## About This Document

- **Purpose**: xvii
- **Audience**: xvii
- **Organization**: xviii
- **Related Documentation**: xix
- **Conventions**: xxi
  - Syntax Notation: xxi
  - Syntax Diagrams: xxii
  - Keywords and Punctuation: xxiv
  - Identifiers and Names: xxiv
- **Customer Support**: xxv
  - Support Solutions Warehouse: xxv
  - General and Technical Questions: xxv
  - Troubleshooting Tips: xxvii
  - Documentation Questions and Comments: xxvii

## 1 Introduction to Red Brick Warehouse

### Red Brick Technology

- 1-2

### Warehouse Components

- 1-3
  - Red Brick Warehouse Server: 1-4
  - Table Management Utility: 1-4
  - RISQL Entry Tool and RISQL Reporter: 1-5
  - Red Brick ODBC Driver: 1-5
  - Red Brick Warehouse Administrator Tool: 1-5
  - Database: 1-5

### Warehouse Processes

- 1-6
  - Interprocess Communication: 1-6
  - Warehouse Daemon Process: 1-7
  - Server Processes: 1-7
  - Administration Daemon Process: 1-7
  - Log Daemon Process: 1-7

### Warehouse Administration Overview

- 1-8
  - Installing Red Brick Warehouse: 1-8
  - Planning the Database Design: 1-9
  - Implementing the Database: 1-9
  - Providing User Access: 1-10
  - Loading the Data: 1-11
  - Maintaining the Database and Tuning for Performance: 1-11

### Sample Database

- 1-12

### Warehouse Limits

- 1-13
2 **Key Concepts**

- Database Loading 2-2
- Parallel Processing 2-3
- Physical Implementation of Databases 2-4
  - Indexes and Retrieval Strategies 2-4
  - Segmented Storage 2-5
  - Precomputed Views for Increased Query Performance 2-11
- Database Directories and Files 2-12
  - Logical Database Names 2-12
  - Segment Names 2-15
- Configuration and Initialization 2-16
  - Configuration File 2-16
  - Initialization Files 2-16
  - SET Commands 2-18
  - Environment Variables 2-18
  - Red Brick Warehouse Administrator Tool 2-19
- Warehouse Locale 2-21
  - Components of a Locale 2-21
  - Defining the Warehouse Locale 2-24
  - Overriding the Warehouse Locale 2-26
  - Ensuring Client/Server Compatibility 2-28
- File Ownership and Permissions 2-31
- Database Authorizations and Privileges 2-32
- Table and Database Locks 2-33
- Referential Integrity 2-35
  - Load and Insert Operations 2-35
  - Delete Operations and Cascaded Deletes 2-35
3 Schema Design

Transaction Processing Versus Decision Support  3-2
  Transaction-Processing Databases  3-2
  Decision-Support Databases  3-3
Star Schemas  3-5
  Performance of Star Schemas  3-6
  Terminology  3-6
  Simple Star Schemas  3-6
  Multi-Star Schemas  3-11
  Views  3-13
Considerations for Schema Design  3-14
Common Schema Examples  3-16
Example: Salad Dressing Database  3-18
  Browsing the Dimension Tables  3-19
  Querying the Fact Table  3-19
  Using Table Attributes  3-20
  Determining Which Attributes to Include  3-20
Example: Reservation System Database  3-22
Example: Investment Database  3-23
Example: Health Insurance Database  3-25
Contents

4 Planning a Database Implementation

Organizing Data into Databases 4-2
Determining When to Create Additional Indexes 4-3
  STAR Indexes 4-4
  B-TREE Indexes 4-7
  TARGET Indexes 4-8
  No Indexes 4-10
Planning For TARGETjoin Processing 4-11
  STARjoin Versus TARGETjoin 4-11
  Administration Considerations for TARGETjoin Processing 4-12
  TARGET Index DOMAIN Clause 4-16
Planning Disk Storage Organization 4-18
Estimating the Size of User Tables and Indexes 4-20
  Estimating Table Sizes 4-20
  Estimating Index Sizes 4-21
Example: Calculating Table, Index, and System Table Sizes 4-24
  Fact1 Table and Its Indexes 4-25
  Market Table and Its Indexes 4-26
  Product Table and Its Indexes 4-28
  Estimating the Size of System Tables 4-29
  Total Space for User Tables, Indexes, and System Tables 4-31
Planning for Temporary Space Requirements 4-32
  How Optimized Index-Building Operations Use Temporary Space 4-33
  Estimating Temporary Space Values for Index-Building Operations 4-34
  How Query Operations Use Temporary Space 4-39
Planning for Segmented Storage 4-41
Considerations for Growing Tables 4-44
5 Creating a Database

Overview 5-2
Creating the Database Structure 5-3
   Initializing the Database 5-3
   Defining a Logical Database Name 5-4
   Changing the DBA Account Password 5-5
Creating the Database Objects 5-7
Creating Segments 5-8
Creating Tables 5-9
   Setting the MAXSEGMENTS and MAXROWS PER SEGMENT Parameters 5-9
   Naming Constraints for Primary and Foreign Keys 5-10
   Maintaining Referential Integrity with ON DELETE 5-10
Creating Indexes 5-12
   INDEX TEMPSPACE Parameters 5-12
   Parallel Indexes 5-13
   Large Dimension Tables 5-13
   Tables with Multiple Indexes 5-13
   STAR Indexes 5-14
   TARGET Indexes 5-14
Creating Views 5-15
Creating and Managing Macros 5-17
   Guidelines for Macro Definitions 5-17
   Availability and Scope 5-18
   Examples 5-19
Contents

6 Providing Database Access and Security

Adding Database Users 6-2
Creating Operating System Accounts for Users 6-2
Granting Database Access 6-3
Changing Passwords 6-4
Granting Access with System Roles 6-5
  DBA, RESOURCE, and CONNECT Capabilities 6-5
  Granting and Revoking the DBA and RESOURCE System Roles 6-6
Granting Database Object Privileges 6-7
Granting Access with Role-Based Security 6-8
  Task Authorizations 6-9
  Role Capabilities 6-11
  Creating Roles 6-12
  Granting Task Authorizations 6-12
  Granting Object Privileges to Roles 6-13
  Granting Roles 6-14
  Revoking Task Authorizations, Object Privileges, and Roles 6-18
  Tracking Role Authorizations and Members 6-20
Administering Password Security 6-24
  Enforcing Password Changes 6-26
  Warning Users of Password Expiration 6-27
  Limiting Reuse of Previous Passwords 6-28
  Limiting Frequency of Password Changes 6-29
  Enforcing Password Complexity and Length 6-30
  Locking User Accounts after Failed Connection Attempts 6-32
  Specifying the Lock-Out Period 6-33
7 Managing Database Activity with the Enterprise Control and Coordination Option

Enterprise Control and Coordination  7-2
Managing Database Activity  7-4
Task Authorizations for Managing Database Activity  7-4
Administration Database  7-4
Monitoring Database Activity  7-5
Controlling Database Activity  7-8
Administration Daemon  7-11
The Statistics Collection Interval  7-11
The DST Refresh Interval  7-12
Event Logging  7-13
Event Logging Process  7-13
Event Log Messages  7-17
Log Files  7-19
Configuring the Logging Subsystem  7-20
Controlling Logging Operations  7-22
Controlling Advisor Logging  7-24
Advisor Log Files  7-24
What the Advisor Logs  7-25
Starting and Stopping the Advisor Log  7-26
ADMIN ADVISOR_LOG_DIRECTORY  7-29
ADMIN ADVISOR_LOG_MAXSIZE  7-29
SET UNIFORM PROBABILITY FOR ADVISOR  7-30
Accounting  7-31
Accounting Process  7-32
Format of Accounting Records  7-32
Accounting Files  7-33
Configuring Accounting  7-34
Controlling Accounting  7-36
Contents

8 Maintaining a Data Warehouse

Locking Tables and Databases  8-2
  Determining When to Lock a Table or Database  8-2
  Specifying Wait Behavior for the Server and TMU  8-3
Monitoring Growth of Tables and Indexes  8-4
  STAR Indexes  8-4
  MAXSIZE Column  8-5
  USED Column  8-6
  TOTALFREE Column  8-6
  Pseudocolumns  8-6
Adding Space to a Segment  8-8
Changing an Index Fill Factor  8-10
Altering Segments  8-11
  ALTER SEGMENT Operations  8-11
  Locking the Database First  8-12
  Attaching and Detaching Segments  8-12
  Moving Entire Segments  8-13
  Specifying a Segmenting Column  8-13
  Specifying a Range  8-13
  Taking a Segment Offline or Online  8-13
  Clearing a Segment  8-14
  Renaming a Segment  8-14
  Changing PSU Sizes  8-14
  Changing PSU Location  8-15
  Verifying a Segment  8-15
  Forcing a Segment into an Intact State  8-16
Recovering a Damaged Segment  8-17
Managing Optical Storage  8-19
  Assigning Optical Storage  8-19
  Specifying Access Behavior for Optical Segments  8-20
  Specifying Index Selection with Optical Segments  8-22
Altering Tables  8-24
  Adding and Dropping Columns  8-24
  Changing a Column’s Name  8-25
  Changing a Column’s Default Value  8-25
  Changing the MAXSEGMENTS and MAXROWS PER SEGMENTS Values  8-26
  Changing How Referential Integrity Is Maintained  8-26
  Changing a Column’s Datatype  8-26
  Adding and Dropping Foreign Keys  8-27
  Recovering from an Interrupted ALTER TABLE Operation  8-28
Copying or Moving a Database  8-30
Modifying the Configuration File  8-34
Monitoring and Controlling Warehouse Processes  8-36
  Warehouse Daemon  8-36
  Findserver Utility  8-36
  Log Files  8-38
  Query Logging  8-38
  Enabling Licensed Options  8-39
  Determining Version Information  8-39
  Deleting Database Objects and Databases  8-40
    Dropping Database Objects  8-40
    Deleting a Database  8-43

9 Tuning a Warehouse for Performance
  Specifying Parameters with rbw.config File Entries or SET Commands  9-2
  Setting Temporary Space Parameters  9-2
    Temporary Space Parameters  9-3
    How Temporary Space Is Allocated  9-4
    TEMPSPACE Syntax  9-7
    Determining Current Values  9-10
    Removing Temporary Files  9-10
    Setting QUERY_MEMORY_LIMIT  9-11
  Setting the Result Buffer for Long-Running Queries  9-13
    RESULT BUFFER Parameter Syntax  9-13
    RESULT BUFFER FULL ACTION Parameter Syntax  9-14
  Setting Segment and Partial Availability Behavior  9-15
    Location of Default Segments  9-15
    Segment Drop Behavior  9-16
    Query Behavior on Partially Available Tables  9-17
    Use of Partially Available Indexes  9-18
  Setting the Index Fill Factor  9-20
  Creating Additional Indexes  9-24
  Understanding Red Brick Query Processing  9-25
    Join Algorithms  9-25
    Operator Model  9-28
    EXPLAIN Command  9-37
  TARGETjoin Query Processing  9-40
    How to Use TARGETjoin Processing  9-40
    When to Use TARGETjoin Processing  9-42
    Examples  9-44
    Reading EXPLAIN Output on a TARGETjoin Query  9-47
    Summary and Recommendations  9-51
  Using Synonyms to Control Fact-to-Fact Joins  9-54
  Making SQL-Based Improvements  9-57
    UNION Versus Interdimensional ORs  9-57
    Subquery in the FROM Clause Versus Correlated Subquery  9-57
Contents

10 Tuning a Warehouse for Parallel Query Processing

Parallel Query Tuning Parameters 10-2
Enabling Parallel Query Processing 10-3
Limiting I/O Contention with the FILE_GROUP Parameter 10-4
Increasing Disk Group Processes with the GROUP Parameter 10-6
Limiting Available Processes 10-7
   TOTALQUERYPROCS 10-8
   QUERYPROCS 10-8
   Using TOTALQUERYPROCS and QUERYPROCS 10-9
Setting Minimum Row Requirements with ROWS_PER_TASK Parameters 10-10
   ROWS_PER_SCAN_TASK 10-11
   ROWS_PER_FETCH_TASK and ROWS_PER_JOIN_TASK 10-13
Forcing the Number of Parallel Tasks with the FORCE_TASKS Parameters 10-16
   FORCE_TASKS Syntax 10-16
   FORCE_SCAN_TASKS 10-18
   FORCE_FETCH_TASKS and FORCE_JOIN_TASKS 10-19
   FORCE_HASHJOIN_TASKS 10-21
Enabling Partitioned Parallelism for Aggregation 10-22
   Partitioned Parallelism Syntax 10-22
   Usage Notes 10-23
Other Limitations to Parallelism 10-24
Analysis of System Resources and Workload 10-26
   Disk Usage 10-26
   Memory Usage 10-27
   CPU Allocation 10-29
Tuning for Specific Query Types 10-30
   Parallel STARjoin Queries 10-30
   Parallel Table Scans 10-32
   SuperScan Technology 10-33
   About Reasonable Values 10-33
Summary: Basic Guidelines 10-34
### A Example: Building a Database

- Building the Aroma Database **A-2**
  - The redbrick Directory and Aroma Input Files **A-2**
- Steps for Building a Database **A-3**
- Logging in as redbrick **A-3**
- Making the Database Directory **A-4**
- Creating the Database **A-5**
- Changing the Default Password **A-6**
- Creating the User Tables **A-7**
  - The CREATE TABLE Statements **A-7**
  - Procedure **A-9**
- Writing the LOAD DATA Statements **A-14**
  - The Period Table **A-14**
  - The Product Table **A-16**
  - The Market Table **A-17**
  - The Sales Table **A-18**
- Loading the Data **A-20**
- Verifying the Database **A-23**
- Summary **A-23**

### B Configuration File

- Sample rbw.config File **B-2**
- Description of File Elements **B-10**
  - Configuration Information **B-10**
  - License Keys **B-12**
  - Tuning and Performance Parameters **B-13**
  - Default Parameters **B-17**
  - Segment Parameters **B-18**
  - ADMIN Parameters **B-19**
  - PASSWORD Parameters **B-21**
  - Database Entries **B-22**
- Summary of Configuration Parameters **B-23**
Contents

C System Tables and Dynamic Statistic Tables

System Catalog  C-2
  RBW_COLUMNS Table  C-4
  RBW_CONSTRAINTCOLUMNS Table  C-5
  RBW_CONSTRAINTS Table  C-6
  RBW_HIERARCHIES Table  C-6
  RBW_INDEXCOLUMNS Table  C-7
  RBW_INDEXES Table  C-7
  RBW_LOADINFO Table  C-8
  RBW_MACROS Table  C-9
  RBW_OPTIONS Table  C-10
  RBW_PRECOMPVIEW_CANDIDATES Table  C-10
  RBW_PRECOMPVIEW_UTILIZATION Table  C-10
  RBW_PRECOMPVIEWCOLUMNS Table  C-11
  RBW_RELATIONSHIPS Table  C-12
  RBW_ROLE_MEMBERS Table  C-13
  RBW_ROLES Table  C-14
  RBWSEGMENTS Table  C-15
  RBW_STORAGE Table  C-17
  RBW_SYNONYMS Table  C-18
  RBW_TABAUTH Table  C-19
  RBW_TABLES Table  C-20
  RBW_USERAUTH Table  C-21
  RBW_VIEWS Table  C-25
  RBW_VIEWTEXT Table  C-26

Dynamic Statistic Tables  C-27
  DST_COMMANDS Table  C-27
  DST_DATABASES Table  C-29
  DST_LOCKS Table  C-30
  DST_SESSIONS Table  C-31
  DST_USERS Table  C-35

Datatypes and Their Sizes  C-39
D  Example: Using Segments with Time-Cyclic Data
   Background  D-2
      The Data Segments  D-3
      The Index Segments  D-5
      The Tables  D-7
      The STAR Index  D-8
      The Data  D-10
   Rolling Off and Reusing Data and Index Segments  D-12
   Adding a New Segment  D-14
   Using an Offline Load Operation  D-16
   Deleting the Oldest Data  D-17
   Reusing the Segments  D-19

Index
About This Document

Purpose

This administrator’s guide provides information needed to create and maintain Red Brick® Warehouse databases. Information in this guide applies to Red Brick Warehouse on all supported hardware platforms and UNIX-based operating systems. For platform- or operating system-specific information, refer to the appropriate Red Brick Installation and Configuration Guide or to the documentation that accompanied the hardware and operating system. For administrative information about Red Brick Warehouse for Windows NT, refer to the Warehouse Administrator’s Guide for Windows NT Platforms. For information about the Table Management Utility (TMU), refer to the Table Management Utility Reference Guide.

Audience

The intended users of this guide are system and database administrators and managers who are responsible for system administration, the administration of a Red Brick Warehouse system, and the design and implementation of warehouse databases. In this guide, these users are collectively designated as the warehouse administrator.

Knowledge of the operating system, basic system administration procedures, and relational databases is assumed.
Organization

This guide is organized into 10 chapters and 4 appendixes:

Chapters 1 through 3 contain introductory and background information that you should read before designing a schema, creating a database, or loading data.

Chapter 4 contains information that helps you plan the physical implementation and space requirements of your warehouse database.

Chapter 5 contains information to guide you through the creation of the database, the user tables, the indexes, and other database objects.

Chapter 6 contains information that helps you determine how to best provide security for your data warehouse, using a combination of object privileges either with the system roles (DBA, RESOURCE, and CONNECT) or with the custom roles and password security available with the Enterprise Control and Coordination option.

Chapter 7 contains information about monitoring database activity using the features included with the Enterprise Control and Coordination option.

Chapter 8 describes the maintenance activities that go on throughout the life of a database such as setting configuration parameters, monitoring and managing growth, reorganizing tables and indexes as they change, monitoring and controlling warehouse processes, and deleting databases and database objects when they are no longer needed.

Chapter 9 provides guidance for tuning a database for performance using system and warehouse parameters.

Chapter 10 is dedicated to parallel query performance.

Appendix A contains an example that illustrates how to build a database, using the Aroma sample database.

Appendix B describes the warehouse configuration file and lists the configuration parameters.

Appendix C describes the warehouse system tables and the Dynamic Statistic Tables.

Appendix D contains an example that illustrates the use of segments with time-cyclic data.
Related Documentation

The standard documentation set for Red Brick Warehouse includes the following documents:

**Installation and Configuration Guide**
- Installation and configuration information, as well as platform-specific material, about Red Brick Warehouse and related products. Customized for either UNIX-based or Windows NT systems.

**Warehouse Administrator’s Guide**
- Description of warehouse architecture, supported schemas, and other concepts relevant to warehouse databases. Procedural information for designing and implementing a warehouse database, maintaining a database, and tuning a database for performance. Includes a description of the system tables and the configuration file (`rbw.config`). Customized for UNIX-based or Windows NT systems.

**Table Management Utility Reference Guide**
- Description of the Table Management Utility, including all activities related to loading and maintaining data. Also includes information about data replication and the `rb_cm` copy management utility.

**SQL Reference Guide**
- Complete language reference for the Red Brick Systems SQL implementation and RISQL® extensions for warehouse databases.

**SQL Self-Study Guide**
- Example-based review of SQL and introduction to the RISQL extensions, the macro facility, and Aroma, the sample database.

**RISQL Entry Tool and RISQL Reporter User’s Guide**
- 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.

**Messages and Codes Reference Guide**
- Complete listing of all informational, warning, and error messages generated by warehouse products, including probable causes and recommended responses. Also includes event log messages that are written to the log files.

**Release Notes**
- Information pertinent to the current release that was unavailable when the documents were printed.
In addition to the standard documentation set, the following documents are included for specific sites:

**Red Brick Vista User’s Guide**
Description of the Red Brick Vista™ aggregate navigation and advice system, including procedures for rewriting queries and getting advice on the best set of aggregate tables and views to create. Includes detailed examples of queries whose performance can be dramatically increased by using aggregate navigation.

**SQL-BackTrack for Red Brick Warehouse User’s Guide**
Complete guide to SQL-BackTrack™ for Red Brick Warehouse, a command-line interface for backing up and recovering warehouse databases. Includes procedures for defining backup configuration files, performing online and checkpoint backups, and recovering the database to a consistent state.

**Client Connector Pack Installation Guide**
Procedures to install and configure the Red Brick ODBC Driver, the RISQL Entry Tool, and the RISQL Reporter on client systems. Included for those sites that purchase the Client Connector Pack.

**ODBC Connectivity Guide**
Information about ODBC conformance levels as well as instructions on compiling and linking an ODBC application using the Red Brick ODBClib SDK.

**Red Brick Data Mine User’s Guide**
Description of the data mining process, and procedural information for using Red Brick Data Mine’s SQL-based interface to find hidden or unpredictable relationships among the data in a data set. Included for those sites that purchase the Red Brick Data Mine™ option.

**Red Brick Data Mine Builder™ User’s Guide**
Description of the data mining process, and procedural information for performing data mining using Red Brick’s GUI-based product in a Microsoft Windows environment.

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

**Online Documentation**

The English version of the Red Brick Warehouse documentation is also available in Adobe Acrobat format (PDF) on a separate CD-ROM.
Conventions

Throughout Red Brick Systems technical publications, the following notation and syntax conventions are used:

- Computer input and output, including commands, code, and examples, appear in Courier.
- Information that you enter or that is being emphasized in an example appears in Courier bold to help you distinguish it from other text.
- Filenames, system-level commands, and variables appear in Palatino italic or Courier italic, depending on the context.
- Document titles always appear in Palatino italic.
- Names of database tables and columns are capitalized (Sales table, Dollars column). Names of system tables and columns are in all uppercase (RBW_INDEXES table, TNAME column).

Syntax Notation

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

<table>
<thead>
<tr>
<th>Command Element</th>
<th>Example</th>
<th>Convention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values and parameters</td>
<td><code>table_name</code></td>
<td>Items that you replace with an appropriate name, value, or expression are in italic type style.</td>
</tr>
<tr>
<td>Optional items</td>
<td>[ ]</td>
<td>Optional items are enclosed by square brackets. Do not type the brackets.</td>
</tr>
<tr>
<td>Choices</td>
<td>ONE</td>
<td>TWO</td>
</tr>
<tr>
<td>Required choices</td>
<td>{ONE</td>
<td>TWO}</td>
</tr>
<tr>
<td>Default values</td>
<td>ONE</td>
<td>TWO</td>
</tr>
<tr>
<td>Repeating items</td>
<td>name, ...</td>
<td>Items that can be repeated are followed by a comma and an ellipsis. Separate the items with commas.</td>
</tr>
<tr>
<td>Language elements</td>
<td>() , ; .</td>
<td>Parentheses, commas, semicolons, and periods are language elements. Use them exactly as shown.</td>
</tr>
</tbody>
</table>
### Syntax Diagrams

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

<table>
<thead>
<tr>
<th>Component</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶️ ▶️</td>
<td>Statement begins.</td>
</tr>
<tr>
<td>▶️ ▶️</td>
<td>Statement syntax continues on next line. Syntax elements other than complete statements end with this symbol.</td>
</tr>
<tr>
<td>▶️ ▶️</td>
<td>Statement continues from previous line. Syntax elements other than complete statements begin with this symbol.</td>
</tr>
<tr>
<td>▶️ ▶️</td>
<td>Statement ends.</td>
</tr>
<tr>
<td>SELECT</td>
<td>Required item in statement.</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>Optional item.</td>
</tr>
<tr>
<td>DBA TO</td>
<td>Required item with choice. One and only one item must be present.</td>
</tr>
<tr>
<td>CONNECT TO</td>
<td>Required item with choice. One and only one item must be present.</td>
</tr>
<tr>
<td>SELECT ON</td>
<td>Required item with choice. One and only one item must be present.</td>
</tr>
<tr>
<td>ASC, DESC</td>
<td>Optional item with choice. If a default value exists, it is printed in <strong>bold</strong>.</td>
</tr>
<tr>
<td>ASC, DESC</td>
<td>Optional items. Several items are allowed; a comma must precede each repetition.</td>
</tr>
</tbody>
</table>
The syntax elements shown above are combined to form a diagram as follows:

```
GRANT ALL PRIVILEGES ON table_name TO db_username,
    role_name, PUBLIC
```

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

```
SELECT select_list from_clause where_clause
    group_by_clause having_clause when_clause
```

The point-of-reference diagram is then followed by an expanded diagram of the shaded portion—in this case, the `select_list`:

```
ALL DISTINCT * expression AS c_alias
    table_name.*
```
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. You can write a keyword in upper- or lowercase, 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

Metavariables serve as placeholders for identifiers and names in the syntax diagrams and examples. A metavariable can be replaced by an arbitrary name, identifier, or literal, depending on the context. Metavariables are also used to represent complex syntax elements that are expanded in additional syntax diagrams. When a metavariable appears in a syntax diagram, an example, or text, it is shown in lowercase italic.

