changing from CHAR 314
fill factor effects 142
modifying fill factor 148
monitoring fill factor 146
setting fill factor for 142
Variables, in syntax diagrams xvii
VERIFY clause, ALTER SEGMENT statement 284, 294, 307
damaged segments 305
Version log
adding space to 188
Version log *(continued)*
allocating space 187
blocks of data in 181
cleaning 188
costs 178
creating 187
deleted blocks in 183
dropping 188
enabling 186
high recoverability for 185
monitoring 190
monitoring databases 190
PSUs for 181
sizing 187
structure of 181
tuning for performance 187
understanding 180
using as a write-ahead log 185
verifying if empty 188
Version of database server software 333
Versioned databases
Aroma example 193
described 45
Versioned transactions, defined 45
Versioning
cleaning dimension tables 178
controlling the vacuum cleaner 192
cost of 178
database availability 175
deleting blocks 183
determining if needed 175
enabling 186
frozen versions, controlling 189
frozen versions, understanding 183
high recoverability and frozen versions 190
high recoverability option 185
loading data with 179
options for 186
periodic commit option 176
with Vista 177
VERSIONING parameter
enabling versioning 186
modifying 325, 329
rbw.config file entry 381
Views
creation guidelines 168
described 65
dropping 336
precomputed 25
Vista
Advisor logging commands 245
aggregate maintenance 26
described 25
query sampling 26
versioning with 177

Visual Basic, database access with 197

W

WAIT/NO WAIT on locks 259
Warehouse processes
  daemon process 7
  listener thread 9
  log daemon process 8
  overview 5, 10
  process checker daemon 9
  server process 7
  vacuum cleaner daemon 9
Warehouse service on Windows
  defined 7
  monitoring 332
  starting 333
  stopping 332
Weak selectivity
  defined 85
  improving performance for queries with 85
working segment 311
Workload accounting 249
WRITE locking option 256
Write locks 257
Write statistics 229

X

XML_MAX_TASKS parameter
  modifying 325, 329
  rbw.config file entry 389
XML_TEMP_SPACE parameter
  modifying 325, 329
  rbw.config file entry 388

Printed in USA

SC18-7395-00