The following syntax diagram uses metavariables to illustrate the general form of a simple SELECT statement:

```
SELECT — column_name — FROM — table_name —
```

When you write a SELECT statement of this form, you replace the metavariables `column_name` and `table_name` with the name of a specific column and table.
Customer Support

Please review the following information before contacting the Customer Support Center at Red Brick Systems.

Support Solutions Warehouse

The Support Solutions Warehouse is the Customer Support Center’s external web site, an online resource that registered Red Brick customers can use to:

• Submit new cases.
• Read release notes.
• Find answers to frequently asked questions (FAQs).
• Search the Problems and Solutions database.

To use the Support Solutions Warehouse, point your web browser to the following URL and enter your registered username and password:

http://www.redbrick.com/RBCustomer/index.htm

If you do not have a registered username and password, contact the Customer Support Center by telephone, fax, or e-mail.

General and Technical Questions

If you have general sales-related questions or technical questions about Red Brick products or services, contact Red Brick Systems as follows:

Telephone
General Questions (408) 399-3200 or 1 (800) 777-2585
Technical Questions (408) 399-7100 or 1 (800) 727-1866

FAX
General Questions (408) 399-3277
Technical Questions (408) 399-3297

Internet e-mail
General Questions info@redbrick.com
Technical Questions support@redbrick.com

World Wide Web www.redbrick.com
**Existing Cases**

If you want to inquire about the status of an existing case, please have the case number ready. The case number will always be given to you by the support engineer who logs the case or first contacts you. This number is used to keep track of all the activities performed during the resolution of each problem.

**New Cases**

If you want to log a new case, please have the following information ready:

- Red Brick Warehouse version
- Platform and operating-system version
- Error messages returned by Red Brick Warehouse or the operating system
- Concise description of the problem, including any commands or operations performed prior to the occurrence of the error message
- List of Red Brick Warehouse and/or operating-system configuration changes made prior to the occurrence of the error message

If you think the problem concerns client-server connectivity, please have the following additional information ready:

- Name and version of the client tool in use
- Version of Red Brick ODBC Driver in use (if applicable)
- Name and version of client network and/or TCP/IP stack in use
- Error messages returned by the client application
- Warehouse and client locale specifications
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 removing columns or functions, or restating WHERE, ORDER BY, or GROUP BY clauses until you can isolate the part of the statement causing the problem.
- For TMU load problems, verify the datatype mapping between the source file and the target table to ensure compatibility. Try loading a small test set of data to determine whether the problem concerns volume or data format.
- For connectivity problems, verify that the network is up and running by issuing the `rbping` command from the client to the host. If possible, try another client tool to see if the same problem arises.

Documentation Questions and Comments

If you have questions or comments about the Red Brick Warehouse documentation, please contact the Technical Publications Department at Red Brick Systems as follows:

- **Telephone**
  - +1 408 399 3200
  - +1 800 727 1866 (USA only)
- **Internet e-mail**
  - docs@redbrick.com
About This Document
Customer Support
Red Brick® Warehouse is a relational database management system (RDBMS) designed for data warehouse, data mart, and online analytical processing (OLAP) applications. Compared to online transaction processing (OLTP) or “universal” database products, Red Brick Warehouse delivers higher query-processing and data-loading performance, greater ease of administration, and a richer set of specialized features for applications that range from a few gigabytes to well over a terabyte, and from a few users to thousands of users.

Red Brick Warehouse can scale from the workgroup to the enterprise, is built for an open client/server environment using industry-standard open database connectivity (ODBC), and is accessed using industry-standard SQL. The server’s RISQL® extensions simplify analyses that require ranks, ratios, and other commonly used business calculations, while the Red Brick Vista™, STARjoin™, STARIndex™, TARGETjoin™ and TARGETIndex™ technologies provide unparalleled ad hoc query and analysis performance against 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:

- Red Brick Technology
- Warehouse Components
- Warehouse Processes
- Warehouse Administration Overview
- Sample Database
- Warehouse Limits
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.

Queries can be expressed in standard SQL and with Red Brick Systems RISQL extensions. The RISQL extensions provide decision-support functions including ranks, ratios, moving averages, and cumulative totals. Red Brick Warehouse also provides a macro facility that simplifies creation of reusable generalized queries.

Red Brick Warehouse meets these requirements through:

- Dimensional segmentation, which allows a table’s data and indexes 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, Symmetric Multi-Processor (SMP), and Massively Parallel Processing (MPP) machines. Parallel processing can be used in query processing, multi-user relation scans, index building, and data loading.
- The RISQL extensions to SQL, which include functions for rank, moving sum, moving average, cumulative total, n-tile analysis, and market share. These functions are specifically designed to take advantage of Red Brick Warehouse technology to provide answers to complex queries submitted to decision-support databases.
- The Table Management Utility (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 option allows aggregation of new data with existing data during the load process. A parallel version provides faster data loading on SMP and MPP machines.
- Support for time-cyclic data with date datatypes 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.
Warehouse Components

Red Brick Warehouse consists of three main software components: the warehouse server, the Table Management Utility, and the RISQL Entry Tool. Red Brick Systems also offers as options the RISQL Reporter, a report generator; and the Red Brick ODBC Driver, a gateway to Red Brick Warehouse for ODBC-compliant query tools provided by other vendors in a client/server computing environment.

The following figure illustrates the components of Red Brick Warehouse.
**Red Brick Warehouse Server**

The Red Brick Warehouse server accepts SQL commands and delivers the results to client applications such as the RISQL Entry Tool, the RISQL Reporter, or any other client tool connected to Red Brick Warehouse through the Red Brick ODBC Driver.

The server functions are performed by the processes described in “Warehouse Processes” on page 1-6.

**Table Management Utility**

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

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

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

- Upgrade databases to run with a newer version of 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 database backup/restore operations (Incremental Backup option).
- Perform data aggregations as the data is loaded (Auto Aggregate option).
- Use multiple processors for load operations (Parallel TMU option).

For information about the TMU, refer to Appendix A, “Example: Building a Database,” and to the *Table Management Utility Reference Guide*. 
RISQL Entry Tool and RISQL Reporter

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

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

Red Brick ODBC Driver

The Red Brick ODBC Driver is a software program that allows a wide variety of ODBC-compliant database applications to work with Red Brick Warehouse. The Red Brick ODBC Driver acts as a translator between the ODBC interface, used by popular front-end query applications, and the Red Brick Warehouse ODBC-native Application Programmer Interface (API). For more information about the Red Brick ODBC Driver, refer to the Client Connector Pack Installation Guide.

Red Brick Warehouse Administrator Tool

Red Brick Warehouse Administrator is a graphical segmentation assistant and administration tool. This client tool runs on Windows 95- or Windows NT-based computers and can be used to perform many database administration tasks. For more information, refer to “Red Brick Warehouse Administrator Tool” on page 2-19.

Database

Database information is stored as system tables and user tables. System tables contain table and column descriptions used by Red Brick Warehouse components and processes. For a list of system tables, refer to Appendix C, “System Tables and Dynamic Statistic Tables.” User tables contain the data loaded into tables with the TMU or with INSERT statements; the data is retrieved by query statements.
Warehouse Processes

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

Interprocess Communication

The following figure illustrates the interprocess communication that occurs among the warehouse daemon, other daemons, warehouse server processes, and the Red Brick ODBC Driver. When a user accesses a warehouse database with a client tool, the client tool communicates through the Red Brick ODBC Driver with the warehouse daemon and server processes. When a user accesses the database with the RISQL Entry Tool or the RISQL Reporter, these tools communicate directly with the warehouse daemon and server processes.
Warehouse Daemon Process

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

Server Processes

The Red Brick Warehouse server uses a process-per-user architecture in which a new independent process named rbwsvr is created for each user session accessing the warehouse. An rbwsvr process accepts SQL statements from clients (for example, the RISQL Entry Tool or other client tools via the Red Brick ODBC Driver), checks the statement syntax, executes the command, and returns any output to the client. Each client session is serviced by its own warehouse server process; each server process exists until the client terminates the session.

Administration Daemon Process

The administration daemon process (rbwadmd) collects statistics for the dynamic statistic tables (DSTs) and performs the actions specified by ALTER SYSTEM commands. This process exists only on systems with the Enterprise Control and Coordination option. The rbwadmd process is started when the rbwapid process is started.

For more information about the rbwadmd process, refer to Chapter 7, “Managing Database Activity with the Enterprise Control and Coordination Option.”

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 warehouse daemon (rbwapid) starts. If the Enterprise Control and Coordination option is installed, the warehouse administrator can specify which events to log; otherwise the log daemon logs only a restricted set of events intended to help technical support diagnose problems.

For more information about the rbwlogd process, refer to Chapter 7, “Managing Database Activity with the Enterprise Control and Coordination Option.”
Warehouse Administration Overview

Administration of Red Brick Warehouse databases includes the following activities:

- Installation, which includes preparing the warehouse environment, installing the software, and setting up and maintaining warehouse directories, files, and the 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, 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 loading with the Table Management Utility (TMU) or Parallel TMU.
- 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

Red Brick Warehouse software is installed and verified from a CD-ROM device using a menu-driven script provided by Red Brick Systems. The warehouse 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 sample database.

The administrator also uses the installation script to activate any purchased options, using the individual license keys provided when the options are purchased. Subsequent installations of maintenance releases are also done from the installation script.
The warehouse administrator must create a specific administrator account to use for the installation and other administrative tasks. This account is referred to throughout Red Brick Warehouse publications as the redbrick user. The installation process ensures correct file access for all users.

For information about the installation procedure, refer to the Red Brick 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 warehouse 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 important technical concepts, refer to Chapter 2, “Key Concepts.” For information about schema design, refer to Chapter 3, “Schema Design.” For information on index strategies, physical storage, and size estimation, refer to Chapter 4, “Planning a Database Implementation.”

**Implementing the Database**

Databases are created and deleted by utilities provided with the warehouse; these utilities are executed as the administrative user (redbrick).

The rb_creator utility creates a new database. Database access is automatically granted to a predefined username and password; to ensure database security, the warehouse 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 information about `rb_creator` and creating segments, tables, and indexes, refer to Chapter 5, “Creating a Database.” For a complete description of the CREATE and GRANT statements, refer to the SQL Reference Guide.

The `rb_deleter` utility deletes database files; execution of this utility requires write permission for the database files and for the parent directory. For information about `rb_deleter`, refer to “Deleting a Database” on page 8-43.

For information about system tables, refer to Appendix C, “System Tables and Dynamic Statistic Tables.” For information about the RISQL Entry Tool, refer to the RISQL Entry Tool and RISQL Reporter User’s Guide.

**Providing User Access**

The warehouse administrator creates user accounts and sets access privileges using GRANT and REVOKE statements entered with the RISQL Entry Tool, RISQL Reporter, or other client tools that accept SQL input. Additional database security is available with the Enterprise Control and Coordination option, which includes role-based security and enhanced password protection.

For more information about user access and security, refer to Chapter 6, “Providing Database Access and Security.” For more information about the Enterprise Control and Coordination option, refer to Chapter 7, “Managing Database Activity with the Enterprise Control and Coordination Option.”

**Initialization Files**

Each time a user session is started to access a warehouse database, that session is initialized by warehouse 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.
Macros

Each user session is affected by macros defined for warehouse databases. The warehouse administrator and privileged users can define macros to simplify creation of reusable generalized queries. A macro can be:

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

For more information about SQL macros, refer to “Creating and Managing Macros” on page 5-17 and to the SQL Reference Guide.

Loading the Data

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

For information about the TMU and loading data, refer to Appendix A, “Example: Building a Database,” and to the Table Management Utility Reference Guide.

Maintaining the Database and Tuning for Performance

As a database is modified over time, the warehouse administrator must perform maintenance tasks that include:

- Monitoring query performance. As circumstances and the warehouse environment change, the warehouse 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 as needed.
- Monitoring database storage requirements and allocating additional space as needed to accommodate growing databases.
- Altering tables and segments to reflect changes in data or the database.
- Performing periodic backups to prevent unrecoverable data loss.

Additional tools for monitoring and controlling database activities and performance and for copying data from one database to another are available with the Enterprise Control and Coordination option.
For information about database maintenance and performance tuning, refer to Chapter 8, “Maintaining a Data Warehouse,” and Chapter 9, “Tuning a Warehouse for Performance.” For more information about the Enterprise Control and Coordination option, refer to Chapter 7, “Managing Database Activity with the Enterprise Control and Coordination Option.”

**Sample Database**

Aroma, a small database for analyzing sales at a chain of specialty coffee stores, is used for many examples in this document, the *SQL Reference Guide*, and the *SQL Self-Study Guide*. The Aroma database is installed during the warehouse installation.
Warehouse Limits

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

Caution: Do not use these limits to estimate database file sizes. For information on calculating space requirements for tables and indexes, refer to Chapter 4, “Planning a Database Implementation.”

- A database can contain a maximum of 32,767 tables.
- Each session can contain a maximum of 4,096 temporary tables.
- A database can contain a maximum of 61,439 segments.
- A segment can contain a maximum of 250 files, and a file can be no larger than 2 gigabytes. Therefore, a segment can contain a maximum of 500 gigabytes.
- A table can have a maximum of 7,280 columns.
- A table can have a maximum of 256 foreign keys.
- A table can contain a maximum of \(2^{48}\) rows.
- A row in a table can contain at most \(8185 - \text{ceil}((\text{ncols} + 2) / 8)\) bytes of data, where \(\text{ncols}\) is the number of columns in the table. (The ceil function rounds any non-integer up to the nearest integer.)
- The maximum length of the text for a macro definition is 1,024 bytes.
- The maximum length of a character column is 1,024 bytes.
- The maximum length of a string literal is 1,024 bytes.
- The maximum length of a database identifier is 128 bytes.
- The maximum length of a database password is 128 characters; however, the RISQL Entry Tool and RISQL Reporter limit passwords supplied in response to the prompt to a maximum of 8 characters.
- The maximum length of a query is 64K (65,536 bytes).

Red Brick Warehouse features or options that are options for Warehouse for Workgroups are described as such throughout this document. The following additional limits apply to databases used with Red Brick Warehouse for Workgroups:

- A warehouse can contain only two databases.
- A table cannot contain more than 5 gigabytes of data.
- The maximum number of named user IDs (1, 5, 10, 20, or 30) is limited by the type of license purchased.
- Only one segment can be associated with each table or index.
In order to perform warehouse administration functions effectively, you should understand the key concepts of Red Brick Warehouse. This chapter is divided into the following sections:

- Database Loading
- Parallel Processing
- Physical Implementation of Databases
- Database Directories and Files
- Configuration and Initialization
- Warehouse Locale
- File Ownership and Permissions
- Database Authorizations and Privileges
- Table and Database Locks
- Referential Integrity
**Database Loading**

Data is loaded in a bulk process using the Table Management Utility (TMU), which indexes data as it is loaded. 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 warehouse database is as fast as or faster than loading databases for other RDBMSs.

The TMU supports the following types of data:

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

Data can be loaded from TAR format or ANSI standard label tapes, disk files, or a pipe from another system program.

For those 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.

The Parallel Table Management Utility (PTMU) option improves load performance on systems with multiple processors.

User-defined indexes can be defined either before or after data is loaded.
Parallel Processing

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

In addition to taking advantage of multiple processors, 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 Red Brick Warehouse dimensional segmentation, parallel processes can perform disk accesses simultaneously, improving performance significantly. When the needed data resides on a single disk, Red Brick SuperScan technology allows each process to take advantage of data read by the other processes to reduce the disk access delays.
**Physical Implementation of Databases**

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

Each database is a self-contained entity containing system tables and 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 and views, which provide logical organizations for table data, and macros, which define frequently executed operations on that data.

**Indexes and Retrieval Strategies**

User tables are supported by a variety of indexes. Indexes are invisible to warehouse users but are critical for data integrity and performance. Red Brick Warehouse automatically provides those indexes required for data integrity; additional indexes can be created by the warehouse administrator 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 multi-table joins. An administrator can create one or more STAR indexes for better performance. When multiple STAR indexes exist, Red Brick Warehouse automatically selects the best STAR index to use for executing each query by applying a cost model.

- A B-TREE index is used to improve performance and ensure data integrity. Red Brick Warehouse 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/space trade-off for the warehouse administrator.

- A TARGET index is used to improve 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 multi-table 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**

There are two types of segments: named segments 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 those tables and indexes that the administrator does not place in a named segment or segments.

*Named segments* offer the warehouse administrator extensive control over disk space allocation, file size and placement, and database growth, but require more effort from the administrator to plan, create, and manage. Using named segments, a warehouse administrator can partition a table horizontally, distributing the data across multiple segments. For example, sales data might be partitioned by time periods, with each time period residing in a separate segment. Control over the individual segments simplifies maintenance tasks and provides better access.
As an example of how segmented storage is used, consider a database that tracks the previous two years’ worth of sales data. The data is distributed across segments so that each segment contains one month of data. Because individual segments can be added, loaded, or dropped independently of the rest of the database, each month the administrator adds a segment containing the loaded, indexed data for the current month and drops the segment containing data for the oldest month. An administrator can make these updates without taking the database offline, and the space in the old segment can be used for data for the next month.

*Default segments* require no specific management and are, for all practical purposes, invisible to both the administrator and users. A table or index in a default segment cannot span multiple segments or files; each must reside entirely within a single PSU, or file. The PSUs for a default segment are placed in the database directory (or in a default directory specified by the user). For small, static tables or databases, default segments are often satisfactory. If circumstances change and the additional control provided by named segments is desirable, these default segments can be altered and manipulated in the same manner as named segments.

**Implementation**

Segmented storage is implemented as follows:

- A segment contains one or more physical storage units (PSUs).
- A PSU can belong at most to one segment.
- A segment can contain either row data for a single table or index data for a single index, but not both.
- A table’s row data and its indexes 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:

**Key Concepts**

**Physical Implementation of Databases**

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

<table>
<thead>
<tr>
<th>Named Segments</th>
<th>Default Segments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row Data</td>
<td>Row Data</td>
</tr>
<tr>
<td>STAR Index</td>
<td>STAR Index</td>
</tr>
<tr>
<td>Primary Index</td>
<td>Primary Index</td>
</tr>
<tr>
<td>TARGET Index</td>
<td>TARGET Index</td>
</tr>
</tbody>
</table>

**Note:** In a Warehouse for Workgroups database, row data and indexes must each reside in a single segment.

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, then the table or index is created in a default segment consisting of one PSU.

**PSU Size and Growth**

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

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

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

**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 avoids “hot spots” but limits the use of offline operations because the location of data is not known. If hashing is used, the entire row is hashed.
A primary index can be segmented based on the key values of the first column of the index, either explicitly specifying ranges of values for each segment or specifying that the index is to be segmented by the same values as the data. (The first column is the column named first in the PRIMARY KEY clause.)

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

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

<table>
<thead>
<tr>
<th>Contents of Segments</th>
<th>Contents Distributed (Segmented) by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row data</td>
<td>Data value of segmenting column or hashing (keywords: SEGMENT BY VALUES OF, SEGMENT BY HASHING)</td>
</tr>
<tr>
<td>STAR indexes</td>
<td>The row IDs of the referenced table containing the segmented column of the STAR index (keywords: none or SEGMENT BY REFERENCES OF)</td>
</tr>
<tr>
<td>Primary index</td>
<td>Data values of first (leading) column in the primary index (keywords: SEGMENT BY VALUES OF)</td>
</tr>
<tr>
<td>TARGET indexes</td>
<td>Values of the indexed column (keywords: SEGMENT BY VALUES OF)</td>
</tr>
<tr>
<td>B-TREE indexes</td>
<td>Values of the first (leading) indexed column (keywords: SEGMENT BY VALUES OF)</td>
</tr>
</tbody>
</table>

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

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

---

**Online and Offline Segments**

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

A segment can be taken offline only when it is one of multiple segments associated with a table or index. When a table or index resides in a single segment, that segment cannot be taken offline. Consequently, default segments, which by definition contain an entire table or index, cannot be taken offline.
The administrator can take a segment offline to load or update it with new data, to restore it in case of media failure or other data loss, or to detach it and remove it from the table forever. While that segment is offline, users can still have access to the partially available table.

**Partial Availability of Tables and Indexes**

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

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

Query behavior on a table that is partially available because of offline index segments is also controlled by a SET command that determines whether to consider all indexes or only fully available indexes in selecting a query-processing strategy. For more information about controlling query behavior with partially available tables, refer to “Query Behavior on Partially Available Tables” on page 9-17 and “Use of Partially Available Indexes” on page 9-18.

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 includes 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. If you have installed the Red Brick Vista option, 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.

The Red Brick Vista option 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 the Red Brick Vista option, refer to the Red Brick Vista User’s 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 containing 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 filesystem.

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 filesystem).

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 tablespace directories and files is controlled by the Red Brick Warehouse server.

Logical Database Names

Users specify the database they wish to access by specifying a logical database name, which is automatically translated by the warehouse server into 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 file path /warehouse/mktg/newdb:

```sql
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.

```
/warehouse/mktg/newdb
    — RB_DEFAULT_IDX
    — RB_DEFAULT_INDEXES
    — RB_DEFAULT_LOCKS
    — RB_DEFAULT_SEGMENTS
    — RB_DEFAULT_TABLES
    — RB_DEFAULT_LOADINFO
    — dfltseg1_psu1
    — dfltseg2_psu1
    — dfltseg3_psu1
    — …
    — dfltseg8_psu1
```

Database system files created by rb_creator utility

Created by TMU LOAD operation

Default segments for tables and indexes created by CREATE TABLE statements

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 (file), you can add more PSUs as needed to hold the data.

**Example**

Assume that the same database is created in the newdb directory (path /warehouse/mktg/newdb) but segments are created to hold the table Fact1 and its indexes. Data in this table is segmented by time period, putting 1995 data in one segment and 1996 data in another segment. The primary key index is created and placed in a third segment.

```
CREATE SEGMENT fact_95
    STORAGE '/disk1/fact_95/95_dat1'
    MAXSIZE 1000 initsize 100 extendsize 100,
    STORAGE '/disk1/fact_95/95_dat2'
    MAXSIZE 1000 initsize 100 extendsize 100;
```

```
CREATE SEGMENT fact_96
    STORAGE '/disk2/fact_96/96_dat1'
    MAXSIZE 1000 initsize 100 extendsize 100,
    STORAGE '/disk2/fact_96/96_dat2'
    MAXSIZE 1000 initsize 100 extendsize 100;
```
Key Concepts
Database Directories and Files

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_95, fact_96)
    SEGMENT BY VALUES OF (perkey)
    RANGES (MIN:'1995-12-31', '1996-01-01':MAX)
    PRIMARY INDEX IN fact_pi;

Note: The directories /disk1/fact_95, /disk1/fact_96, and disk2/fact_pi must exist before the CREATE SEGMENT commands can be issued; however, it is not necessary that the segment name and directory name be identical.
The following figure illustrates how the database files are stored using named segments.

```
warehouse/mktg/newdb
  └── RB_DEFAULT_IDX
      └── RB_DEFAULT_INDEXES
          └── Database system files
              created by rb_creator utility
          └── RB_DEFAULT_LOCKS
          └── RB_DEFAULT_SEGMENTS
          └── RB_DEFAULT_TABLES
          └── RB_DEFAULT_LOADINFO
              └── Default segments for tables and indexes
                  created by CREATE TABLE statements
              └── created by TM U LOAD operation
                └── dfltseg1_psu1
                └── dfltseg2_psu1
                └── dfltseg3_psu1
                └── ...dfltseg8_psu1
/disk1
  └── fact_95
      └── 95_dat1
          └── 95_dat2
/disk2
  └── fact_96
      └── 96_dat1
          └── 96_dat2
      └── fact_star
          └── 96_star1
              └── 96_star2
```

**Segment Names**

To determine the segment name of a table or index, look at the RBW_SEGMENTS system table:

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

<table>
<thead>
<tr>
<th>NAME</th>
<th>TNAME</th>
<th>INAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBW_SYSTEM</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_1</td>
<td>PERIOD</td>
<td>NULL</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_2</td>
<td>PERIOD</td>
<td>PERIOD_PK_IDX</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_3</td>
<td>PRODUCT</td>
<td>NULL</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_4</td>
<td>PRODUCT</td>
<td>PRODUCT_PK_IDX</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_5</td>
<td>MARKET</td>
<td>NULL</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_6</td>
<td>MARKET</td>
<td>MARKET_PK_IDX</td>
</tr>
<tr>
<td>FACT_95</td>
<td>FACT1</td>
<td>NULL</td>
</tr>
<tr>
<td>FACT_96</td>
<td>FACT1</td>
<td>NULL</td>
</tr>
<tr>
<td>FACT_STAR</td>
<td>FACT1</td>
<td>FACT1_STAR_IDX</td>
</tr>
</tbody>
</table>

**Key Concepts**

Database Directories and Files
Key Concepts
Configuration and Initialization

Configuration and Initialization

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

Configuration File

Warehouse configuration information is contained in the rbw.config file, which resides in the warehouse directory. This file is generated during the installation process and contains configuration and performance-tuning parameters used by the warehouse daemons, warehouse servers, and TMU processes; it also contains license keys for options and logical database name definitions. It can be edited with a text editor.

Initialization Files

Red Brick Warehouse uses three types of initialization files: .rbwrc files for server initialization; .rbretrc files for RISQL Entry Tool and RISQL Reporter initialization; and .odbc.ini files which contain definitions for ODBC applications.

.rbwrc Files

The .rbwrc files can contain any non-query SQL commands, such as temporary macro definitions and INSERT and SET commands. There can be up to four .rbwrc files that affect a user. These files are processed in the following order:

$RB_CONFIG/.rbwrc
A global warehouse file in the directory containing the rbw.config file.

$RB_PATH/.rbwrc
A database-specific file in each database directory.

$HOME/.rbwrc
A user-specific file in each user’s home directory for those users who access warehouse databases with RISQL Entry Tool or RISQL Reporter running on the same machine as Red Brick Warehouse.

$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
where DBUSERNAME is the database username, not the UNIX user account name. The extension must be in uppercase.
The processing order allows database settings to override warehouse settings, and user settings to override both database and warehouse settings.

The warehouse and database initialization files are managed by the warehouse administrator. These files generally contain global or database-specific temporary macros, access-control statements, values that limit the sizes of warehouse 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 warehouse operation.

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 server process (rbwsvr) is started, these log files are deleted to prevent their unlimited growth.

Note: File permissions must be set so that the server process, which runs as the redbrick user, can read the .rbwrc files and write to the .rbwerr file.

.rbretc Files

The .rbretc files contain SET commands for the RISQL Entry Tool and RISQL Reporter. For example, a command that specifies the editor of choice; or a command that specifies display widths for system table columns to neatly format the screen displays. Two .rbretc files, processed in the following order, can affect a user:

$RB_CONFIG/.rbretc
  A warehouse-wide file in the directory containing the rbw.config file; this file affects all databases in the warehouse.

$HOME/.rbretc
  A user-specific file in each user’s home directory; this file affects all sessions for that user.

The processing order allows user settings to override warehouse settings. Command-line settings override settings in the .rbretc files but only exist for the current session.
Key Concepts
Configuration and Initialization

.odbc.ini Files

The .odbc.ini files contain ODBC Data Source Name (DSN) definitions. This file is a user-specific initialization file in each user’s home directory containing DSN definitions for ODBC applications. The DSNs are used by the RISQL Entry Tool, RISQL Reporter, or by other client applications connecting to Red Brick Warehouse through the UNIX ODBC Drivers. A DSN is a logical name that defines a server, a logical database name, a database username, and optionally a database password.

A template file named odbc.ini is created in the redbrick directory upon installation. From this template file, each user can make a customized ODBC initialization file located in $HOME/.odbc.ini.

SET Commands

SET commands are used to configure and customize the warehouse 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 the .rbretrc initialization files
• In a TMU control file
• From inside the RISQL Entry Tool or RISQL Reporter
• From a client tool sending SQL commands

Environment Variables

The warehouse uses the following environment variables:

**RB_CONFIG**
Pathname to the directory containing the rbw.config file, which is usually the directory containing the bin directory of warehouse executable files.

**RB_DSN**
Data source name (DSN), which specifies a server, logical database name, database user, and optionally a password.

**RB_HOST**
Logical name used to identify the warehouse daemon process.

**RB_PATH**
Logical database name, as defined in rbw.config file by a pathname to a database directory. Used to determine which warehouse database to access.
**Red Brick Warehouse Administrator Tool**

The Red Brick Warehouse Administrator tool is a GUI-based segmentation assistant and administration tool. A standalone application that runs under the Windows 95 and Windows NT operating systems, Red Brick Warehouse Administrator gives direct access to multiple databases while providing a GUI for segmentation tasks and basic database administration.

Red Brick Warehouse Administrator provides easy-to-use, Wizard-based capability to:

- Create, alter, and drop users, roles, macros, tables, indexes, segments, synonyms, and views.
- Perform more detailed segment-related tasks such as examining and verifying PSUs, attaching and detaching segments, and defining PSU attributes.
- Grant privileges and authorizations to, and revoke them from, 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.
- Set the database backup option (SQL-BackTrack for Red Brick Warehouse option only).
- Determine the actual and maximum size of tables and indexes in your database.

Additionally, Red Brick Warehouse Administrator allows you to graphically view:

- The relationships between referenced and referencing tables in the database.
- The structure of your filesystems.
- Information about the data, index, unattached segments in your database, as well as the backup segment (SQL-BackTrack for Red Brick Warehouse option only).
- 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.
Red Brick Warehouse Administrator also provides an interactive SQL window, to allow you to enter SQL commands manually, and a Show DDL window where you can view the SQL commands that were used to create a selected database object.
Warehouse Locale

This section defines the term locale, describes how the warehouse locale is specified during installation, explains how to override the warehouse locale for 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. The following sections are included:

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

Components of a Locale

The unique combination of a language and a location is known as a locale. The data warehouse locale, defined during installation, is the administrator’s mechanism for controlling the run-time behavior of warehouse databases. A locale specification consists of four components: language, territory, character set, and collation sequence. For example:

Japanese_Japan.MS932@Binary

where

- Japanese = language
- Japan = territory
- MS932 = character set
- Binary = collation sequence

The following sections explain each locale component in detail. For detailed information about specifying a warehouse locale, refer to page 2-24.

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 commands, are not translated.
Key Concepts
Warehouse Locale

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, the use of these languages differs according to the location. (In some cases, a single territory applies to more than one country in a region.)

The formatting rules for various non-character datatypes 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 in some parts of the world, numbers are not separated into groups of 1,000.

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.

Character Set

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

In the United States, a 7-bit encoding known as ASCII is commonly used; however, this encoding is inadequate for international use because all 128 encodings are assigned and they do not include all the characters necessary to represent languages other than English. Therefore, Red Brick 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.
Key Concepts
Warehouse Locale

- 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 character sets and supported conversions between character sets, refer to the Installation and Configuration Guide.

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. There are two main types of character comparisons: 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, note the following limitations:

- 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 128–255 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 may also need to be context sensitive (where a character sorts may depend on what characters precede or follow it). Because of the table lookups and context sensitivity, linguistic comparisons are relatively slow compared to binary comparisons.

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

Defining the Warehouse Locale

During the installation of the Red Brick Warehouse software, a locale specification is requested for the data warehouse. The locale supplied during installation is stored as the NLS_LOCALE LOCALE parameter in the rbw.config file. If no locale is supplied, the default value of this parameter is used: English_UnitedStates.US-ASCII@Binary

This locale specification applies to the whole Red Brick Warehouse installation, regardless of the number of databases that will be 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 Red Brick products that regulates their run-time 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 “non-client” products, including the Red Brick Warehouse server, the TMU, and the miscellaneous utility programs, use the locale defined in the rbw.config file to determine their run-time behavior. For example, if the language is set to Japanese, all server and TMU error messages are displayed in Japanese. Similarly, if the locale specifies a multibyte character set such as MS932, database object names and character strings can contain multibyte characters.

**Note:** For full instructions about defining the warehouse locale during installation and for a list of supported locales, refer to the *Installation and Configuration Guide*.

**System Table References to Locales**

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

**Non-Translated Text**

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

- Installation scripts
- Contents of the rbw.config file; all text in this file must be in ASCII characters.
- Contents of system tables (except for object names)
- Log messages
- Output of the EXPLAIN statement (except for object names)
- Output of all Red Brick Warehouse utility programs (rb_creator, 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 warehouse locale. Also note that messages for the service-related utilities (tblchk, ixvalid) are in English and data formatting is not localized.
Overriding the Warehouse Locale

In the client/server environment, the operating locale of the client can differ from the warehouse locale defined in the `rbw.config` file; therefore, the locale defined for a client tool can differ from the locale of the warehouse server. However, only differences in language and character 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, while changing the character set allows the client and server to use different character sets, as long as a successful conversion can be made. For a list of supported character sets for each language, refer to the *Installation and Configuration Guide*.

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

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

Specifying a Locale for a Client Tool

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

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

The RISQL Entry Tool and RISQL Reporter are Red Brick client applications. An operating locale can be set for each of these clients by setting the `RB_NLS_LOCALE` environment variable. If no client locale is specified, the client uses the warehouse locale, as specified by the current NLS_LOCALE LOCALE entry in the `rbw.config` file; this entry should not be modified.

*Note:* Non–Red Brick client tools must provide their own means of specifying a locale but they can still use the RB_NLS_LOCALE environment variable to control the locale of the Red Brick-generated messages and to control character set conversions.
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.

For example, from the UNIX C shell prompt:

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

where

- `German` = language
- `Austria` = territory
- `Latin1` = character set
- `Default` = collation sequence

It is not necessary to specify all four parts of a locale; however, it is strongly recommended that if the locale is not fully specified, the language should be one of the specified components. Otherwise, the unspecified components might default to incompatible values.

Note the following rules regarding 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 will be as follows:

  `Japanese_Japan.JapanEUC@Binary`

  For a list of default components for each language, refer to the *Installation and Configuration Guide*.

- If only the territory is specified, the language defaults to English, the character set set to US-ASCII, and the sort to Binary. For example, if the locale is set to:`_Japan`

  the complete, but *impractical*, locale specification will be as follows:

  `English_Japan.US-ASCII@Binary`

- Similarly, if only the character set is specified, the language defaults to English, the territory defaults to UnitedStates, and the sort component defaults to Binary.
**Key Concepts**

*Warehouse Locale*

- Finally, if only the sort component is specified, the language defaults to English, the territory defaults to UnitedStates, and the character set defaults to US-ASCII.

**Note:** It is not necessary to 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(s) is required—such as the underscore character ( _ ) in the previous territory example.

**Ensuring Client/Server Compatibility**

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

- A Japanese user working on a PC might set the client locale to:
  
  Japanese_Japan.MS932

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

- A French-speaking user might connect to a database with a German locale:
  
  French_France.Latin1 (client locale)
  German_Germany.Latin1 (warehouse locale)

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

- An English-speaking user might attempt to use a Japanese database:
  
  English_UnitedStates.US-ASCII (client locale)
  Japanese_Japan.JapanEUC (warehouse locale)

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

The following sections explain how certain client/server incompatibilities can be avoided and corrected.
Character Set Conversions

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

Caution: Red Brick Warehouse provides no recovery mechanism when data loss or data corruption occurs because of incompatible character 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 warehouse 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:

- Server error messages: RB.LLTTT.MB
- Server log messages: RL.LLTTT.MB
- Client messages: RC.LLTTT.MB

where LL identifies the language and TTT identifies the territory.

All letters in message filenames are in uppercase. For example, the 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 (rbwadm) and log (rbwlogd) daemons 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 server and is therefore in the language of the warehouse locale.

For more information about the administration and log daemons, refer to Chapter 7, “Managing Database Activity with the Enterprise Control and Coordination Option.”
**File Ownership and Permissions**

In the warehouse installation procedure, you create an operating-system user account. The default name for this account is `redbrick`, although you can choose any name. Throughout the Red Brick Warehouse documentation, this user account is referred to as `redbrick`, and this user ID is used for all warehouse 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 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 warehouse administrator uses a DBA account in each database to grant users database access. This account exists on each new database with username system and password manager and has membership in the DBA system role. From this account, the warehouse administrator uses GRANT statements to set up user access to each database.

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

- **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; it can be 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.

With the Enterprise Control and Coordination option, the warehouse administrator can use role-based security features to control database access with a finer degree of detail. These features allow you to break down the tasks of the RESOURCE and DBA system roles into separate tasks authorizations and to create custom roles.

For more information about system roles, privileges, and role-based security, refer to Chapter 6, “Providing Database Access and Security,” and to the SQL Reference Guide.
**Table and Database Locks**

To preserve consistency within a database, operations that modify data must be allowed to complete without interruption, blocking out other read and write operations until the modification (write operation) is complete. Locking is performed automatically by those operations that require it for database consistency. However, when uninterrupted execution and table access for multiple operations is more convenient, the warehouse administrator can lock a table manually with a `LOCK` statement. For example, if you wanted to delete multiple rows of data from a table and wanted to complete the operations without other users accessing and perhaps locking the table between your operations, you could lock the table, complete your delete operations, and then unlock the table. For a description of the `LOCK` command, refer to “Locking Tables and Databases” on page 8-2.

Red Brick Warehouse supports the following types of locks:

- **Read locks**, which lock a table for read access only, allowing multiple readers but no writers. This access is often called “shared access.”
- **Write locks**, which prevent the specified table from being read or modified by anyone else until the write operation on the table is completed. This access is often called “exclusive access.”
- **Delete locks** for delete operations. A delete lock locks the named table for write access to perform the delete operation. It also locks all tables that reference the named table for read or write access as needed to maintain referential integrity. This type of lock cascades through the complete family of referencing tables, as described on page 2-35.
- **Database locks** applied with the `LOCK DATABASE` command. These locks prevent the entire database—all tables including system tables—from being read or modified by anyone else until the `UNLOCK DATABASE` command 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.

Individual segments cannot be explicitly locked. If a table is locked, access to all online segments is controlled by the table lock of their owning table. 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’s being dropped or altered during the offline operation.
Locks applied manually with the LOCK command ensure the necessary table or database access throughout multiple modification operations: If a table is locked with the LOCK command, no sessions except the one that obtained the lock can access the table until it is unlocked with the UNLOCK command (or until the locking session terminates).

A write lock cannot be placed on a table or database while it is already locked for a read or write operation. A WAIT/NO WAIT option allows a user to specify whether a request by the server or TMU for a write lock on an already locked database or table should either fail and return immediately or wait until any existing locks are dropped and then lock the object.

A user can explicitly lock only one object (table or database) at a time with the LOCK command; the first object must be unlocked before a second one can be locked. The system, however, can implicitly lock as many objects as necessary to process a query or command.
**Referential Integrity**

Referential integrity is the relational property that each foreign key value in a table exists as a primary key in the referenced table. Red Brick 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/PRIMARY KEY clauses in the CREATE TABLE statement and are automatically maintained both during load, update, and insert operations into 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, then 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 has been edited to correct data conversion or content errors.

An alternative to discarding those 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 a table’s immediate and complete family is used:

- A table’s immediate family is the set of all tables that reference that table with a FOREIGN KEY reference clause.
- A table’s complete family is its 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 a table’s complete family overrides any cascaded delete conditions.

**Note:** 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.

**Example**

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

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Immediate Family</th>
<th>Complete Family</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fact1</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Fact2</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Market</td>
<td>Fact2</td>
<td>Fact2</td>
</tr>
<tr>
<td>Product</td>
<td>Fact1, Fact2</td>
<td>Fact1, Fact2</td>
</tr>
<tr>
<td>Period</td>
<td>Fact2</td>
<td>Fact2</td>
</tr>
<tr>
<td>Brand</td>
<td>Product</td>
<td>Product, Fact1, Fact2</td>
</tr>
<tr>
<td>Monthname</td>
<td>Period</td>
<td>Period, Fact2</td>
</tr>
</tbody>
</table>

A delete operation (and the FOR DELETE option to the LOCK table command) 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 those tables’ immediate family 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:

```sql
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:

```sql
prodkey char(5) not null,
...
foreign key prodkey references product (prodkey)
on 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:

<table>
<thead>
<tr>
<th>From Brand, delete row containing:</th>
<th>Rows deleted from:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Brand</td>
</tr>
<tr>
<td>x</td>
<td>Yes</td>
</tr>
<tr>
<td>y</td>
<td>Yes</td>
</tr>
<tr>
<td>z</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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.

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:

<table>
<thead>
<tr>
<th>From Monthname, delete row containing:</th>
<th>Rows deleted from:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Monthname</td>
<td>Period</td>
<td>Fact</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>Yes</td>
<td>No rows deleted from Period</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>z</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

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:

```sql
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:

```sql
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:

```sql
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.

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.
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 work in this case:

<table>
<thead>
<tr>
<th>From Brand, delete row containing:</th>
<th>Rows deleted from:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Brand</td>
</tr>
<tr>
<td>x</td>
<td>Yes</td>
</tr>
<tr>
<td>y</td>
<td>No</td>
</tr>
<tr>
<td>z</td>
<td>No</td>
</tr>
</tbody>
</table>

**Note:** This example does not say anything about 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 determines the behavior in those families. For example, if the Market table is referenced by Fact2 with a cascaded delete, then 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; and Product, the only table in its immediate family, is locked for read access. The Fact1 and Fact2 tables are not locked. Note that even though y is not referenced by Fact2, it is not deleted from Brand, Product, or Fact1.

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

---

**Key Concepts**

**Referential Integrity**
Key Concepts
Referential Integrity
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
- Common Schema Examples
- Example: Salad Dressing Database
- Example: Reservation System Database
- Example: Investment Database
- Example: Health Insurance Database
**Transaction Processing Versus Decision Support**

Although 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 very 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, cyclic, and are interpreted at the time of the query.
- **No redundancy:** Redundant and aggregate data is avoided in order to ensure data integrity and reduce lockout contention.
- **Relational integrity:** The reliability of the data depends on transaction integrity; relational integrity checks are too slow and would 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, monitoring warehouse 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).
- **Mostly static**: Most changes to the database occur in a controlled manner when data is loaded at regular intervals.
- **Join paths**: Join paths are simple, non-cyclic, and are 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 SQL SELECT statements that make comparisons or require sequential processing; these queries might reference many thousands, millions, or even billions of records in a database.
- **Large result sets**: Extensive and frequent browsing must be supported.
- **Recoverability**: Regular backups, or snapshots, of the static database ensure against data loss.
Red Brick 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.

Two types of star schemas, a simple star schema and a multi-star schema, are described in the following sections.
A star schema is composed 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 reflect 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 as 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 sales person 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 company’s products 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 where you routinely query very large amounts of data. Red Brick Warehouse supports all schema designs; however, star schemas tend to perform the best in decision-support applications.

Terminology

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

Simple Star Schemas

Any table that references or is referenced by another table must have a primary key, which is a column or group of columns whose contents uniquely identify each row. In a simple star schema, the primary key for the fact table is composed of one or more foreign keys; a foreign key is a column or group of columns in one table whose values are defined by the primary key in another table. In Red Brick 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. Non-key columns in a fact table are referred to as data columns; in a dimension table, as attributes.

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. Notice that 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.

Notice the many-to-one relationships 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 the previous year’s sales:

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:
Multi-Column Foreign Key

Another way to define a many-to-many relationship is to have a dimension table with a multi-column 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 is modeled by defining a multi-column foreign key in the Sales_Current table that references the Product table, as in the following figure:

Note that in the preceding figure, the Product_id and Group_id columns are the 2-column primary key of the Product table and are a 2-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:

Note that in the preceding figure, the Market table, since 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.
**Multi-Star 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 multi-star schema.

In a multi-star schema, the fact table has both a set of foreign keys, which reference dimension tables, and a primary key, which is composed 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 multi-star schema; this fact distinguishes a multi-star schema from a single-star schema.

The following figure illustrates the relationship of the fact and dimension tables within a multi-star 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. Note that the primary key can be composed of any combination of foreign key and other columns in a multi-star schema.
The following figure illustrates a retail sales database designed as a multi-star 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.

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.

Notice that in a multi-star schema, unlike a simple star schema, the same value for the concatenated foreign key in the fact table can occur in multiple rows, so that 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(s); 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, if you are licensed for the Red Brick Vista option, you can create precomputed views so that queries are automatically rewritten to access the appropriate aggregate table. For information on precomputed views and automatic query rewriting, refer to the 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 be considered in designing a warehouse 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 data warehouse. 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 as the basis for 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 very frequently, it probably should be measured as a fact, not stored as a dimension.
• What are the facts to 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, although 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 warehouse 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" can be stored along with the data (a column to indicate whether the data is a daily, monthly, or yearly amount). 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/or 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.
**Common Schema Examples**

The following figures illustrate some common schema examples. Tables named *Fact* or *Factx* represent fact (referencing) tables. The other tables represent dimension (referenced) tables.

**Note:** The following figures apply to both single-star and multi-star schemas.

A schema can consist of a single dimension table:

![Single Dimension Table](image)

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

![One Fact Table and One Dimension Table](image)

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

![One Fact Table and Several Dimension Tables](image)
A schema can be a multiple star schema, with a family of fact tables that share some, but not necessarily all, dimension tables:

---

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

---

A schema can be a star schema with a fact table that contains multiple foreign keys that reference single dimension table(s):
**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 4-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.

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.
Browsing the Dimension Tables

To write effective queries, a user needs to be familiar with the contents of a database. 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, a user can enter the following:

```
select market_desc from market
```

which 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 down 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” into 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 then 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 table(s).
Using Table Attributes

Non-key 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:

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

Determining Which Attributes to Include

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 could 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. Note, however, that 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 in some cases hierarchy might be required, 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.
Example: Reservation System Database

This example illustrates a multi-star 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.

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.
**Example: Investment Database**

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

Although separate tables for aggregated data require more disk space, they can sometimes provide performance benefits. Furthermore, the aggregates are usually relatively small compared with the unaggregated fact table. In some situations, this can be a worthwhile space-for-performance tradeoff. Consider, for example, 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% 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 may 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. Remember, however, that every situation is different, and aggregated data is not the right approach in every case.

**Note:** If aggregated and non-aggregated data are stored in the same table, each query must specify the level of aggregation as a constraint.

If you are licensed for the Red Brick Vista option, you can use the aggregate tables, along with the some special logical definitions (views and hierarchies), to create precomputed views. The precomputed views allow the system to automatically rewrite queries submitted against a detail table to access an aggregate table. For detailed information about this option, refer to the *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.
Example: 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 and their policies and claims against those policies.
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 multi-star 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 bold type, 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.
This chapter describes the planning that must be done before you implement a Red Brick Warehouse database and includes the following sections:

- Organizing Data into Databases
- Determining When to Create Additional Indexes
- Planning For TARGETjoin Processing
- Planning Disk Storage Organization
- Estimating the Size of User Tables and Indexes
- Example: Calculating Table, Index, and System Table Sizes
- Planning for Temporary Space Requirements
- Planning for Segmented Storage
- Considerations for Growing Tables
Organizing Data into Databases

In planning for a new Red Brick Warehouse installation, the warehouse 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, so 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.”

You must also determine whether to create any indexes in addition to those created automatically. An index on a column constrained in a query improves query performance, but it requires storage space and it must be updated when data is loaded; you must weigh these considerations. In general, if space is available and load performance is not an issue, you should index any column that will be constrained in queries.

Also consider how the tables will change over time; for example, determine whether and at what rate the tables will grow as new records are added.
Determining When to Create Additional Indexes

In a Red Brick Warehouse database, a B-TREE index is automatically created on the primary key of a table during table creation. To improve query performance, you should create additional indexes. The improved query performance must be balanced against the additional storage space for each index and the additional time to build or update it during each load procedure.

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 may result in referential integrity violations unless you build either another B-TREE index on the primary key or a STAR index.

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

Creating additional indexes can have a large impact on query performance. For information about how Red Brick Warehouse chooses the indexes to use in a query and the algorithms used to join tables, refer to “Understanding Red Brick Query Processing” on page 9-25.

Consider the following guidelines about creating additional indexes:

• If your schema design is a star schema where you have a central referencing (fact) table with primary key-foreign key references to one or more referenced (dimension) tables, create one or more STAR indexes. If queries against a fact table will constrain only the trailing foreign key column of an existing STAR index or if queries will constrain the trailing columns more tightly than the leading columns: Create an additional STAR index naming the more tightly constrained columns as the leading columns in the index key to improve query performance. STAR indexes are usually more useful when they cover more columns. A single-column STAR index is not a very useful index.

• If your schema design is a star schema, if you have many referenced (dimension) tables, and if the STAR indexes you have created are not giving optimal performance for all of your queries, you can create TARGET indexes on the foreign key columns of the referencing (fact) table to enable TARGETjoin processing. For more information about TARGETjoin processing, refer to “Planning For TARGETjoin Processing” on page 4-11 and to Chapter 9, “Tuning a Warehouse for Performance.”
Planning a Database Implementation
Determining When to Create Additional Indexes

- Queries against multiple fact tables (fact-to-fact table joins) place two specific requirements on STAR indexes:
  - The fact tables must each have at least one foreign key reference to at least one common dimension table.
  - All the shared foreign keys must appear in the same relative order in each STAR index.
Create additional STAR indexes to satisfy these requirements in cases where queries against multiple fact tables are needed. If these conditions are not satisfied, the query will proceed using a different join algorithm and may not perform as well.
- Define B-TREE indexes on each UNIQUE column to enforce the uniqueness constraint.
- Queries against a table will 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% 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/NA.)
  - Create a TARGET index on any columns that contains a large number of duplicate values.

Note: 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.

Each of these cases is illustrated in the examples that follow.

Note: In B-TREE and STAR indexes, the order of the columns named in the CREATE INDEX statement determines the order of the column values in the index key.

**STAR Indexes**

Red Brick Warehouse uses STAR indexes to greatly improve performance on queries involving tables that have foreign keys that are the primary keys of another table. A STAR index uses STARjoin technology, a proprietary method of joining tables with a primary key/foreign key relationship in the schema design. When you have such schemas, known as star schemas, you must create one or more STAR indexes to take advantage of this technology.
If you have multiple STAR indexes, the order of the foreign key constraints in a query determines which index is used. Red Brick Warehouse uses the STAR index built on columns with the closest match to the query constraints.

**Example**

This example illustrates a case where 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:

```sql
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_pkcl primary key (pk, fk1, fk2)  
) ;
```

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:

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

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

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

This example illustrates a case where a STAR index is used to perform an efficient join between two fact tables (fact-to-fact table joins). In this case, assume you built a STAR index that includes all the foreign keys, in the order they are specified in the CREATE TABLE statement, for each table. This means 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))
```

Note that 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. Since the STAR indexes you already built are in the foreign key order specified in the CREATE TABLE statement, they do not have their keys in the same order, which is a requirement for joins between fact tables.

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

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

The requirements listed on page 4-4 for STAR indexes used for fact-to-fact joins are met: All foreign keys shared by the fact tables are present in a STAR index for each table (the STAR index on all the foreign keys for Fact1 and Star2 on Fact2); and the shared keys are in the same relative order (Dimension1, Dimension2, Dimension3). If the analyst’s queries will constrain on only the shared dimension tables, the index Star2 is sufficient.
If, however, the analyst wants to constrain table Fact2 on its non-shared foreign key Fk5, which references the table Dimensiony, that column must also be included in a STAR index:

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

Note that 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 there is no STAR index or TARGET index available, but there is a B-TREE index available, then the B-TREE index is used, causing a nested loops join.

**Note:** If you create a multi-column B-TREE index, all columns included in the index must be declared NOT NULL or the index creation will terminate with an error.

**Example**

This example illustrates a case where a non-primary key column in a table is constrained; an additional index will improve performance.

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

```sql
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 Personality table’s primary key). If queries constrain any of the other columns in the Product table, creating B-TREE indexes on those columns will improve performance. If, for example, you have queries that constrain on the Distributor column, then you can create a B-TREE index as follows:

```sql
create index on product (distributor) ;
```
TARGET Indexes

There are two different applications for TARGET indexes:

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

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

For information about using TARGET indexes to enable TARGETjoin query processing, refer to “Planning For TARGETjoin Processing” on page 4-11 and to Chapter 9, “Tuning a Warehouse for Performance.”

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 will be weakly selective; they will 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 Warehouse selects the appropriate storage method, or “representation,” for the TARGET index information.
If you do not specify the domain size when you create the TARGET index, Red Brick Warehouse dynamically selects the storage representation for each distinct key value (each different value in the column) of the TARGET index column. The storage representation could be different for different key values, depending on the number of occurrences of each distinct key value in the indexed column. This method works well in situations where the data is non-uniform and in situations where you are not sure of the data in the column you are indexing.

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

Knowing Your Data

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

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

Example

Suppose you have a 10,000,000 row Customer table that has 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.

```sql
create target index customer_region_idx on customer (region) domain size medium;
```
Planning a Database Implementation
Determining When to Create Additional Indexes

**No Indexes**

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

If the join is not an equijoin and there are no indexes 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` command. For more information on the `SET CROSS JOIN` command, refer to the *SQL Reference Guide*. 
## Planning For TARGETjoin Processing

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

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

For more information about using TARGETjoin query processing to improve performance, refer to “TARGETjoin Query Processing” on page 9-40 of Chapter 9, “Tuning a Warehouse for Performance.”

### STARjoin Versus TARGETjoin

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

When you submit a query to Red Brick Warehouse, the server first generates a plan for execution. This plan is based on the performance of each join method for the indexes available. The server chooses between STARjoin, TARGETjoin, B-TREE 1-1 match, hybrid hash join, and cross join. The actual query path is chosen at query execution time. The Red Brick Warehouse server chooses a TARGETjoin query plan when a good STARjoin query plan is not available and when the TARGETjoin operation will perform better than a table scan.
Remember that TARGETjoin processing is not a replacement for STARjoin processing; they are complementary technologies. All other things being equal, with a good STARindex (an index that contains the columns constrained in the query as leading keys in the STARindex), STARjoin processing is a faster join method than TARGETjoin processing. However, with a STARindex where the leading keys are not constrained in the query, TARGETjoin processing is likely to provide better performance.

**Administration Considerations for TARGETjoin Processing**

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

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

**Cost Versus Performance**

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

There are two types of costs involved in creating the indexes to enable TARGETjoin processing:
- Increased disk space usage to store the indexes
- Increased load times to update the indexes
Load Operations

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

- Faster load times because the TARGET and B-TREE indexes do not need to be built for the load operation to complete.
- Increased database availability because the database is available for querying while the TARGET and B-TREE indexes are building. Queries do not use TARGETjoin processing until the appropriate indexes become available. When the indexes become available, Red Brick Warehouse automatically considers them when compiling queries. The net gain of this is a shorter downtime for the database for load operations, although the foreign key indexes are not immediately availability.

Consider the following two cases regarding loading data:

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

Load Operations that Roll Segments Off and On

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

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

- When there are relatively small INSERT, UPDATE, DELETE, or LOAD operations.
- With databases that are designed for growth by allocating segments for future loads when the tables are created.
- When the database updates are infrequent.

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

Large Dimension Tables

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

There are two potential difficulties with queries that use TARGETjoin processing to join large dimension tables to a fact table:

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

The first problem is purely a resource problem. If you can afford the space and the time needed to create the index, then this is not a problem. The index size and the time to create it will vary depending on your schema. Use the `dbsize` tool to estimate the index size.
The second problem occurs when you join the large dimension table to the fact table with a query that has a large number of qualifying rows (weakly selective) on the dimension table. For example, suppose you have a customer table with 1,000,000 rows. Suppose also that you want to know something about all of your customers who are male, and that 75% of your customers are male. That means that 750,000 rows of the dimension table qualify for this join. In this case, TARGETjoin processing will not perform well. (STARjoin processing performs well on these weakly selective constraints on large dimensions when there is a STARindex in which one of the trailing keys is the foreign key corresponding to the large dimension table.)

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

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

If you have only one customer with a last name that begins with “X,” then this query will perform very well using TARGETjoin processing.

**Multi-Column Foreign Keys**

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

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

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

For information about the `CREATE INDEX` command, refer to the SQL Reference Guide.

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

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

- ROWS_PER_FETCH_TASK
- ROWS_PER_JOIN_TASK
- FORCE_FETCH_TASKS
- FORCE_JOIN_TASKS

For information about setting these parameters and about parallel queries, refer to Chapter 10, “Tuning a Warehouse for Parallel Query Processing.”

TARGET Index DOMAIN Clause

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

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

Use the dbsize utility to estimate the size of TARGET indexes. The size will change depending on the DOMAIN clause specification of SMALL, MEDIUM, or LARGE. To estimate the range of sizes of a TARGET index where the DOMAIN clause is not specified (the “hybrid” representation), use dbsize to calculate the size of indexes of all three DOMAIN values. The upper and lower bounds from these results define the range of sizes.
The following table shows the different types of TARGET indexes, the representation that each type uses to store data, and recommendations on when to use each type.

<table>
<thead>
<tr>
<th>DOMAIN Size</th>
<th>Representation</th>
<th>When to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not specified</td>
<td>Hybrid; changes with data</td>
<td>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.</td>
</tr>
<tr>
<td>SMALL</td>
<td>Bitmap</td>
<td>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). Tends to offer the best performance, but can also use the most disk space when the number of distinct values is high.</td>
</tr>
<tr>
<td>MEDIUM</td>
<td>Compressed row list</td>
<td>Use when the number of distinct values is between 100 and 1,000 or when the expected number of rows in the fact table for each distinct foreign key value in the fact table is medium (between 1,000 and 100,000).</td>
</tr>
<tr>
<td>LARGE</td>
<td>Uncompressed row list</td>
<td>Use when the number of distinct values is greater than 1,000 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 1,000).</td>
</tr>
</tbody>
</table>

When you consider the number of unique values, the important number is the number of unique values in the fact table. Note that this might be a smaller number than the number of unique values in the dimension table. For example, a Day table might contain ten years worth of days, or 3,650 unique values, but the database (and therefore the fact table) might only contain one year of data, or 365 unique days. In this case, the best choice for a TARGETindex on the foreign key column that references the Day table is a DOMAIN MEDIUM TARGETindex.

For the complete syntax for creating TARGET indexes, refer to the CREATE INDEX section of the SQL Reference Guide.
Planning Disk Storage Organization

Before creating a new database, carefully estimate the disk space requirements for the database and its contents so that you can decide whether to organize the database contents entirely into default segments or to use named segments for some or all tables or indexes.

To decide on your disk space requirements, you need to estimate:

- The size of each table and index in the database.
- The total disk space requirements for the database, which is the sum of the space required for each permanent and temporary user table, each automatic and optional index, and the system tables.

You should also consider the expected growth patterns of tables.

In order to estimate space requirements and make space organization decisions, you need to have designed the database schema and identified 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 every table, the columns to be indexed and the types of indexes to be used. Consider automatic indexes (primary key B-TREE indexes) and optional indexes (additional B-TREE indexes, STAR indexes, and TARGET indexes).
- Anticipated growth patterns for all tables, including growth rate and maximum expected number of rows, within your planning horizon.

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

1. Use the `dbsize` utility to estimate the storage space for each user table to be included in the database.
2. Use the `dbsize` utility to estimate the storage space for each automatic and optional index to be included in the database.
3. Use the `dbsize` utility to estimate the size of the system tables for the database.
4. Determine how you want to use segmented storage, planning a strategy for distributing data across multiple disks if necessary. If any of your user tables are expected to grow (in terms of number of rows stored), review the discussion about growing tables on page 4-44.

**Note:** In Red Brick Warehouse for Workgroups, the warehouse is limited to two databases and no table can exceed 5 gigabytes of data.
Estimating the Size of User Tables and Indexes

To calculate the size of user tables and indexes, you must know the number of columns, the datatypes of the columns, and the expected number of rows in each table. From that information, you can use the database sizing tool included with Red Brick Warehouse to calculate the length of each row and the number of bytes required to store all the rows. This tool is named `dbsize` and is in the `redbrick_dir/util/service` directory. For information about this tool, refer to the README file, also in the `service` directory.

Example

Assume a table is created as follows:

```sql
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, located in the `redbrick_dir/util/service` directory.
2. Choose the option to estimate the size of user tables.
3. Respond to the prompts for the number of columns, the datatypes of the columns, the precision of the decimal datatype, and the total number of estimated rows for the table.
4. For the preceding CREATE TABLE statement, `dbsize` estimates the size of the table to be 4,156,872 kilobytes.
Estimating Index Sizes

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

- How fill factors determine how much space is 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.

Fill Factors

Each index has a fill factor associated with it. The 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 command.
- During incremental load operations when new nodes are created because the previous node reached the fill factor.

Note that 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% full before a full node splits to form two new 50%-full nodes. Therefore, some nodes might be fuller than specified by the fill factor. Each new node must then fill to 100% before it splits again.

Note: 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% and 75% 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% tells you the absolute minimum amount of space the index would take.
- A fill factor of 75% 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% 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% 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%, in which case that fill factor should be used to determine the practical maximum).

Fill Factor Specification for an Index

Use the following guidelines to choose a 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 very 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% of the data in the initial load, with a relatively small amount of data to be added later, specify a 95% fill factor. Conversely, if you are loading only 5% of the data initially before you load the remaining data for the production database, specify a 5% fill factor.

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

**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 will fail. 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.

Use the `dbsize` utility, located in the `redbrick_dir/util/service` directory, to determine the size of your STAR indexes.

To estimate the size of a STAR index, you need to know the following:

- For each foreign key column that participates in the STAR index: the estimated maximum number of rows and number of segments in the referenced (dimension) table (MAXROWS PER SEGMENT multiplied by MAXSEGMENTS).
- The estimated number of rows to be included in the referencing (fact) table.
- The fill factor for the index.
Planning a Database Implementation
Example: Calculating Table, Index, and System Table Sizes

**B-TREE Indexes**

Use the `dbsize` utility, located in the `redbrick_dir/util/service` directory, to determine the size of your B-TREE indexes.

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

- The number of columns to be indexed.
- The datatype 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**

Use the `dbsize` utility, located in the `redbrick_dir/util/service` directory, to determine the size of your TARGET indexes.

To estimate the size of a TARGET index, you need to know the following:

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

**Example: Calculating Table, Index, and System Table Sizes**

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

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

```sql
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));
```
Planning a Database Implementation
Example: Calculating Table, Index, and System Table Sizes

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

Table Market contains approximately 5,000 rows and is defined as follows:

```sql
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:

```sql
CREATE TABLE product (
    prodkey integer NOT NULL,
    category integer,
    PRIMARY KEY (prodkey));
```

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

**Fact1 Table and Its Indexes**

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

**Table Size: Fact1**

Use `dbsize` to calculate table size for the table Fact1.

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

**STAR Index: Fact1**

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

1. Run the `dbsize` utility.
2. Choose the option to estimate the size of indexes.
3. Choose the option to estimate the size of STAR indexes.
4. Enter the maximum number of rows for the Market and Product tables.
Planning a Database Implementation
Example: Calculating Table, Index, and System Table Sizes

5. Enter the estimated number of rows for the fact table, which is 53,000,000 in this example.
6. Enter the fill factor. Assume the data is ordered for the initial load but will be updated through incremental loads, so use a fill factor of 66%. Note that for a fill factor of 66%, you must enter .66 in \texttt{dbsize}.
7. For the CREATE TABLE statement for Fact1, \texttt{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 \texttt{dbsize} to calculate the B-TREE index size.

1. Run the \texttt{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 datatype for each column.
6. Enter the estimated number of rows for the table, which is 53,000,000 in this example.
7. Enter the fill factor. Assume the data is ordered for the initial load but will be updated through incremental loads, so use a fill factor of 66%. Note that for a fill factor of 66%, you must enter .66 in \texttt{dbsize}.
8. For the CREATE TABLE statement for Fact1, \texttt{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 \texttt{dbsize} to calculate table size for the table Market.

1. Run the \texttt{dbsize} utility.
2. Choose the option to estimate the size of user tables.
3. Respond to the prompts for the number of columns, the datatypes of the columns, and the total number of estimated rows for the table, which is 5,000 in this example.
4. For the CREATE TABLE statement for Market, \texttt{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.

\textbf{Index Size: Primary Key B-TREE Index}

1. Run the \texttt{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 datatype for the column, which is integer for this example.
6. Enter the estimated number of rows for the indexed table, which is 5,000 in this example.
7. Enter the fill factor. Assume the data is ordered and updated infrequently, so use a fill factor of 100%. Note that for a fill factor of 100%, you must enter 1.00 in \texttt{dbsize}.
8. For the CREATE TABLE statement for Market, \texttt{dbsize} estimates the size of the primary key B-TREE index to be 80 kilobytes.

\textbf{Index Size: TARGET Index on Mktname Column}

1. Run the \texttt{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 datatype 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 5,000 in this example.
8. Enter the estimated percentage of NULL rows for the Market table. For this example, assume no NULL values, and enter 0.0.
9. Enter the domain size, which is SMALL in this example. For a description of domain size for TARGET indexes, refer to “TARGET Indexes” on page 4-8.
10. Enter the number of segments in the table, which is 1 for this example.
Planning a Database Implementation
Example: Calculating Table, Index, and System Table Sizes

11. For the CREATE TABLE statement for Market, \textit{dbsize} estimates the size of the TARGET index to be 200 kilobytes.

\textbf{Product Table and Its Indexes}

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

\textbf{Table Size: Product}

Use \textit{dbsize} to calculate table size for the table Product.
1. Run the \textit{dbsize} utility.
2. Choose the option to estimate the size of user tables.
3. Respond to the prompts for the number of columns, the datatypes of the columns, and the total number of estimated rows for the table, which is 520,000 in this example.
4. For the CREATE TABLE statement for Product, \textit{dbsize} estimates the size of the table to be 4,584 kilobytes.

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

\textbf{Index Size: Primary Key B-TREE Index}

1. Run the \textit{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 datatype 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\%. Note that for a fill factor of 66\%, you must enter .66 in \textit{dbsize}.
8. For the CREATE TABLE statement for Product, \textit{dbsize} estimates the size of the primary key B-TREE index to be 9,312 kilobytes.
Index Size: B-TREE Index on Category Column

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 one (Category) in this example.
5. Enter the datatype 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%. Note that for a fill factor of 66%, you must 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 9,312 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 Red Brick databases, the total space required for the system tables is less than 1 megabyte.

Use *dbsize* to estimate the size of the system tables. You will need to know the following information in order to supply *dbsize* with the information for which it prompts you.

- 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.
Planning a Database Implementation
Example: Calculating Table, Index, and System Table Sizes

- 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, for this example, there are 9 segments.
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 multi-column 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, as shown in the following table.

<table>
<thead>
<tr>
<th>Database Object</th>
<th>Space Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fact1 table</td>
<td>4,038,104 KB</td>
</tr>
<tr>
<td>STAR index</td>
<td>946,448 KB</td>
</tr>
<tr>
<td>Primary B-TREE index</td>
<td>1,265,704 KB</td>
</tr>
<tr>
<td>Market Table</td>
<td>136 KB</td>
</tr>
<tr>
<td>Primary B-TREE index</td>
<td>80 KB</td>
</tr>
<tr>
<td>TARGET index</td>
<td>200 KB</td>
</tr>
<tr>
<td>Product table</td>
<td>4,584 KB</td>
</tr>
<tr>
<td>Primary B-TREE index</td>
<td>9,312 KB</td>
</tr>
<tr>
<td>B-TREE index on Category</td>
<td>9,312 KB</td>
</tr>
<tr>
<td>System tables</td>
<td>392 KB</td>
</tr>
<tr>
<td><strong>Total space</strong></td>
<td><strong>6,274,272 KB</strong></td>
</tr>
</tbody>
</table>
Planning for Temporary Space Requirements

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

This section describes temporary space requirements in terms of:

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

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

- Multiple databases: Each Red Brick Warehouse 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; you might, however, 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.
- Maximum data segment size: This operating system value, usually set as a kernel configuration parameter, determines how much memory a process can use. This memory space is used by Red Brick Warehouse as buffer cache for the TMU (specified by the TMU_BUFFERS parameters), staging arrays for index building (specified by the INDEX_TEMPSPACE_THRESHOLD parameter), and the remainder as general working space.
- Swap space: Even though a system is configured with sufficient swap space, out-of-memory errors can occur if insufficient swap space is available for the process stack and the data when a swap occurs.
For information about how temporary disk space is allocated and the temporary space parameters, refer to “Setting Temporary Space Parameters” on page 9-2.

**How Optimized Index-Building Operations Use Temporary Space**

Database administration operations that build an index 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 non-optimized 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.

**Note:** 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 require 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 warehouse 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 parameter(s) (or system temporary space if this parameter is not set). The shift from the sort phase to the merge phase occurs when there is no more input data 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 in setting the INDEX_TEMPSPACE parameters to reflect the available memory and disk space. Some guidelines for setting these parameters are provided in the following sections.

**Estimating Temporary Space Values for Index-Building Operations**

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

**DIRECTORY Locations**

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

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

For more information about how space is allocated among the various directories designated as temporary space directories, refer to “Setting Temporary Space Parameters” on page 9-2.
The threshold specifies how much memory is used before the data spills to temporary space on disk. The goal in selecting a threshold for INDEX_TEMPSPACE is to select a large value relative to the system on which Red Brick 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.

To estimate a reasonable threshold for both online and offline load operations and other index-building operations:

1. First determine the maximum program data space allowed on your machine. This value is usually set as part of the UNIX kernel configuration for your machine (sometimes called maxdsize). A typical value is 64 megabytes. If you do not know the value configured for your machine, consult your system administrator or system vendor.

2. Then select a value for INDEX_TEMPSPACE_THRESHOLD that is one quarter of the maximum data space size for your system. For example, if the maximum data space size on your machine is 64 megabytes, choose a value of 16 megabytes. (The value you enter is automatically rounded up to the nearest 8-kilobyte block.)

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

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

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

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

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

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

   When determining the number of indexes involved in a CREATE INDEX operation, include the number of indexes (excluding TARGET indexes of DOMAIN SMALL or DOMAIN MEDIUM) being created in one command (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:

   \[
   \text{total_space_required} \leq \text{MAXSPILLSIZE} \times \text{number_of_cycles}
   \]

   and

   \[
   \text{MAXSPILLSIZE} \leq \text{available_temp_space}
   \]

   where:

   - `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, multi-segment 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 will fail. 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 limits 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 has completed. 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, then the temporary space used by the load will be 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.

Use the following procedure to calculate the disk space requirements for an offline load operation.

1. For each index (including TARGET indexes), use the \texttt{dbsize} utility to estimate the size of each index.

2. Add the values calculated with \texttt{dbsize} for each index. This sum is the \textit{total\_space\_required} for the index-building operation.
3. Set MAXSPILLSIZE according to the following relationships:

\[ \text{total\_space\_required} \leq \text{MAXSPILLSIZE} \leq \text{available\_temp\_space} \]

where \text{available\_temp\_space} is 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 fact table to enable TARGETjoin processing), note the following temporary space requirements:

- 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:

\[
\text{Temporary Space} = (\text{keysize} + 11) \times \text{rows}
\]

where \text{keysize} is the width of the column (in bytes) being indexes and \text{rows} is the number of rows in the table. You do not necessarily need to allocate this much space for the index build operation, however. If the temporary space is exhausted, it triggers a sort-and-merge cycle from the temporary space into the actual index. The more sort-and-merge cycles that occur, the more time the index build operation takes.

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

\[(4 + 11) \times 1,000,000,000 = 15,000,000,000 \text{ bytes}, \text{ or approximately } 15 \text{ gigabytes (G)}.\]

In this case, allocating 15G of temporary space will ensure that the index will require only one sort-and-merge cycles, thus provided the fastest index build time. If you allocate 2G of temporary space, this will trigger seven sort-and-merge cycles, adding to the total time of building the index.
How Query Operations Use Temporary Space

The amount of memory allocated to a query is set with the QUERY_MEMORY_LIMIT parameter. 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 filesystems. The entire memory limit and maximum spill size values specified are allocated on a per-query basis, where a query is defined to include each of its subqueries, if any. For more information about how space is allocated among the various directories designated as temporary space directories, refer to “Setting Temporary Space Parameters” on page 9-2.

Note that 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 datatype 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 is the number that 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:
   - Determine the maximum program data space that a single process can use; 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 multi-user environment, however, lower values for QUERY_MEMORY_LIMIT often work better because they avoid excessive paging that occurs when multiple individual queries consume large amounts of physical memory.

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

**Estimating a MAXSPILLSIZE Value for Queries**

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

1. Determine the maximum amount of disk space available for query temporary operations and the number of users that will be running queries that spill. Multiple users running large queries will compete for space in the query temporary-space directories; you might want to consider providing separate temporary-space directories for individual users (with SET commands).

2. Choose a MAXSPILLSIZE value that limits the space used by each spill so that all users can be accommodated during peak loads. (The value you enter is automatically rounded up to the nearest 8-kilobyte block.)
Planning for Segmented Storage

After you have planned the database design and estimated the disk storage required for fully loaded and indexed user and system tables, you must decide whether you want to use named or default storage segments. Depending on the initial space requirements and growth patterns expected for the database, you might choose to assign all user tables and indexes to default segments. The creation and maintenance of default segments is simpler, but you do not have the flexibility offered by named segments. For more information about default and named segments, refer to “Segmented Storage” on page 2-5 and Chapter 8, “Maintaining a Data Warehouse.”

Default segments are created when a CREATE TABLE statement containing no explicit segment assignment is issued. 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 command.

**Note:** 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 commands. The files associated with a named segment reside at locations specified when the segment is created or altered.

Determining When to Use Default and Named Segments

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

- If the estimated size for a single user table or its associated automatic and optional indexes exceeds 2 gigabytes, that table must be placed in a named segment to spread database files across multiple filesystems.
- 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 filesystem where the database directory is to reside, then some tables or indexes must be placed in named segments with PSUs in other filesystems.
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 additional (named) segments, you can do so after specifying a segmenting column (with an ALTER TABLE command). 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 command to move the single PSU to another location or add additional PSUs to the segment.

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

- A named segment can contain only one table or one index (or nothing).
- A segment can contain up to 250 PSUs (files). You must decide whether to assign a few large files or more smaller files to the segment. A few large files might require reservation of entire disk partitions, whereas use of smaller files might allow fragments of disk space on various file systems to be effectively used for a single table or index. In making this decision, consider that managing and maintaining a few large files is generally easier than managing and maintaining many small files.
- You must also decide how to allow files to grow. For each PSU, you can specify an initial size, which is always reserved at the time the segment is created; a maximum size to which the PSU can grow; and an extend size, which is the increment by which the PSU will grow. Large initial sizes for PSUs require more space to be reserved initially, whereas smaller initial sizes are more likely to result in fragmentation.
- Note that Red Brick Warehouse uses disk space in 8-kilobyte blocks, so when a maximum file size is specified in a CREATE SEGMENT statement, the space allocated is rounded up to the nearest 8 kilobytes. The first file in a segment is always allocated a minimum of 2 blocks, or 16 kilobytes.
- If your system configuration permits, spread segment files across multiple disk drives and multiple I/O buses to improve access times.
- Segments (of multi-segmented tables and indexes) can be taken offline for loading or restore operations and continue to provide limited query access. If this feature is of interest at your site, carefully consider how to segment the data and the locations and sizes of PSUs needed to accommodate the data, keeping in mind the uses you intend to make of offline operations on segments.
• The amount of parallelism used for query processing is limited by the number of PSUs in which the data being queried is stored. For example, if a table contains only a single PSU, no parallel processing occurs for queries that require a relation scan of that table. Therefore, keep in mind the desired degree of parallelism when you define PSUs. For more information about PSUs and parallelism, refer to Chapter 10, “Tuning a Warehouse for Parallel Query Processing.”
Considerations for Growing Tables

In some applications, database tables grow over time. Red Brick Warehouse supports the addition of records to existing tables with the incremental load facility of the Table Management Utility 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.

With growing tables, you should 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.

Caution: 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 your 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 referencing 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 will fail.

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 Table Management Utility or dropped and recreated. 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 is used to expand the segment. If sufficient space is reserved in advance for growing dimension tables with accurate MAXSEGMENTS and MAXROWS PER SEGMENT parameters, the need to alter a segment and perform a REORG operation can be avoided or, if not avoided, anticipated and planned in advance.

Red Brick Systems strongly recommends that you specify MAXSEGMENTS and MAXROWS PER SEGMENT values for each table because you cannot create a STAR index that references a table unless these parameters have been defined.
Planning a Database Implementation
Considerations for Growing Tables
Creating a Database

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

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

Overview

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

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

To create a database:
1. Specify the locale when you install Red Brick Warehouse.
2. Create the database structure using the rb_creator utility.
3. 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 have created the database, you must provide user access, as described in Chapter 6, “Providing Database Access and Security,” and load data into the database with the Table Management Utility (TMU), as described in the Table Management Utility Reference Guide.
Creating a Database

Creating the Database Structure

To create the database structure, you use the `rb_creator` utility, which initializes the database and creates the system tables. Then you must define a logical name for the database in the `rbw.config` file; you should also change the default password for the database administrative account. Each of these tasks is described in this section.

Before you can create the database structure, you must install Red Brick Warehouse and specify a locale for all databases in your installation. For information about specifying a locale, refer to “Warehouse Locale” on page 2-21.

Initializing the Database

Initialize a new database with the `rb_creator` utility:

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 by entering:

   ```
   $ mkdir dirname
   ```

   where `dirname` is the pathname to the directory, such as `/disk1/database`. The `redbrick` user must have access permissions to create this directory, and the directory must be empty.

3. Verify that permissions are set correctly for this directory by entering:

   ```
   $ ls -l
   ```

   Permissions on the directory should be:

   ```
   redbrick: rwx (read, write, execute)
   group:     --- (none)
   other:     --- (none)
   ```

   If permissions do not match these settings, then 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 by entering:

```
$ rb_creator dirname
```

where `dirname` is the name of the database directory you just created.

If this directory does not designate an empty directory or if you (as the `redbrick` user) do not have sufficient write privileges, then `rb_creator` exits with an error message and does not create a new database. Otherwise, `rb_creator` initializes the database by creating the following database system files:

<table>
<thead>
<tr>
<th>Filename</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>RB_DEFAULT_IDX</td>
<td>System tables</td>
</tr>
<tr>
<td>RB_DEFAULT_LOCKS</td>
<td>System information for database and table locks</td>
</tr>
<tr>
<td>RB_DEFAULT_INDEXES</td>
<td>System information about indexes</td>
</tr>
<tr>
<td>RB_DEFAULT_SEGMENTS</td>
<td>System information about segments</td>
</tr>
<tr>
<td>RB_DEFAULT_TABLES</td>
<td>System information about tables</td>
</tr>
</tbody>
</table>

**Note:** 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:

```plaintext
# Logical database name mappings
#
DB AROMA /disk1/aroma/db
```
2. Add a line of the following form near the end of the file:

   DB database_name dirname

   where:

   database_name    Logical database name for new database;
                   case-insensitive. (Users can access using uppercase, 
                   lowercase, or a combination of both cases.)

   dirname          Full pathname to the database directory (case-sensitive).

   **Example**

   If you want to add a new database named NEW_DB, which is in the 
   /disk1/new_db directory, add the following line to the rbw.config file:

   DB NEW_DB /disk1/new_db

   If the file previously contained an entry only for the AROMA database, it would 
   now look like this:

   # Logical database name mappings
   #
   DB AROMA /disk1/aroma/db
   DB NEW_DB /disk1/new_db

**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. You should change the default password from manager to a secure 
password immediately after creating the new database.

To change the default password:

1. Invoke the RISQL Entry Tool by entering:

   
   $ risql -d logical_database_name system manager
   
   where logical_database_name refers to the new database.

2. Change the password by entering:

   RISQL> grant connect to system with new_password;
Creating a Database
Creating the Database Structure

A password can be any valid database identifier or string literal, as defined in the SQL Reference Guide. If your site has installed the Enterprise Control and Coordination option, there might be additional security restrictions imposed on passwords.

Note: Database passwords for accounts that use the RISQL Entry Tool should not exceed 8 characters. If database passwords exceed 8 characters, they cannot be typed at a password prompt and must be entered as a command-line option, which might compromise security.

Example

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

```
$ risql -d new_db system manager
(C)Copyright 1991-1998, Red Brick Systems, Inc., Los Gatos, CA, USA
All rights reserved
Version 5.1
RISQL> grant connect to system with mysecret;
RISQL> quit;
```

Creating the Database Objects

After the database has been created with `rb_creator`, you can create any needed segments, the user tables, additional indexes, synonyms, macros, and views. Each component of the table structure is created using a CREATE statement.

You can enter very simple table organizations interactively from the RISQL Entry Tool command line or with any tool that supports direct entry of SQL. However, the easiest way to enter longer, more complex CREATE statements is to write them in a text file to use as an input script for the RISQL Entry Tool or RSIQL Reporter.

For information about using script files, refer to the RISQL Entry Tool and RSIQL Reporter User’s Guide.

Create database objects in the following order:

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, as described on page 5-8.

2. Write CREATE TABLE statements for the dimension (referenced) tables, as described on page 5-9. 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, as described on page 5-9.

4. Write CREATE INDEX statements for any STAR indexes, additional B-TREE indexes, or TARGET indexes that you have selected, as described on page 5-12.

   **Note:** If you have written the CREATE statements in a script file, then 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.

6. Write any desired CREATE MACRO statements to simplify repetitive query components or to share procedures.

For a complete description of the SQL syntax, refer to the SQL Reference Guide.
Creating Segments

The CREATE SEGMENT command creates a storage segment consisting of one or more PSUs that together will contain a table or index. If you plan to use named segments, then you must define them before you create the tables and indexes. Segments can be modified (altered) as needed with an ALTER SEGMENT statement. For information about modifying segments, refer to “Altering Segments” on page 8-11.

Remember the following rules about segments:

- 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 default location for default segments is the database directory. If you have multiple databases and want to specify a different default directory for one or more of the databases, use a SET command to define these locations rather than an rbw.config file entry for DEFAULT DATA SEGMENT or DEFAULT INDEX SEGMENT.
- Named segments for all tables and their primary key indexes are assigned with the CREATE TABLE statement. Named segments for all optional indexes are assigned with the CREATE INDEX statement.
- Any table can reside in multiple segments, with data distributed by the data values or by a hashing algorithm.
- Each index on a table can reside in multiple segments. For primary indexes, the index entries can be distributed among segments in the same way as the indexed data (SEGMENT LIKE DATA) or by ranges that are independent of the data distribution. For STAR indexes, the index entries can be distributed sequentially among segments or distributed by references to internal storage locations (SEGMENT BY REFERENCES OF).
- Segments can span multiple PSUs.
- If you are using Red Brick Warehouse for Workgroups, tables and indexes cannot be split across segments.

Disk space is first allocated to a segment based on the INITSIZE of the first PSU in the segment (according to PSU sequence ID in the RBW_SEGMENTS table). Additional space is then allocated as data is stored in the PSU. Segments with large INITSIZE values take longer to create (because the INITSIZE space is being allocated) but are faster to load than segments with smaller INITSIZE values.
Creating Tables

Each user table is defined with a name; a description of the columns; a primary key definition; and optional foreign key definitions, referential integrity action, segment identifiers, and the values for MAXSEGMENTS and MAXROWS PER SEGMENT (for tables that will participate in STAR indexes).

Tables can be modified as needed with an ALTER TABLE statement. For information about modifying tables, refer to “Altering Tables” on page 8-24.

Remember the following rules about 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 Parameters

Red Brick Systems 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, then the STAR index creation will fail.

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 the section “STAR Indexes” on page 4-23 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.
Creating a Database
Creating Tables

**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 multi-column foreign key references, constraint names are required if the multi-column foreign key is referenced in a STAR index. Although constraint names are optional for single-column foreign keys and for primary keys, having meaningful constraint names can make your CREATE TABLE statements more understandable to other people. Constraint names are identified in the RBW_RELATIONSHIPS and RBW_CONSTRAINTS system tables. If constraint names are not supplied, they are given default names.

For more information on the CONSTRAINT keyword of the CREATE TABLE statement, refer to the *SQL Reference Guide*.

**Maintaining Referential Integrity with ON DELETE**

To maintain referential integrity during delete operations, you must consider both table creation and delete operations on related tables. Referential integrity is the relational property that each foreign key value in a table exists as a primary key value in the referenced table.

The ON DELETE clause in the FOREIGN KEY clause of the CREATE TABLE statement specifies how referential integrity is maintained during delete operations. The ON DELETE clause has the following options:

- **CASCADE** If a to-be-deleted row is referenced by a row or rows in another table, then both that row and the referencing row(s) 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, then neither that row nor the referencing row(s) are deleted from either table; this type of delete is also called a restricted delete. Note, however, that a row that is not referenced by another row will be deleted. (Default if ON DELETE is omitted.)

**Note:** This setting is stored in the DELACTION column of the RBW_RELATIONSHIPS system table.
A delete operation cannot perform both cascaded and restricted deletes. A NO ACTION reference anywhere in a table’s complete family (as defined in “Delete Operations and Cascaded Deletes” on page 2-35) 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.

**Caution:** Although performance during delete operations is better when referential integrity is not checked, use OVERRIDE REFCHECK only in cases where you know deletions will not violate referential integrity or in cases where you are planning to ensure referential integrity by performing a REORG operation on the referenced table after the delete operation.

You can also change the ON DELETE action associated with the foreign key for all future delete operations by using the ALTER TABLE statement to alter the column.

To ensure that referential integrity is maintained in the most appropriate manner for your database, be sure that you understand both actions and select the most effective combination for all tables in your database.
Creating Indexes

A B-TREE index is automatically created on the primary key columns of a table at table creation. You can create optional indexes to improve query performance. Red Brick Warehouse has three types of index technologies: STAR indexes, TARGET indexes, and B-TREE indexes. For guidelines about when to create additional indexes and when to use the various types of indexes, refer to “Determining When to Create Additional Indexes” on page 4-3.

The CREATE INDEX statement provides a parallel-index-creation capability, which allows multiple indexes to be built simultaneously on multi-processor hardware platforms. Although designed to improve performance on multi-processor systems, the ability to create multiple indexes with a single statement is convenient on any system.

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 backup 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 non-empty table. You can drop any index at any time.

INDEX TEMPSPACE Parameters

You should set the INDEX TEMPSPACE DIRECTORIES location to large empty disk partitions in order 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. However, 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 do encounter out-of-memory errors when building indexes in parallel, try building fewer indexes in parallel, reducing the INDEX TEMPSPACE THRESHOLD value, or increasing the swap space.
The goal in selecting the INDEX TEMPSPACE THRESHOLD value is to select a large value relative to the system on which Red Brick 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, refer to “Estimating Temporary Space Values for Index-Building Operations” on page 4-34.

**Parallel Indexes**

Creating multiple indexes in parallel on a table with a single CREATE INDEX statement is quicker on multi-processor 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 warehouse 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, then manually remove that index entry from RBW_INDEXES with the DROP INDEX command.

**Large Dimension Tables**

For large dimension tables (more than 10,000 rows) with user-defined indexes, creating the indexes after the load process has completed is usually more efficient than creating them before the load process.

**Tables with Multiple Indexes**

If a dimension table will have several indexed columns, it is usually faster to drop indexes before loading data and then re-create the indexes using the parallel indexes feature after the load operation is complete. This method also has the advantage that the table is available for query while the indexes are being built, although with lower query performance.
Creating a Database
Creating Indexes

**STAR Indexes**

You can create one or more STAR indexes on any table that has foreign key references. 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 can greatly improve query performance when you are using a star schema design. For more information about STAR indexes, refer to “STAR Indexes” on page 4-4.

**TARGET Indexes**

You can create a TARGET index on any column of a table. TARGET indexes can improve performance on queries involving columns that have weakly selective constraints. For example, if you have a column that 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 very fast way of counting the number of occurrences of a weakly selective constraint. For more information about TARGET indexes, refer to “TARGET Indexes” on page 4-8.
Creating Views

A view is composed 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, then define a view containing only those columns and rows. If some tables contain confidential information, then 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 added to the table are not reflected in the view. Views cannot be updated (you cannot insert, update, or delete rows in a view); however, views do reflect changes to 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.

If you are licensed for the Red Brick Vista option, you can also create precomputed views to automatically rewrite queries to access an aggregate table instead of a detail table. For information about the Red Brick Vista option, refer to the 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:

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 will be those in markets that the sales rep 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 warehouse 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 1,024 characters.
- A macro definition can include other macros so long as the definition is not 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 command to see how a macro is expanded, including any parameter values.

The CATEGORY and COMMENT values are stored in the corresponding columns of the RBW_MACROS system table. The CATEGORY value is not used by Red Brick 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, refer to the SQL Reference Guide.
Creating a Database
Creating and Managing Macros

Availability and Scope

The way in which a macro is defined determines its availability and scope. 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.

Note: For tools that start and end a new session for each query, temporary macros are of limited use unless they are stored in .rbwrc files.

The following table defines the macro definition rules and scope.

<table>
<thead>
<tr>
<th>Macro Type</th>
<th>Required Authorization</th>
<th>For User</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>DBA (CREATE ANY task)</td>
<td>All</td>
<td>CREATE PUBLIC MACRO ...</td>
</tr>
<tr>
<td>Private</td>
<td>DBA or RESOURCE (CREATE ANY or CREATE OWN task)</td>
<td>Creator</td>
<td>CREATE MACRO...</td>
</tr>
<tr>
<td>Temporary</td>
<td>CONNECT (any user)</td>
<td>Creator</td>
<td>CREATE TEMPORARY MACRO...</td>
</tr>
</tbody>
</table>

Note: The way in which macros are handled varies among the individual tools used with Red Brick Warehouse. For information about macro use by a particular tool, refer to specific Red Brick 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, then the private macro is used.
- If no temporary or private macro exists and a public macro with the name exists, then 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 \texttt{std\_select} is defined to shorten an often-repeated select list:

```
create macro \texttt{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 this macro in a SELECT statement that finds sales data for San Jose in 1994:

```
select \texttt{std\_select} and year = 1994 and city = 'San Jose';
```

This example illustrates how frequently used blocks of similar instructions can be replaced with a parameterized macro. A macro named \texttt{std\_constraint} is defined with two parameters to provide a general-purpose constraint for a frequently asked query:

```
create macro \texttt{std\_constraint} (yr, cty) as
  year = yr and city = cty;
```

\textbf{Note:} 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 \texttt{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 table; note that the macro identifier is stored as uppercase letters.

```
select name, text from rbw_macros
where name = '\texttt{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 command:

```
expand \texttt{std\_constraint} (1994, 'San Jose');
```

```
STATEMENT
YEAR=1994 AND CITY='San Jose';
```
To use the parameterized macro in a SELECT statement to find 1994 sales data for both San Jose and Miami:

```
select std_select and std_constraint (1994,'San Jose');
...
select std_select and std_constraint (1994,'Miami');
...
```

The following macro is defined to provide a single command 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);
```

This macro allows you to use a short statement to find the desired sales data:

```
std_query (1994,'San Jose');
std_query (1994,'Miami');
```

For more examples of macro definitions, refer to the SQL Self-Study Guide.
Providing Database Access and Security

After creating a database, the warehouse administrator has access through a single user account. In order to provide access to other users, the warehouse 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 data warehouse users at your site will access the database via LAN-connected client applications, you need not create operating system accounts for those users. However, if they access the database locally, via the RISQL Entry Tool or RISQL Reporter running on the same machine, 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 using the SQL GRANT command.

Creating Operating System Accounts for Users

After 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 each user’s path.
- The RB_CONFIG and RB_HOST environment variables are set correctly and are accessible by each user.
- The RB_PATH environment variable is set 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:

```bash
... PATH=$PATH:/usr/redbrick_dir/bin;export PATH
RB_CONFIG=/usr/redbrick_dir; export RB_CONFIG
RB_HOST=RB_HOST; export RB_HOST
...
```

For more information about these environment variables, refer to Chapter 2, “Key Concepts.”

No special permissions or privileges are required for RISQL Entry Tool or RISQL Reporter user accounts. For more information about database access from these tools, refer to the RISQL Entry Tool and RISQL Reporter User’s Guide.
Granting Database Access

Whether warehouse 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 authorization command.

**Note:** Some ODBC client applications—for example, Microsoft Access™ and Visual Basic™—require that you grant resource privileges with the SQL GRANT command 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 warehouse 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 command to add the user’s 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 command to make the user a member of the appropriate system role and to assign the appropriate object privileges, as described in “Granting Access with System Roles” on page 6-5 and “Granting Database Object Privileges” on page 6-7.

**Note:** If you have installed and licensed the Enterprise Control and Coordination option, refer to “Granting Access with Role-Based Security” on page 6-8 for information about other methods of granting and controlling database access.

Database users can be dropped from a database with the REVOKE CONNECT command.

For information about the GRANT, REVOKE, and CREATE ROLE commands, refer to the SQL Reference Guide.
Providing Database Access and Security
Adding Database Users

Examples

The following GRANT CONNECT statement creates the database username 
*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, she 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

Using the GRANT CONNECT command, database users can change their own 
passwords and the warehouse administrator can change any user’s password. 
To change a password, simply specify the database username and supply a 
new password.

A password can be any valid database identifier or string literal, as defined in 
the *SQL Reference Guide*. If your site has installed the Enterprise Control and 
Coordination option, there might be additional security restrictions imposed 
on passwords.

**Note:** Database passwords for accounts that use the RISQL Entry Tool should 
not exceed 8 characters. If database passwords exceed 8 characters, they 
cannot be typed at a password prompt and must be entered as a 
command-line option, which might compromise security.

Example

The following GRANT CONNECT statement changes *drew’s* password to 
*se2cure*. Either *drew* or the warehouse administrator can issue this statement.

```
grant connect to drew with se2cure ;
```
Granting Access with System Roles

All Red Brick Warehouse databases have three predefined system roles: CONNECT, RESOURCE, and DBA:

- 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:

<table>
<thead>
<tr>
<th>Tasks Permitted</th>
<th>System Roles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DBA</td>
</tr>
<tr>
<td>GRANT/REVOKE system roles</td>
<td>Yes</td>
</tr>
<tr>
<td>CREATE database objects</td>
<td>Yes</td>
</tr>
<tr>
<td>ALTER database objects</td>
<td>Yes</td>
</tr>
<tr>
<td>DROP database objects</td>
<td>Yes</td>
</tr>
<tr>
<td>Select data</td>
<td>Yes</td>
</tr>
<tr>
<td>Modify data</td>
<td>Yes</td>
</tr>
<tr>
<td>GRANT/REVOKE object privileges</td>
<td>Yes</td>
</tr>
<tr>
<td>LOCK the database</td>
<td>Yes</td>
</tr>
<tr>
<td>BACKUP the database</td>
<td>Yes</td>
</tr>
<tr>
<td>RESTORE the database</td>
<td>Yes</td>
</tr>
<tr>
<td>UPGRADE the database</td>
<td>Yes</td>
</tr>
<tr>
<td>REORG a table</td>
<td>Yes</td>
</tr>
<tr>
<td>Perform offline loads</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Providing Database Access and Security
Granting Access with System Roles

For information about using role-based security with the Enterprise Control and Coordination option to break down the tasks of the system roles and recombine them into new roles, refer to “Granting Access with Role-Based Security” on page 6-8.

Granting and Revoking the DBA and RESOURCE System Roles

As a member of the DBA system role, you can use the GRANT command 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 using the REVOKE authorization and role command. For more information about the GRANT and REVOKE commands, refer to the SQL Reference Guide.

Examples

The following GRANT statement grants the RESOURCE system role to drew, who has already been granted CONNECT. The database user drew will be able to create database objects and access and modify these objects.

grant resource to drew ;

The following REVOKE statement removes 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. There are five object privileges:

- 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 command 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 using the REVOKE command.

The following table defines the actions permitted on database objects for a user who is a member of the DBA system role, who created the object, who has been granted object privileges, and for all others.

<table>
<thead>
<tr>
<th>Object Privileges Permitted</th>
<th>Users</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DBA</td>
</tr>
<tr>
<td>SELECT from object</td>
<td>Yes</td>
</tr>
<tr>
<td>INSERT into object</td>
<td>Yes</td>
</tr>
<tr>
<td>UPDATE object</td>
<td>Yes</td>
</tr>
<tr>
<td>DELETE from object</td>
<td>Yes</td>
</tr>
<tr>
<td>Use public macro</td>
<td>Yes</td>
</tr>
</tbody>
</table>

¹Members of the RESOURCE system role have all object privileges on tables they create.
²To insert rows, users must have INSERT and SELECT privilege on the object.
³Not applicable.

For a complete discussion of the GRANT and REVOKE commands, refer to the SQL Reference Guide.

Example

This example illustrates how curly (with RESOURCE) can grant the SELECT privilege on a table named t1 that he created to moe (who has already been granted CONNECT):

```
grant select on t1 to moe ;
```
The Enterprise Control and Coordination option includes role-based security, a feature that 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 into new roles, and group database users into custom, or user-created, roles.

A user-created role can consist of any combination of the following:
- Task authorizations, as defined in the table on page 6-9
- Object privileges, as defined in “Granting Database Object Privileges” on page 6-7
- 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.

Note: The DBA and RESOURCE system roles cannot be altered.

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.

<table>
<thead>
<tr>
<th>Task Authorization</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS_ADVISOR_INFO</td>
<td>Query the Advisor system tables. This is part of the Red Brick Vista option. For information about the Advisor, refer to the <em>Red Brick Vista User’s Guide</em>.</td>
</tr>
<tr>
<td>ACCESS_ANY</td>
<td>Select data from all database objects and access private user information (such as private macros) in the system tables.</td>
</tr>
<tr>
<td>ACCESS_SYSINFO</td>
<td>Query the dynamic statistic tables for statistics about database activity. For information about the dynamic statistic tables, refer to “Monitoring Database Activity” on page 7-5.</td>
</tr>
<tr>
<td>ALTER_ANY</td>
<td>Alter columns, indexes, macros, segments, synonyms, tables, and views.</td>
</tr>
<tr>
<td>ALTER_SYSTEM</td>
<td>Issue the ALTER SYSTEM command to perform database administration tasks.</td>
</tr>
<tr>
<td>BACKUP_DATABASE</td>
<td>Back up the database.</td>
</tr>
<tr>
<td>CREATE_ANY</td>
<td>Create any object, including those that use another user’s resources. For example, create an index on another user’s table or create a table that resides in another user’s segment.</td>
</tr>
<tr>
<td>DROP_ANY</td>
<td>Drop objects created by any user.</td>
</tr>
<tr>
<td>GRANT_TABLE</td>
<td>Grant object privileges to database users and roles.</td>
</tr>
<tr>
<td>IGNORE_QUIESCE</td>
<td>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. This is part of the Enterprise Control and Coordination option.</td>
</tr>
<tr>
<td>LOCK_DATABASE</td>
<td>Lock the database.</td>
</tr>
<tr>
<td>MODIFY_ANY</td>
<td>Insert, update, delete, and load any data.</td>
</tr>
</tbody>
</table>
### Task Authorization and Definition

<table>
<thead>
<tr>
<th>Task Authorization</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFFLINE_LOAD</td>
<td>Use any segment as a working segment for offline loads; synchronize segments after offline loads.</td>
</tr>
<tr>
<td>PUBLIC_MACROS</td>
<td>Create and drop PUBLIC macros.</td>
</tr>
<tr>
<td>REORG_ANY</td>
<td>Reorganize any table or index.</td>
</tr>
<tr>
<td>RESTORE_DATABASE</td>
<td>Restore the database.</td>
</tr>
<tr>
<td>ROLE_MANAGEMENT</td>
<td>Create, drop, grant, revoke, and alter roles.</td>
</tr>
<tr>
<td>UPGRADE_DATABASE</td>
<td>Upgrade the database.</td>
</tr>
<tr>
<td>USER_MANAGEMENT</td>
<td>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.</td>
</tr>
</tbody>
</table>

The following table defines the task authorizations included in the RESOURCE system role.

<table>
<thead>
<tr>
<th>Task Authorization</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER_OWN</td>
<td>Alter own indexes, segments, and tables.</td>
</tr>
<tr>
<td>ALTER_TABLE_INTO_ANY</td>
<td>Alter own tables into other users’ segments.</td>
</tr>
<tr>
<td>CREATE_OWN</td>
<td>Create own objects (indexes, private macros, segments, synonyms, tables, and views).</td>
</tr>
<tr>
<td>DROP_OWN</td>
<td>Drop own objects.</td>
</tr>
<tr>
<td>GRANT_OWN</td>
<td>Grant object privileges on own objects to other users.</td>
</tr>
<tr>
<td>TEMP_RESOURCE</td>
<td>Create TEMPORARY tables.</td>
</tr>
</tbody>
</table>

**Note:** 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 command and the ALTER SYSTEM command, refer to the *SQL Reference Guide.*
Role Capabilities

In addition to the GRANT and REVOKE capabilities available with the predefined system roles, you can also use role-based security to:

- Grant a task authorization to one or more database users. For example:
  
  ```sql
  grant restore_database to db_user ;
  ```

- Create a user-defined role. For example:
  
  ```sql
  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:
  
  ```sql
  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:
  
  ```sql
  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:
  
  ```sql
  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:
  
  ```sql
  grant table_select_role to marketing_role ;
  ```

  Note that 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:
  
  ```sql
  grant resource to marketing ;
  ```

  Note: 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:
  
  ```sql
  revoke select on period from table_select_role ;
  ```

- Revoke a task authorization from a user-created role or a database user. For example:
  
  ```sql
  revoke upgrade_database from db_management_role ;
  ```

- Drop a user-created role. For example:
  
  ```sql
  drop role table_select_role ;
  ```
Creating Roles

Use the CREATE ROLE command 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 command 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 command to grant object privileges to a role.

For complete syntax of the CREATE ROLE and GRANT commands, refer to the SQL Reference Guide.

Example

The following CREATE ROLE statement creates the role security_management and assigns chris and judy as direct members:

```sql
create role security_management for chris, judy ;
```

Granting Task Authorizations

Use the GRANT authorization and role command 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 maria, john, and joe to upgrade and restore the database:

```sql
grant upgrade_database, restore_database to maria, john, joe ;
```

The following example illustrates how to create a role for a group of users, then grant task authorizations to the role.

1. Create the security_management role listing chris and judy as direct members:

```sql
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 are able to 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 command to grant an object privilege to a user-created role. Granting an object privilege to a role provides all direct and indirect members of the role with the privilege. For information about granting object privileges to users and for a list of the object privileges, refer to “Granting Database Object Privileges” on page 6-7.

Note: You cannot grant object privileges to the system roles; system roles cannot be altered.

For complete syntax of the GRANT privilege command, refer to the SQL Reference Guide.

Example

The following example illustrates how to create the roles table_select and marketing, 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, assume you want to assign select privileges on several database tables to the members of the marketing department; you could accomplish this with 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:

   ```sql
   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:

   ```sql
   grant marketing to db_user1, db_user2, db_user3, db_user4;
   ```

4. Grant the `table_select` role to the `marketing` role:

   ```sql
   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.

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 command 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.
Note that 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 which users have which capabilities. 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 6-20.

For complete syntax of the GRANT authorization and role command, refer to the SQL Reference Guide.

**Example**

The following example illustrates how to create a role, grant task authorizations to the role, then grant the role to a user. Suppose you need to allow a database user to restore or 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`:
   ```sql
   create role database_management ;
   ```

2. Grant the necessary subset of the DBA task authorizations, in this example `LOCK_DATABASE`, `RESTORE_DATABASE`, and `UPGRADE_DATABASE`, to the `database_management` role:
   ```sql
   grant lock_database, restore_database, upgrade_database to database_management ;
   ```

3. Grant this new role to the user who will have this responsibility:
   ```sql
   grant database_management to db_user ;
   ```
Providing Database Access and Security
Granting Access with Role-Based Security

The user `db_user` becomes a member of the `database_management` role and can lock, restore, and upgrade the database.

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 `sudhir`, `nasi`, and `cody` direct role members:
   ```sql
   create role marketing for sudhir, nasi, cody ;
   ```
2. Create the role `object_management`:
   ```sql
   create role object_management ;
   ```
3. Grant task authorizations to the `object_management` role:
   ```sql
   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 `sudhir`, `nasi`, and `cody` indirect members of the `object_management` role and allows them to perform all tasks granted to this role.
   ```sql
   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:

- **Role1**
  - Members: Kirsten, Susan
  - Tasks: CREATE_ANY, MODIFY_ANY

- **Role2**
  - Members: Brian, Hedy
  - Tasks: UPGRADE_DATABASE, REORG_ANY

- **Role3**
  - Members: Emily, Elena
  - Tasks: ACCESS_ANY, ALTER_ANY

As members of Role2, 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. Note that 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. Emily and Elena can now create and modify any database objects and can upgrade and reorganize databases, in addition to performing their Role3 tasks.

Note that if you want to disallow Emily from creating 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 disallow 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 command to revoke task authorizations and roles. Use the REVOKE privilege command 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.

Note: 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 commands, refer to the SQL Reference Guide.

Example

This example shows how a user might be granted a task authorization multiple times and how to completely revoke the authorization from that user. Assume the user Ken is first granted the UPGRADE_DATABASE task authorization directly. Then Ken is granted the database_management role, which has also been granted the UPGRADE_DATABASE task authorization.

If you do not want ken to upgrade databases but are unsure if he has the authorization to do so, check the RBW_USERAUTH table:

```
select grantee, grantor, upgrade_database
from rbw_userauth ;
```

<table>
<thead>
<tr>
<th>GRANTEE</th>
<th>GRANTOR</th>
<th>UPRG</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATABASE_MANAGEMENT</td>
<td>SYSTEM</td>
<td>Y</td>
</tr>
<tr>
<td>KEN</td>
<td>DBA</td>
<td>Y</td>
</tr>
</tbody>
</table>

Revoke the UPGRADE_DATABASE task authorization from ken:

```
revoke upgrade_database from ken ;
```

To see if ken can still upgrade databases, check the RBW_USERAUTH table again:

```
select grantee, grantor, upgrade_database
from rbw_userauth ;
```

<table>
<thead>
<tr>
<th>GRANTEE</th>
<th>GRANTOR</th>
<th>UPRG</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATABASE_MANAGEMENT</td>
<td>SYSTEM</td>
<td>Y</td>
</tr>
<tr>
<td>KEN</td>
<td>SYSTEM</td>
<td>R</td>
</tr>
</tbody>
</table>
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, which verifies that *ken* can no longer upgrade databases:

```
select grantee, grantor, upgrade_database
from rbw_userauth ;
```

<table>
<thead>
<tr>
<th>GRANTEE</th>
<th>GRANTOR</th>
<th>UPGR DATABASE_MANAGEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATABASE_MANAGEMENT</td>
<td>SYSTEM</td>
<td>Y</td>
</tr>
</tbody>
</table>

If *ken* belonged to multiple roles or if multiple roles had this task authorization, you would have to determine through which role *ken* had the task authorization and either eliminate his membership in that role or remove the authorization from it.

Note that instead of revoking the *database_management* role from *ken*, you could revoke UPGRADE_DATABASE from the *database_management* role. Either method disallows *ken* from performing the task. However, keep in mind that revoking a task from a role prevents all members of the role from performing the task.

**Note:** A role is dropped from the database with the DROP ROLE command. If you drop a role, remember that role members might still have indirect task authorization for some of the dropped role’s tasks. For more information about dropping a role, refer to “Roles” on page 8-41. For complete syntax of the DROP ROLE command, refer to the SQL Reference Guide.
Tracking Role Authorizations and Members

Query the system tables to determine the task authorizations, object privileges, and roles that each user has been granted. The following table lists each system table that you might want to query:

<table>
<thead>
<tr>
<th>System Table</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBW_ROLES</td>
<td>Roles that exist in the database</td>
</tr>
<tr>
<td>RBW_ROLE_MEMBERS</td>
<td>Members of each role</td>
</tr>
<tr>
<td>RBW_USERAUTH</td>
<td>Task authorizations of each user and role</td>
</tr>
<tr>
<td>RBW_TABAUTH</td>
<td>Object privileges of each user and role</td>
</tr>
</tbody>
</table>

The following examples illustrate how to query the system tables to determine the access rights within your database.

Example

The following statement returns a list of all user-created roles in the database:

```
slect name, creator
from rbw_roles
order by name ;
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>CREATOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATABASE_MANAGEMENT</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>MARKETING</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>OBJECT_MANAGEMENT</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>SECURITY_MANAGEMENT</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>TABLE_SELECT</td>
<td>SYSTEM</td>
</tr>
</tbody>
</table>
Example

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 ;
```

<table>
<thead>
<tr>
<th>ROLENAME</th>
<th>USERNAME</th>
<th>INDI</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATABASE_MANAGEMENT</td>
<td>JOHN</td>
<td>N</td>
</tr>
<tr>
<td>MARKETING</td>
<td>CODY</td>
<td>N</td>
</tr>
<tr>
<td>MARKETING</td>
<td>NASI</td>
<td>N</td>
</tr>
<tr>
<td>MARKETING</td>
<td>SUDHIR</td>
<td>N</td>
</tr>
<tr>
<td>OBJECT_MANAGEMENT</td>
<td>CODY</td>
<td>Y</td>
</tr>
<tr>
<td>OBJECT_MANAGEMENT</td>
<td>MARKETING</td>
<td>N</td>
</tr>
<tr>
<td>OBJECT_MANAGEMENT</td>
<td>NASI</td>
<td>Y</td>
</tr>
<tr>
<td>OBJECT_MANAGEMENT</td>
<td>SUDHIR</td>
<td>Y</td>
</tr>
<tr>
<td>SECURITY_MANAGEMENT</td>
<td>CHRIS</td>
<td>N</td>
</tr>
<tr>
<td>SECURITY_MANAGEMENT</td>
<td>JUDY</td>
<td>N</td>
</tr>
</tbody>
</table>

**Note:** The RISQL Entry Tool truncates headings of columns of datatype CHAR(1) to four characters.
Example

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, restore_database, upgrade_database,
  alter_table_into_any, create_own, alter_own, grant_own,
  isrole
from rbw_userauth
order by grantee;
```

| GRANTEE | DBAA | RESA | USER | GRAN | ROLE | ALTE | PUBL | ACCE | MODI | DROP | CREA | LOCK | REST | UPGR | ALTE | CREA | ALTE | GRAN | ISRO |
|---------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| CHRIS   | N    | N    | R    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    |
| CODY    | N    | N    | N    | N    | I    | I    | I    | I    | I    | N    | N    | N    | N    | N    | N    | N    | N    |
| DATABASE_M | N  | N    | N    | N    | N    | N    | N    | N    | N    | N    | Y    | Y    | Y    | N    | N    | N    | N    |
| JOE     | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    |
| JOHN    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    |
| JUDY    | N    | N    | R    | R    | R    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    |
| MARIA   | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | Y    | Y    | Y    | N    | N    | N    | N    |
| MARKETING | N | N    | N    | N    | N    | R    | R    | R    | R    | R    | N    | N    | N    | N    | N    | N    | N    |
| NASI    | N    | N    | N    | N    | N    | I    | I    | I    | I    | I    | N    | N    | N    | N    | N    | N    | N    |
| OBJECT_Man | N | N    | N    | N    | Y    | Y    | Y    | Y    | Y    | Y    | N    | N    | N    | N    | N    | N    | N    |
| SECURITY_M | N | N    | Y    | Y    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    |
| SUDHIR  | N    | N    | N    | N    | N    | I    | I    | I    | I    | I    | N    | N    | N    | N    | N    | N    | N    |
| SYSTEM  | Y    | N    | Y    | Y    | Y    | Y    | Y    | Y    | Y    | Y    | Y    | Y    | Y    | Y    | Y    | Y    | Y    |
| TABLE_SEL | N | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    | N    |

Note: The RISQL Entry Tool truncates headings of columns of datatype CHAR(1) to four characters.

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 one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>User or role does not have the task authorization.</td>
</tr>
<tr>
<td>Y</td>
<td>User or role has the task authorization directly.</td>
</tr>
<tr>
<td>R</td>
<td>User or role has the task authorization through direct membership in a role.</td>
</tr>
<tr>
<td>I</td>
<td>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.</td>
</tr>
</tbody>
</table>
Example

The following statement returns a list of all users and roles with object privileges in the database:

```
select grantee, grantor, tname, sela, insa, dela, upda,
     updauth
from rbw_tabauth
order by grantee ;
```

<table>
<thead>
<tr>
<th>GRANTEE</th>
<th>GRANTOR</th>
<th>TNAME</th>
<th>SELA</th>
<th>INSA</th>
<th>DELA</th>
<th>UPDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>MARIA</td>
<td>SYSTEM</td>
<td>PRODUCT</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>TABLE SELECT</td>
<td>SYSTEM</td>
<td>SALES</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>TABLE SELECT</td>
<td>SYSTEM</td>
<td>PERIOD</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>TABLE SELECT</td>
<td>SYSTEM</td>
<td>PRODUCT</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>TABLE SELECT</td>
<td>SYSTEM</td>
<td>MARKET</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
Administering Password Security

With the Enterprise Control and Coordination option, password security features allow you, as the warehouse 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 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 warehouse administrator can implement the appropriate password security features by setting only the parameters that apply to that site or database environment.

The warehouse administrator (or any user with the USER_MANAGEMENT task authorization) initially creates passwords for users by using the GRANT CONNECT command. To continually use the database, users must comply with the configured password parameters, using the GRANT CONNECT command to change their own passwords as required. For the syntax of the GRANT CONNECT command, refer to the SQL Reference Guide.
The following table lists the password parameters and describes their functions.

<table>
<thead>
<tr>
<th>PASSWORD Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPIRATION_DAYS</td>
<td>Maximum number of days passwords exist. Forces users to change their passwords regularly.</td>
</tr>
<tr>
<td>EXPIRATION_WARNING_DAYS</td>
<td>The number of days prior to password expiration that the user receives a warning message.</td>
</tr>
<tr>
<td>RESTRICT_PREVIOUS</td>
<td>Minimum number of password changes that must occur before a password can be reused. Forces users to use different passwords.</td>
</tr>
<tr>
<td>CHANGE_MINIMUM_DAYS</td>
<td>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 PASSWORD RESTRICT_PREVIOUS.</td>
</tr>
<tr>
<td>MINIMUM_LENGTH</td>
<td>Minimum number of total characters required in each password.</td>
</tr>
<tr>
<td>COMPLEX_NUM_ALPHA</td>
<td>Minimum number of alphabetic characters (A–Z, a–z) required in each password.</td>
</tr>
<tr>
<td>COMPLEX_NUM_NUMERIC</td>
<td>Minimum number of numeric characters (0–9) required in each password.</td>
</tr>
<tr>
<td>COMPLEX_NUM_PUNCTUATION</td>
<td>Minimum number of punctuation characters (for example, !@#$%) required in each password.</td>
</tr>
<tr>
<td>LOCK_FAILED_ATTEMPTS</td>
<td>Maximum number of failed connection attempts allowed before a user’s account is locked.</td>
</tr>
<tr>
<td>LOCK_PERIOD_HOURS</td>
<td>Number of hours a user’s account is locked following failed connection attempts.</td>
</tr>
</tbody>
</table>


Enforcing Password Changes

You can force users to periodically change their passwords 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 command within the specified number of days.

Syntax

To specify a password expiration period, enter a line in the rbw.config file using the following syntax:

```
num_days
```

Specify the number of days passwords can exist before they expire; this must be an integer in the range of 0 to 512. The default is 0, which sets no restriction —users need never change their passwords. To compute a password expiration date, the num_days value is added to the base value stored in the PASSWORD_TS column in the RBW_USERAUTH system table.

Usage Notes

- Users can change their passwords at any time before the expiration date unless the PASSWORD CHANGE_MINIMUM_DAYS parameter is set, which sets the minimum number of days that must pass before users can change their passwords. If PASSWORD EXPIRATION_DAYS and PASSWORD CHANGE_MINIMUM_DAYS are both set, users must change their passwords after the minimum number of days have passed and before the expiration date. For more information about PASSWORD CHANGE_MINIMUM_DAYS, refer to “Limiting Frequency of Password Changes” on page 6-29.
- If an account expires, the user cannot connect to the database until the warehouse administrator (or any user with the USER_MANAGEMENT task authorization) assigns a new password with the GRANT CONNECT command.
- When an account expires, the user’s status changes from valid to expired, as indicated in the RBW_USERAUTH system table. After the warehouse 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 WARNING_DAYS parameter. This parameter sets the number of days before the password expiration date that users receive a warning message. (The message is displayed each time they connect to the database.)

Syntax

To specify a password warning period, enter a line in the rbw.config file using the following syntax:

```
num_days
```

Specifies the number of days before password expiration that users receive a warning message; this 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 PASSWORD EXPIRATION_DAYS parameter must be greater than the value of the PASSWORD WARNING_DAYS parameter. If not, the PASSWORD WARNING_DAYS parameter 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 WARNING_DAYS parameter is ignored.

Example

Assume that 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.
Providing Database Access and Security
Administering Password Security

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 using the following syntax:

```
num_passwords
```

Specifies the number of password changes required before users can re-create their passwords; this 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. If, for example, 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 warehouse 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 using the following syntax:

```
<table>
<thead>
<tr>
<th>num_days</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASSWORD CHANGE_MINIMUM_DAYS num_days</td>
</tr>
</tbody>
</table>
```

`num_days`

Specifies the number of days that must pass between password changes; this must be an integer in the range of 0 to 128. The default is 0, which indicates no restriction on the frequency of password changes.

Example

Assume the rbw.config file contains the following entries:

```
PASSWORD EXPIRATION_DAYS 60
PASSWORD RESTRICT_PREVIOUS 5
PASSWORD CHANGE_MINIMUM_DAYS 20
```

Users must create new passwords at least every 60 days. They cannot re-create old passwords until they have created 5 subsequent passwords. After changing their passwords, they must wait 20 days before changing passwords again.
Enforcing Password Complexity and Length

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

The PASSWORD MINIMUM_LENGTH parameter sets the minimum number of characters required in each password, while the complexity parameters set the minimum number of alphabetic, numeric, and punctuation characters. (A punctuation character is any printing character that is not a letter, a number, or a space.) The complexity parameters are:

- 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 using the following syntax:

```
PASSWORD — MINIMUM_LENGTH — num_characters
```

**num_characters**

Specifies the minimum number of characters required in each password; this must be an integer in the range of 0 to 128. The default is 0, which sets no restriction.

Syntax: COMPLEX_NUM_ALPHA

To set the number of alphabetic characters required in each password, enter a line in the rbw.config file using the following syntax:

```
PASSWORD — COMPLEX_NUM_ALPHA — num_alpha
```

**num_alpha**

Specifies the minimum number of alphabetic characters required in each password; this 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 using the following syntax:

```
PASSWORD — COMPLEX_NUM_NUMERICS — num_numerics
```

**num_numerics**

Specifies the minimum number of numeric characters required in each password; this 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 using the following syntax:

```
PASSWORD — COMPLEX_NUM_PUNCTUATION — num_punctuation
```

**num_punctuation**

Specifies the minimum number of punctuation characters required in each password; this must be an integer in the range of 0 to 42. The default is 0, which sets no restriction.

**Usage Notes**

- You can set password complexity parameters without also setting the PASSWORD MINIMUM_LENGTH parameter. The combined total of the complexity parameters becomes the minimum required length.
- To enforce a minimum length that is different from the combined total of the complexity parameters, the minimum length parameter must be greater than the combined total. If the combined total exceeds the minimum length value, the minimum length parameter is ignored and the combined total becomes the minimum required length.

**Note:** When a password is created that is not a valid SQL identifier, it must be enclosed in single quotes. For example:

'flo12we$rs', '2flolwers', but flol2_wers
Example

Assume the rbw.config file contains the following entries:

```
PASSWORD COMPLEX_NUM_ALPHA 4
PASSWORD COMPLEX_NUM_NUMERICS 2
PASSWORD COMPLEX_NUM_PUNCTUATION 2
PASSWORD COMPLEX_MINIMUM_LENGTH 10
```

When users create new passwords, the passwords must have at least 10 characters with at least 4 alphabetic, 2 numeric, and 2 punctuation characters. The following GRANT CONNECT commands create valid passwords:

```
grant connect to craig with 'dbslare2fun%%';
grant connect to james with 'sq67lis%fu*%';
```

The following GRANT CONNECT commands return error messages because, given the above configuration, the passwords are invalid:

```
grant connect to maria with dbuser;
grant connect to prema with perforl2mance;
```

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 using the following syntax:

```
num_attempts
```

Specifies the number of failed connection attempts allowed before user accounts are locked; this 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.
Providing Database Access and Security
Administering Password Security

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 has passed, users can connect to the database using the same password.

Syntax

To specify the duration of the lock-out period, enter a line in the rbw.config file using the following syntax:

```
PASSWORD --- LOCK_PERIOD_HOURS --- num_hours
```

`num_hours`

Specifies the number of hours accounts are locked following a lock-out caused by failed connection attempts; this 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 warehouse 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 warehouse administrator has assigned a new password or until the number of hours set in the PASSWORD LOCK_PERIOD_HOURS has passed.

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 warehouse administrator has assigned 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:

```sql
select grantee, locked
from rbw_userauth
where grantee = 'user_name';
```
Providing Database Access and Security
Administering Password Security
Managing Database Activity with the Enterprise Control and Coordination Option

This chapter describes the database management, logging, and accounting features of the Red Brick Warehouse Enterprise Control and Coordination option, and is organized into the following sections:

- Enterprise Control and Coordination
- Administration Daemon
- Event Logging
- Controlling Advisor Logging
- Accounting
Managing Database Activity with the Enterprise Control and Coordination Option

Enterprise Control and Coordination

The Enterprise Control and Coordination option provides functionality intended for enterprise-wide Red Brick Warehouse implementations. In these scenarios, Red Brick Warehouse is installed and used throughout the organization.

Although enterprise scenarios differ from case to case, they do share some common characteristics:
• Multiple Red Brick Warehouse installations on different platforms
• Duplication of data over multiple warehouses
• Relatively high number of users accessing the warehouse
• Users with varying levels of expertise accessing the warehouse
• Users from different departments accessing the warehouse
• Users with different roles within the organization accessing the warehouse

There are some warehouse administration tasks associated with these common characteristics:
• Monitoring and controlling contention for warehouse resources (for example, CPU time, database tables, disk I/O)
• Granting users access to data appropriate to their role or department
• Determining level of warehouse use by user or department in order to implement a charge-back accounting system
• Copying data between tables in different warehouse installations

The Enterprise Control and Coordination option for Red Brick Warehouse provides the following features that allow warehouse administrators to perform these tasks:
• System Administration
  There are two aspects of this feature—a set of dynamic statistic tables (DSTs) for the warehouse and an ALTER SYSTEM command. The DSTs are nonpersistent and store database activity statistics for all active databases in the warehouse. The ALTER SYSTEM command is a RISQL command for controlling database activity. A warehouse administrator can monitor database use by querying the DSTs and control database use with the ALTER SYSTEM command.
Managing Database Activity with the Enterprise Control and Coordination Option
Enterprise Control and Coordination

- Event Logging
  The logging subsystem writes records to log files that describe various warehouse events. All warehouse installations have the logging subsystem but only installations with the Enterprise Control and Coordination option can use the ALTER SYSTEM command to control which events are logged. A log viewer for displaying the contents of the log file is also provided.

- Advisor Logging
  For users of the Red Brick Vista option, the Advisor logs historical query information that is used to determine the effectiveness of any precomputed views you have created and to suggest new views that, if created, will improve query performance. For information about the Advisor and about precomputed views, refer to the Red Brick Vista User’s Guide.

- Accounting
  In addition to logging events, the logging subsystem writes accounting records to accounting files. Accounting records contain metrics for determining the workload for which individual queries and loads are responsible.

- Copy Management Utility
  The copy management utility, rb_cm, provides an interface for directing the output from a TMU unload operation to a TMU load operation. The unload and load operations can occur on the same computer or on different computers over a network. For information on the rb_cm utility, refer to the Table Management Utility Reference Guide.

- Role-Based Security
  With this feature you can grant a user specific task authorizations and define roles consisting of task authorizations, object privileges, and users.

- Password Security
  The password security package consists of a set of configuration parameters for controlling the longevity and content of database passwords. For information on password security, refer to “Administering Password Security” on page 6-24.

- System Table Comment Columns
  Many of the Red Brick System tables have a COMMENT column. This allows the warehouse administrator to store a comment for each table, column, view, synonym, index, segment, user, role, and macro in the database.

- System Table for Load Operations
  The RBW_LOADINFO table contains information on each load operation performed against a database.
**Managing Database Activity**

In an enterprise there may be many users issuing queries against a Red Brick Warehouse database. Some of these queries may have higher priority than others. In this situation the warehouse administrator must be able to:

- Monitor the database activity of users, their sessions, and the queries issued by those sessions, to identify resource contention, misuse of the system, or queries that are consuming too many resources.
- Perform actions to control database activity.

The Enterprise Control and Coordination option provides both of these capabilities with the dynamic statistic tables and the `ALTER_SYSTEM` command.

**Task Authorizations for Managing Database Activity**

To monitor and control database activity, a user must have two task authorizations: the `ACCESS_SYSINFO` and `ALTER_SYSTEM` authorizations. The DBA system role includes these authorizations. For more information on task authorizations, refer to Chapter 6, “Providing Database Access and Security.”

Users with the `ACCESS_SYSINFO` authorization can monitor activity on the database to which they are currently connected. Users with the `ALTER_SYSTEM` authorization can control the use of the database to which they are currently connected.

**Administration Database**

When you install Red Brick Warehouse with the installation script, you have the option of creating a special administration database named ADMIN. The installation script builds this database in a subdirectory of the installation directory named `admin_db`. As its name suggests, this database is for administrative purposes only. The database only contains system tables and you cannot create any segments or tables in it.

The administration database provides the following special privileges:

- A user who has `ACCESS_SYSINFO` authorization for the administration database and is connected to that database can obtain database activity statistics for all warehouse 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:

![Diagram illustrating the role of the administration database]

**Monitoring Database Activity**

Statistics on the activity associated with each database are available through a set of dynamic statistic tables (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:

- DST_DATABASES
- DST_USERS
- DST_SESSIONS
- DST_COMMANDS
- DST_LOCKS
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 only holds information on those databases that have been accessed at least once since the administration daemon was started. Similarly, the administration daemon only holds information on those users who have accessed a database at least once since the administration daemon was started. For more information on the administration daemon, refer to “Administration Daemon” on page 7-11.

**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 cache writes
- Logical reads and logical writes
- Physical reads and physical writes

These statistics are important but easily misinterpreted; therefore, they are defined here.

### Definition of Read Statistics

Consider the case in which a session must read data. It first determines the location of that data (the particular block) and attempts to lock the block. The 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 only counts block read requests. If the required block is not already in the local buffer cache, however, a logical read request is issued by the session 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 exists in an operating system buffer, no physical read to disk is required. If the data does not exist 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:

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.

**Note:** 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 server must read the block from disk. The server process can then lock the block for writes. When the server is finished 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 server process can lock the block for writes. Again the 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 server process can lock the block to perform additional writes. Since 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 logical write is performed, the operating system usually performs a corresponding physical read or physical write to disk. However, the physical I/O data is kept by the operating system, so the availability of these statistics is platform dependent.

Controlling Database Activity

The command for controlling database activity is the ALTER SYSTEM command. Users with ALTER_SYSTEM authorization can issue the ALTER SYSTEM command to control use of the current database. Users with ALTER_SYSTEM authorization on the administration database who are connected to the administration database can issue the ALTER SYSTEM command to control use of all databases in the warehouse.

With the ALTER SYSTEM command, a user with the necessary authority can perform the following operations:

- Quiesce a database
- Activate a database
- Reset accumulated statistics
- Cancel a user command
- Close a user session
- Change user priority for current sessions

The following sections describe all of these operations. For a complete description of the ALTER SYSTEM syntax, refer to the SQL Reference Guide.

Quiescing a Database

You can use the ALTER SYSTEM QUIESCE DATABASE command to bring a database into 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 command as preparation for shutting down the rbwapid daemon process. The quiescent state is also useful for performing maintenance tasks such as disk drive maintenance.

Note: A user with the IGNOREQUIESCE task authorization can perform actions on a quieced database, thus overriding another user’s ALTER SYSTEM QUIESCE DATABASE command. All users with the DBA system role automatically have the IGNOREQUIESCE task authorization.
Activating a Database

You can use the ALTER SYSTEM RESUME DATABASE command to bring a database into normal working mode.

The RESUME DATABASE clause must be issued by an existing session (since 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 command 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 command for a specific session. You can also cancel the currently executing commands 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 commands 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 command 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 commands 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 command 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 command described in the SQL Reference Guide.

Note: Your platform must have the UNIX renice command in order to support user priorities. You must specify the full pathname of the renice executable file with the ADMIN RENICE_COMMAND configuration parameter.
Administration Daemon

The administration daemon collects statistics for the DSTs and carries out ALTER SYSTEM commands. The administration daemon process (rbwadmd) is started at the same time as the warehouse (rbwapid) and log (rbwlogd) daemon processes. If the administration daemon goes down for some reason but the warehouse daemon is still running, you can restart the administration daemon with the rbw.start script—the script checks which daemons are already running before starting up new ones. To stop the administration daemon, use the ALTER SYSTEM TERMINATE ADMIN DAEMON command.

The administration daemon collects statistics from the TMU and server processes and returns DST data to a server process. The following figure illustrates statistics data flow between the administration daemon and the other warehouse processes:

The administration daemon accepts ALTER SYSTEM commands from the server processes and performs the appropriate administrative actions on the TMU or on another server process. The following figure illustrates the ALTER SYSTEM flow of control between the administration daemon and the other warehouse processes:

The Statistics Collection Interval

Collection of database activity statistics begins as soon as the rbwadmd daemon process is started, and continues for as long as the that process is running. Unless you specifically reset the statistics for a database, the collection interval for that database is the interval that the rbwadmd process has been running.
If you terminate the *rbwadmd* process, you will lose all statistics currently held in the dynamic statistic tables.

The *rbwadmd* process only collects statistics for databases that have been accessed at least once since the *rbwadmd* process was started. Similarly, the *rbwadmd* process only collects statistics for database users who have accessed a database at least once since the *rbwadmd* process was started.

To reset all statistics for a database (or all databases) to zero, use the ALTER SYSTEM RESET STATISTICS command.

### The DST Refresh Interval

You can set the maximum interval between dynamic statistic table refreshes by using the ADMIN REPORT_INTERVAL configuration parameter and the SET REPORT_INTERVAL command.

Whenever a statement requires a change of state or whenever a session requests, acquires, or releases a lock, the 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. You can override the ADMIN REPORT_INTERVAL value for the duration of a session using 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:

```
 ADMIN REPORT_INTERVAL ——— integer
```

The `integer` value is in minutes.

To set the DST refresh interval for a session, issue a SET REPORT_INTERVAL command using the following syntax:

```
 SET REPORT_INTERVAL ——— integer ;
```

You can turn off statistics collection by setting the DST refresh interval to zero (either by setting the configuration parameter or issuing a SET REPORT_INTERVAL command for a session).
Event Logging

Enterprise systems will often have many users from different departments accessing the same 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 Enterprise Control and Coordination option provides such information through the event-logging feature. 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 warehouse 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.

Event Logging Process

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. The log daemon is started by the warehouse daemon (`rbwapid`) when you start that process. Any Red Brick process can send log request messages to the log daemon. For example, the `rb_tm` process might generate a load-initiated message, while the `rbwapid` process might generate a message when an abnormal 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 server and the log daemon is one-way. For example, when a user drops a table, the 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:

![Event Logging Diagram](image-url)
When the log daemon receives a log request message from a server process, it adds a timestamp and writes the information contained in that message to a log file. There is at most a single active log file at any given time. If you are not running event logging, there is no active log file. The log daemon closes the file when the disk space limits specified by the ADMIN LOG_MAXSIZE configuration parameter have been reached, and initializes a new file. For more information, refer to “Log Files” on page 7-19.

Log Viewer

You can use the log viewer utility to view the contents of the log. The log daemon writes the parameter values contained in a log message to the log file. The log daemon does not write full message text to the log file. This text is stored in the form of message templates in a separate message base file. When you view the event messages, the log viewer combines the appropriate message template with the parameter values stored in the log file to give you a readable output. The resulting message is displayed to stdout or written to a file. For a discussion of the message parameters, refer to “Event Log Messages” on page 7-17.

The log viewer executable is named rbwlogview. Any user who has read permission for the log files can view event messages using rbwlogview. The log files are owned by the redbrick account.
The `rbwlogview` command has the following syntax:

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

where:

- `a` Specifies the active log file. If you use this option you cannot specify `logfile`.
- `t` Specifies terse output with shorter headers.
- `e` Specifies continuous display of the active log file. As new records are written to the active file they are displayed to stdout. (Similar to the UNIX `tail` command).
- `f` Specifies continuous display of active log file. All records currently in the active file are first displayed, followed by any new records as they are written to file. (Similar to the UNIX `tail` command).
- `p pid` Displays only those log records that originate from the process with the specified process ID.
- `d database` Displays only those log records generated by processes accessing the specified database. Multiple databases can be specified.
- `i sourceid` Displays only those log records that originate from processes with the specified `sourceid` (for example, `rb_tm1`).
- `c` Limits display to the specified categories, one or more of the following: a (audit), e (error), o (operational), s (schema), u (usage). For more information on event categories, refer to “Event Category” on page 7-17.
- `s` Limits display to the specified severities, one or more of the following: u (urgent), a (alert), r (routine). For more information on event severities, refer to “Event Severity” on page 7-17.
- `logfile` Specifies a particular log file to read from. Multiple files can be specified and will be read in the order that they were saved by the log daemon. For more information on log files, refer to “Log Files” on page 7-19.
**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 graphical user interface program:

```
rbwlogview -a -e
```

**Example**

The following command displays log records in the closed log file named `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.
```

The next command displays records for all logged warehouse events as they are logged:

```
% rbwlogview -a -e
Jun 21 15:50:46 rbwapid[20500] OPE077R: Server process PID:6603 started for userid:"redbrick"
```
Event Log Messages

All log messages issued by Red Brick 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 server process generates a message with a message category, a message number, a message severity, and an additional `table_name` parameter. The log daemon writes these parameters to the active log file as a variable-length record.

When viewed using the log viewer, all messages have the following form:

```
CC###S: Message Text
```

The first two characters (CC) indicate the message category, the next three numeric characters (###) represent the message number, the sixth character (S) indicates the message severity. The message text follows the six-character code. An example is:

```
OP081A: New accounting level is WORKLOAD
```

Event Severity

Each event has one of the following event severity levels:

- ROUTINE
- ALERT
- URGENT

The highest severity is URGENT; the lowest severity is ROUTINE.

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 could specify that the minimum severity for ERROR events is ALERT. In this case, only those ERROR events that have ALERT or URGENT severity are logged. To specify the minimum severity level for an event category, you can either use the ALTER SYTEM CHANGE LOGGING LEVEL command or set the appropriate configuration parameter directly. For more information on setting the log severity level, refer to “Setting the Log Severity Filter Level” on page 7-21.

**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:

```
AU011A: User smith supplied incorrect password.
```

**Error Events**

Error events are user actions or changes in the warehouse 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:

```
ER717R: Pipe command not allowed with tape output.
```

**Operational Events**

Operational events are administrative actions taken by warehouse administrators (or other users who are members of the DBA system role) such as component startup or shutdown and changes to operational states. The default minimum severity level is ALERT.

An example of an OPERATIONAL event record is:

```
OP081A: 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 commands). The default minimum severity level is ROUTINE.
An example of a SCHEMA event record is:

```
SC300R: CREATE TABLE FOO completed successfully.
```

**Usage Events**

Usage events are end-user operations in the warehouse system including LOAD DATA, UNLOAD, and all DML statements (for example, SELECTs). The default minimum severity level is ALERT.

An example of a USAGE event record is:

```
US302R: INSERT INTO FOO completed.
```

Setting the USAGE ROUTINE event logs all SQL queries that Red Brick Warehouse processes. This is useful in debugging problems with query tools. Note that this logs all SQL queries and, depending on system usage, could 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 process and 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:

```
rbwlog.<daemon_name>.active
rbwlog.<daemon_name>.<datetime_stamp>
```

The `<datetime_stamp>` suffix on saved file names 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. You can specify a different location by setting the ADMIN LOG_DIRECTORY configuration parameter. All log files are owned by the `redbrick` account.
Old log files are not removed automatically. The warehouse administrator must provide a script to periodically remove these files or remove them manually.

**Note:** If you do not remove old log files, these files will 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 will write to a single log file that grows until limited by available disk space.

**Configuring the Logging Subsystem**

You can configure the logging subsystem by setting 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:

```
ADMIN LOGGING [ON] [OFF]
```

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.

**Note:** You can start or stop logging while the log daemon is running by using the `ALTER SYSTEM START LOGGING` and `ALTER SYSTEM STOP LOGGING` commands.

**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:

```
ADMIN LOG_DIRECTORY pathname
```

The `pathname` can be an absolute pathname or a relative pathname. Relative pathnames are interpreted as 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`. 
**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:

\[
\text{ADMIN LOG_MAXSIZE } \text{integer} \begin{cases} \text{integerK} & \text{if K is specified} \\ \text{integerM} & \text{if M is specified} \end{cases}
\]

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 (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 10K (10,240) bytes.

**Caution:** 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 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**

There are separate configuration parameters for setting 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 shown below:

\[
\text{ADMIN LOG_AUDIT_LEVEL} \quad \text{ROUTINE} \\
\text{ADMIN LOG_ERROR_LEVEL} \quad \text{ALERT} \\
\text{ADMIN LOG_OPERATIONAL_LEVEL} \quad \text{URGENT} \\
\text{ADMIN LOG_SCHEMA_LEVEL} \\
\text{ADMIN LOG_USAGE_LEVEL}
\]

**Note:** You can change the filtering level for each logging category during warehouse operation by using the ALTER SYSTEM CHANGE LOGGING LEVEL command.
Controlling Logging Operations

You can control logging operations using the ALTER SYSTEM command. This command has options for:

- Starting logging
- Stopping logging
- Switching log files
- Changing log filter levels
- Terminating the log daemon

**Note:** The log daemon must be running in order to perform ALTER SYSTEM commands. If you issue an ALTER SYSTEM command to perform one of the actions listed above and nothing happens, verify that the log daemon is running.

For the full syntax of the ALTER SYSTEM command, refer the SQL Reference Guide.

Starting Logging

The ALTER SYSTEM START LOGGING command starts logging. The log daemon:

- Creates a log file.
- Begins accepting log request messages from warehouse processes and writes the corresponding log records to the file.

If logging is already running, this command is ignored.

Stopping Logging

The ALTER SYSTEM STOP LOGGING command stops logging and closes the active log file. The log daemon remains running and you can restart logging at any time. If logging is already stopped, this command has no effect.

Switching Log Files

The ALTER SYSTEM SWITCH LOGGING FILE command closes the active log file, creates a new active log file, and resumes logging to this new file. If logging is stopped, this command has no effect.
Managing Database Activity with the Enterprise Control and Coordination Option

Event Logging

Changing Log Filter Levels

The ALTER SYSTEM CHANGE LOGGING LEVEL command 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 command terminates the log daemon process (`rbwlogd`). The log daemon closes and saves all active files (both log and account files) before shutting down. For information on account files and the accounting feature, refer to the following section, “Accounting.”
Managing Database Activity with the Enterprise Control and Coordination Option
Controlling Advisor Logging

Controlling Advisor Logging

If you have installed the Red Brick Vista option, there are log files for the Advisor. This section describes the commands that control Advisor logging. For detailed information about using the Advisor, refer to the Red Brick Vista User’s Guide.

Note: The Advisor logs cannot be read with the rbwlogview utility. 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 upon 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:

rbwadvlog.<daemon_name>.active
rbwadvlog.<daemon_name>.<datetime_stamp>

The <datetime_stamp> suffix on saved file names 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. You can specify a different location by setting the ADMIN ADVISOR_LOG_DIRECTORY configuration parameter. All log files are owned by the redbrick account.

Old log files are not removed automatically. The warehouse administrator must provide a script to periodically remove these files or remove them manually.

Note: If you do not remove old log files, these files will 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 will write to a single log file that grows until limited by available disk space.
What the Advisor Logs

The Advisor logs the following information:

- **Timestamp**: 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 time a view was referenced to answer queries asking for a subset of the view’s grouping columns 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 candidate view’s definition.
Starting and Stopping the Advisor Log

Use the ADMIN ADVISOR_LOGGING parameter or the ALTER SYSTEM command 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 (`rbwadmd`) 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, then the log records are captured when aggregate views are used and when candidate views are generated.

**Syntax**

The following syntax diagram shows how to set the ADMIN ADVISOR_LOGGING parameter:

```
ADMIN ADVISOR_LOGGING [ON] [OFF]
```

**ON/OFF**

Specifies whether the Advisor log files are created upon system startup. When this parameter is set to ON, a log file is created when the log daemon (`rbwadmd`) starts. When this parameter is set to OFF, no log file is created and data is not logged.
ALTER SYSTEM Operations

ALTER SYSTEM operations control database activity and various administrative actions. The two ALTER SYSTEM commands that control logging activities of the Advisor are the ADVISOR_LOGGING command and ALTER SYSTEM SWITCH ADVISOR_LOG FILE command.

Syntax

The following syntax diagram shows how to construct an ALTER SYSTEM statement:

```
ALTER SYSTEM START ADVISOR_LOGGING
    STOP
    SWITCH ADVISOR_LOG FILE
```

START/STOP ADVISOR_LOGGING

Offers the option to start or stop logging information into the log file. There is no default setting for this command. This command overrides the value set with the ADMIN ADVISOR_LOGGING $rbw.config file parameter.

SWITCH ADVISOR_LOG FILE

Creates a new active log file with a default name.
SET ADVISOR LOGGING

The SET ADVISOR LOGGING command enables or disables advisor query logging for the current session. 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 command, in order for the SET ADVISOR LOGGING command to take effect.

Use this command to control whether a particular query is or is not logged in the advisor log. Use the OPTION ADVISOR_LOGGING rbw.config file parameter to set this parameter globally for all sessions. The default for the rbw.config file parameter is ON.

Syntax

The following syntax diagram shows how to construct a SET ADVISOR LOGGING statement:

```
ON
OFF
ON_WITH_CORR_SUB
```

ON
Specifies that queries that get rewritten are logged (except queries that contain correlated subqueries).

OFF
Specifies that queries are not logged.

ON_WITH_CORR_SUB
Specifies that correlated subqueries, along with other queries that get rewritten, are logged.
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 ADMIN ADVISOR_LOG_DIRECTORY parameter:

```
| ADMIN ADVISOR_LOG_DIRECTORY | pathname |
```

*pathname*

Specifies the directory in which log files are created. The *pathname* must specify an existing directory, and can be a relative or absolute pathname. Relative pathnames are relative to the directory specified by the RB_CONFIG environment variable. If the configuration parameter is not specified, the default logging directory is `$RB_CONFIG/logs`.

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:

```
| ADMIN ADVISOR_LOG_MAXSIZE | integer, integerK, integerM |
```

*integer, integerK, integerM*

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 (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 10K (10,240 bytes).

**Note:** 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.
The SET UNIFORM PROBABILITY FOR ADVISOR command determines whether the log file is scanned in order to compute the reference count for each view when the RBW_PRECOMPVIEWS_UTILIZATION advisor system table is queried. When it is set to ON, it is assumed that all of the views on a base table are referenced the same number of times. The default setting is OFF.

The following syntax diagram shows how to construct a SET UNIFORM PROBABILITY FOR ADVISOR command.
Accounting

It is often useful to have a means of calculating the warehouse workload generated by individual users. For example, you might want to implement a charge-back accounting system whereby users are charged for their warehouse use. The accounting feature described in this chapter provides a record of the warehouse workload generated by each user.

The accounting feature generates records for those server operations that comprise the basic warehouse workload and stores these records on disk. Accounting records are generated when a warehouse 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 generated by warehouse users as a basis for cost accounting and charge-back systems.

Workload accounting generates records that contain the same information as job accounting but that include some additional detail. Workload accounting is intended primarily for the use of Red Brick Systems support personnel for system analysis.

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 redbrick_dir/util/readacct directory contains a sample program for accounting record processing. For more information, refer to the README file in this directory.
**Accounting Process**

Accounting is performed by the log daemon \((rbwlogd)\). The process that generates accounting records is basically the same as the process that generates event log records. The \(rbwsvr\), \(rb_tmu\), or \(rb_ptmu\) processes send accounting request messages to the log daemon after performing basic workload operations. When a process generates an accounting request message, it does not wait for a response. To minimize performance impact, all communication between the generating process and the log daemon is one-way: from the process to the log daemon.

When the log daemon receives an accounting request message from a \(rbwsvr\), \(rb_tmu\), 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 have a self-describing, variable-length record format with the following components:

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/length header. These values represent the actual accounting data.

More information on this record format can be obtained by reading the README file in the \(redbrick_dir/util/readacct\) directory and by examining the sample code for processing accounting records that is located in that same 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 upon 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 contain any header or trailer information.

The active accounting file and the saved accounting files have the following naming conventions:

\[
\text{rbwacct.<daemon_name>.active} \\
\text{rbwacct.<daemon_name>.<datetime_stamp>}
\]

The \(<\text{datetime_stamp}>\) 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. You can specify a different location by setting the ADMIN ACCT_DIRECTORY configuration parameter. All accounting files are owned by the redbrick account.

Old accounting files are not removed automatically. The warehouse administrator must provide a script to periodically remove these files or remove them manually.

**Caution:** If you do not remove old accounting files, these files will 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 will write to a single accounting file that grows until limited by available disk space.
Managing Database Activity with the Enterprise Control and Coordination Option

Accounting

Configuring Accounting

You can configure the accounting subsystem by setting 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:

```
 ADMIN ACCOUNTING [ON | OFF]
```

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.

Note: When the log daemon is running, you can start or stop accounting via the ALTER SYSTEM START ACCOUNTING and ALTER SYSTEM STOP ACCOUNTING commands.

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:

```
 ADMIN ACCT_DIRECTORY pathname
```

The `pathname` can be an absolute pathname or a relative pathname. Relative pathnames are interpreted as relative to the directory specified by the `RB_CONFIG` environment variable. If you do not set the ADMIN ACCT_DIRECTORY parameter, the default accounting directory is `$RB_CONFIG/logs`. 
Managing Database Activity with the Enterprise Control and Coordination Option
Accounting

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:

```
 ADMIN ACCT_MAXSIZE  integer  [integerK | integerM]
```

When an accounting 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 (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 10K (10,240) bytes.

Caution: 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:

```
 ADMIN ACCT_LEVEL  JOB  [WORKLOAD]
```

The level of detail captured in the accounting files depends on whether the warehouse 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. Note that workload accounting has the potential to quickly produce very large accounting files. The accounting detail level can be changed during warehouse operation with the ALTER SYSTEM CHANGE ACCOUNTING LEVEL command.
Managing Database Activity with the Enterprise Control and Coordination Option

Accounting

Controlling Accounting

You can control accounting operations while the warehouse is running by using the ALTER SYSTEM command. This command 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 command, refer to the SQL Reference Guide.

Starting Accounting

The ALTER SYSTEM START ACCOUNTING command starts accounting. The log daemon:

- Creates an accounting file.
- Begins accepting accounting request messages from warehouse processes and writes the corresponding accounting records to the file.

If accounting is already running, this command is ignored.

Stopping Accounting

The ALTER SYSTEM STOP ACCOUNTING command stops accounting and closes the active accounting file. The log daemon remains running and you can restart accounting at any time. If accounting is already stopped, this command has no effect.

Switching Accounting Files

The ALTER SYSTEM SWITCH ACCOUNTING FILE command 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 command has no effect.

Changing Accounting Mode

The ALTER SYSTEM CHANGE ACCOUNTING LEVEL command changes the accounting mode from job to workload or vice versa. This change takes place immediately.
Maintaining a Data Warehouse

This chapter discusses ongoing tasks involved in maintaining a data warehouse, as necessitated by users’ needs or 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 any named segments to accommodate the current batch of input data. Any needed restoration of the database can be done from the original input data files.

For databases that are modified by incremental load operations or INSERT, UPDATE, or DELETE statements, maintenance includes accommodating growth in the database, as well as adjusting to changes in users’ needs or the warehouse environment. For databases that change, backing up the data regularly is an important part of maintenance. For information about backup operations, refer to the Table Management Utility Reference Guide.

The following sections are included in this chapter:

- Locking Tables and Databases
- Monitoring Growth of Tables and Indexes
- Adding Space to a Segment
- Changing an Index Fill Factor
- Altering Segments
- Recovering a Damaged Segment
- Managing Optical Storage
- Altering Tables
- Copying or Moving a Database
- Modifying the Configuration File
- Monitoring and Controlling Warehouse Processes
- Enabling Licensed Options
- Determining Version Information
- Deleting Database Objects and Databases
Locking Tables and Databases

If you are modifying a table or segment, you might want to manually lock the table or database. Changes made to a table or database during a LOCK operation are automatically committed. A rollback operation is not supported.

For a general description of locks, refer to “Table and Database Locks” on page 2-33.

To lock a table or database manually for server activity, use the LOCK and UNLOCK commands.

The TMU automatically locks the tables and database as needed for its operations.

Determining When to Lock a Table or Database

You can lock a table in the following cases:

- If you want to perform consecutive modifications to the data in a table, lock the table to prevent access by other users until all the modifications are done.
- Before beginning a delete operation in order to ensure the access needed to maintain referential integrity of affected tables (with LOCK…FOR DELETE on those tables). Although all required locking is done automatically, the DELETE operations can complete sooner if you manually lock the tables to prevent access by other users between DELETE operations. For information on how the ON DELETE clause works and which tables are locked, refer to “Maintaining Referential Integrity with ON DELETE” on page 5-10.

You must lock a database when you are performing a restore operation or any operation that affects the entire database.

You might also choose to lock the database in the following cases:

- If you are performing a consecutive series of ALTER TABLE statements and want to prevent any intervening operations by other users.
- When you are issuing operations that affect the system tables, such as ALTER SEGMENT.
Specifying Wait Behavior for the Server and TMU

When a server or TMU process encounters a locked table or database, it either waits for the lock to be released or immediately returns a message saying the operation failed because a table (or database) was locked. The default behavior for both the server and the TMU is to wait. You can override the default behaviors.

Change the default wait behavior for warehouse server activities such as ALTER SEGMENT, INSERT, UPDATE, or DELETE to NO WAIT by any of the following means:

- Interactively for the current session with a LOCK statement.
- For all of a user’s server sessions with a SET command in that user’s .rbwrcc file.
- For all server sessions in the warehouse .rbwrcc file.

Change the wait behavior for TMU or Parallel TMU activity to WAIT or NO WAIT by including one or more TMU SET LOCK statements in a TMU control file.

Deadlocks

If waiting for existing locks to be released could result in a deadlock, the lock request is denied and control is returned immediately to the lock requestor. Note that the possibility of deadlocks occurs only when the LOCK TABLE or LOCK DATABASE command is used; a deadlock cannot be caused by the automatic locking operations of the server or TMU. 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_DEADLOCKS ON
```

If you set this option ON and deadlocks occur, you must then use a system command to kill all deadlocked processes.

Note: The default value is OFF.
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 and/or PSUs 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.

Note: If no available filesystem contains additional space, you must make filesystem space available before you can add more data to the segment.

To monitor the growth of a table or index, use the information found in the RBW_SEGMENTS table (TNAME or INAME, NPSUs, TOTALFREE columns) and RBW_STORAGE table (SEGNAME, MAXSIZE, USED columns). Keep in mind that default segments will 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 is issued stating that the STAR indexes based on that table may not be valid and may need to be either rebuilt with the REORG command or dropped and recreated. For information about STAR index growth, refer to “Considerations for Growing Tables” on page 4-44.

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 you see an error message stating 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 recreate the STAR indexes to accommodate table growth. If space still remains in a default segment, you can use the
REORG statement to rebuild the index. If any segments containing the STAR indexes are full, you must use the ALTER SEGMENT command 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 (8,192-byte) blocks, the minimum allocation unit for disk storage in 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 filesystem reaches its MAXSIZE value.

**Example**

The following statement creates a segment containing two PSUs, `mkt1` and `mkt2`:

```sql
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 filesystem 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 will dynamically reduce the PSU’s MAXSIZE value to its current size and start 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. Note that this number 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. The USED value also provides the lower limit on the new MAXSIZE in the CHANGE MAXSIZE option for the ALTER SEGMENT statement.

**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 filesystem(s) contains 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 segment(s) associated with that table.

Red Brick Warehouse re-uses space by row, which means that 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 used to be stored. The value for TOTALFREE only measures the space that has not yet been used, not the space freed up from deleting rows. If you have deleted large numbers of rows from your table, you may have more free space than the value of TOTALFREE indicates.

**Pseudocolumns**

Every user table in a Red Brick Warehouse database has three pseudocolumns: RBW_ROWNUM, RBW_SEGID, and RBW_SEGNAME. A pseudocolumn is a column that can be selected from like any other column but does not appear in the system tables.

**RBW_ROWNUM**

The RBW_ROWNUM pseudocolumn contains the row number for each row in a segment, where the first row in the segment is 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**

The RBW_SEGID pseudocolumn 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; it is the relative order of the values that allows you to determine the relative order of the segment.

**RBW_SEGNAME**

The RBW_SEGNAME pseudocolumn 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;
```

<table>
<thead>
<tr>
<th>RBW_ROWNUM</th>
<th>RBW_SEGNAME</th>
<th>RBW_SEGID</th>
<th>DOLLARS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DAILY_DATA1</td>
<td>0</td>
<td>34.00</td>
</tr>
<tr>
<td>1</td>
<td>DAILY_DATA1</td>
<td>0</td>
<td>60.75</td>
</tr>
<tr>
<td>2</td>
<td>DAILY_DATA1</td>
<td>0</td>
<td>270.00</td>
</tr>
<tr>
<td>3</td>
<td>DAILY_DATA1</td>
<td>0</td>
<td>36.00</td>
</tr>
<tr>
<td>0</td>
<td>DAILY_DATA2</td>
<td>1</td>
<td>348.00</td>
</tr>
<tr>
<td>1</td>
<td>DAILY_DATA2</td>
<td>1</td>
<td>123.25</td>
</tr>
<tr>
<td>2</td>
<td>DAILY_DATA2</td>
<td>1</td>
<td>121.50</td>
</tr>
<tr>
<td>3</td>
<td>DAILY_DATA2</td>
<td>1</td>
<td>56.00</td>
</tr>
</tbody>
</table>

RISQL>

**Note:** In this example, there are two rows that 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.
Adding Space to a Segment

A segment runs out of space only when all available space in the segment is allocated, as indicated by a value of zero for the corresponding TOTALFREE column in the RBW_SEGMENTS system table.

If a PSU runs out of space because the filesystem is full, the PSU MAXSIZE parameter is dynamically decreased to the file’s current size. If space is available in subsequent PSUs in other filesystems, the operation continues.

Note: PSUs are used sequentially in the order in which they were defined (by sequence number in the RBW_STORAGE.PSEQ column). The current PSU is the one being written to. More space can be added to a PSU only if the segment containing the PSU is unattached or if the PSU is the current PSU or a subsequent PSU. (In other words, you can never add space to a previous PSU once the next PSU is in use.)

When segments and PSUs are created, only the amount of storage specified by the INITSIZE value for each PSU is allocated immediately; the rest of the storage is not allocated until it is needed. Consequently, a filesystem can fill up before a PSU reaches the limit specified by its MAXSIZE parameter. If this happens, the warehouse server automatically checks the subsequent PSUs for available space—either already allocated but unused INITSIZE space or space on another filesystem.

If the warehouse 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 filesystem 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 advise you which filesystems ran out of space and which PSU is being used.

However, if there are no subsequent PSUs with space available in the segment, the operation terminates with an out-of-space error, and no changes are made to the MAXSIZE values.

Whenever a dynamic adjustment of MAXSIZE values occurs, the following conditions apply to the affected segment and PSUs:

- Each new adjusted MAXSIZE value is reflected in the MAXSIZE and USED columns in the RBW_STORAGE table and the TOTALFREE column in the RBW_SEGMENTS table.
• After a PSU has been 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 filesystem. (If space becomes available, the server can allocate that space to PSUs subsequent to the then-current PSU.)
• If a segment’s owning table 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 filesystem 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 filesystems on disks d1 and d2. PSU p1 is full and data is being written to PSU p2—the current PSU—when the filesystem on disk d1 fills up, although PSU p2 has used only 620 out of 680 blocks and PSU p3 is still empty. The warehouse 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 filesystem, in this case PSU p3, and then begins writing to the next PSU on a filesystem with space available, in this case PSU p4 on disk d2.

The following figure and table illustrate this scenario:

<table>
<thead>
<tr>
<th>PSU</th>
<th>INITSIZE</th>
<th>EXTENDSIZE</th>
<th>Original MAXSIZE</th>
<th>New MAXSIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>120</td>
<td>100</td>
<td>680</td>
<td>No change</td>
</tr>
<tr>
<td>p2</td>
<td>120</td>
<td>100</td>
<td>680</td>
<td>620</td>
</tr>
<tr>
<td>p3</td>
<td>120</td>
<td>100</td>
<td>680</td>
<td>120</td>
</tr>
<tr>
<td>p4</td>
<td>300</td>
<td>150</td>
<td>750</td>
<td>No change</td>
</tr>
</tbody>
</table>
Maintaining a Data Warehouse
Changing an Index Fill Factor

Because no new space is ever added to a PSU that precedes the current one, if space later becomes available in the filesystem on disk \( d1 \), that space will not be used for PSU \( p2 \) or PSU \( p3 \) (while they are attached to the current table or index).

If, however, the current PSU is PSU \( p4 \) on disk \( d2 \) and a PSU \( p5 \) exists in the filesystem 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 command after PSU \( p4 \) was defined). The following figure illustrates this scenario:

---

Changing an Index Fill Factor

You can change the fill factor used during the creation of new index nodes with the ALTER INDEX…CHANGE FILLFACTOR command. Altering the fill factor on a non-empty index 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.

The fill factor specifies the percentage of space filled in new index nodes as they are created. The amount of empty space left in each node affects the amount of space required when an index is built. It also affects the frequency with which nodes fill up completely and must be split, which affects performance. For more information about fill factors, refer to “Fill Factors” on page 4-21.
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.

**ALTER SEGMENT Operations**

The operations you can perform on segments are as follows:

- Attach or detach a segment from a table or its indexes, allowing you 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 has been loaded with data or restored from a backup. (OFFLINE, ONLINE)
- Remove all data from a segment, leaving it attached for reuse. (CLEAR)
- Change the range specification for segments, allowing you to change how data will be distributed among segments. (RANGE)
- Add additional PSUs to a segment to accommodate growing tables. (ADD STORAGE)
- Change the maximum size, the extend size, or the path of PSUs, allowing you to control their growth and move them as needed. (CHANGE MAXSIZE, CHANGE EXTENDSIZE, CHANGE PATH)
- Rename the segment as needed for easier identification. (RENAME)
- Specify a segmenting column, allowing you 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 (files) 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.

If a segment is the backup segment for use with the SQL-BackTrack for Red Brick Warehouse option, not all ALTER SEGMENT operations are valid. For information about the backup segment, refer to the SQL-BackTrack for Red Brick Warehouse User’s Guide.
Locking the Database First

Before using ALTER SEGMENT, you should lock the database with the LOCK DATABASE command to ensure exclusive access and unlock it with the UNLOCK DATABASE command when finished:

```sql
lock database;
alter segment ...;
unlock database;
```

The following sections describe the tasks you can perform with the ALTER SEGMENT command; for syntax descriptions and more detailed information, refer to the SQL Reference Guide.

Attaching and Detaching Segments

As tables increase or decrease in size or access patterns change, you can attach or detach segments for the tables or their indexes to meet your needs.

Note: You cannot attach segments to tables segmented by hashing.

A segment must exist before it can be attached. User-defined segments are created with a CREATE SEGMENT statement; default segments, with a CREATE TABLE or CREATE INDEX statement. Newly attached segments are automatically set to ONLINE mode.

To attach a segment to a table or its B-TREE or TARGET index, you must supply a valid range specifier based on the datatype 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 referenced (dimension) table’s segmenting column. The segmenting column of the referenced table is defined in the CREATE TABLE statement.

Note: RBW_ROWNUM values start with 0 (zero), not 1.

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 re-attachment to another object. Default segments are deleted.

If you want to move a segment from one end of the segment range to the other—for example, if you are dropping the oldest data in a time-cyclic database and want to reuse the segment for the new data—then you need to detach the segment and attach it with a new range specification.
Moving Entire Segments

You can move an entire segment from one location to another—for example, from one disk to another, from disk to optical storage, or from optical storage to disk—by using the ALTER SEGMENT...MIGRATE TO command. For the syntax of ALTER SEGMENT and for details on the MIGRATE TO clause, refer to the SQL Reference Guide.

Specifying a Segmenting Column

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

You cannot specify a segmenting column to change the segmenting column of a table or index or to assign a segmenting column to a table or index that is segmented by hash values.

Specifying a Range

You can change the range of data or index entries that reside in a segment by specifying a new range of data values or row IDs for the segment. Range modifications cannot leave any gaps or overlaps in the segmentation ranges for the table or index, nor can they require the movement of any row data or index entries from one segment to another.

Taking a Segment Offline or Online

Taking a segment offline makes the segment temporarily unavailable to users, while allowing partial access to the rest of the table. Offline mode is useful when you need to load additional data or restore a damaged segment but still want to keep the remainder of the table or index available to users. You must take a segment offline before you can detach it.

You can control how the partial unavailability of a table or index affects users with the SET PARTIAL AVAILABILITY options in the rbw.config file. For more information about partial availability, refer to “Partial Availability of Tables and Indexes” on page 2-10.
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 by using a TMU SYNCH command. For more information about the SYNCH command, refer to the Table Management Utility Reference Guide.

Note: 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 multi-segment 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.

Renaming a Segment

You can rename a user-created or default segment while it is attached to or detached from a table or index. The primary reason to rename a segment is to give it a meaningful, relevant name.

Changing PSU Sizes

You can change the MAXSIZE and EXTENDSIZE values for a PSU in order 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. Additional space is allocated 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). 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 8-8.
Changing PSU Location

You can change the location of a PSU in order to effectively manage disk storage as database tables, indexes, and access patterns change.

Caution: After using the CHANGE PATH option, you must perform a complete (level 0) backup before you can perform an incremental backup.

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 and 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 you should prevent user access while the operation is in progress by locking the affected table. If you are moving several PSUs, you should lock the database to ensure that you can complete all the changes to RBW_STORAGE before users attempt to access PSUs in transition.

Use caution when moving files, especially files of the same size and with similar names. If you do not correctly correlate the pathname and object, system corruption might result. If you change the path of the wrong object, but realize your mistake before running the TMU or making any modifications (INSERT, UPDATE, or DELETE) to that object, then you can reverse the operation by executing this option again with the correct name.

You must also use the CHANGE PATH command if you are moving or copying a table or database containing full pathnames. For example, if you want to move a table in a database, you can copy all the PSUs in the table to the new location and then use this option for each relocated PSU to specify the new location (pathname) for each PSU.

Verifying a Segment

You can determine whether a segment is damaged—that is, PSUs in the segment cannot be opened—and what the damage is with the ALTER SEGMENT...VERIFY option. After you have repaired the damage, use the VERIFY option as part of the recovery process to make sure the damage has been 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 filesystem is not mounted when a query is issued or if transient NFS errors occur, 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 actually examines each PSU for physical damage and takes more time to execute.)

Caution: Use the FORCE INTACT option only when you know the segment is undamaged. If you are uncertain, use the VERIFY option.
Recovering a Damaged Segment

Occasionally an operation on a table or index fails with a message that a segment is damaged, or the ALTER SEGMENT...VERIFY command might indicate that a segment is damaged. The damage is to a PSU (file) 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 (file); 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 multi-segment 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.

Procedure

Use the following procedure 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:

   ```sql
   select name, intact from rbw_tables where intact = 'N';
   select name, intact from rbw_indexes where intact = 'N';
   ```
To find which segment(s) is damaged, enter a query similar to:

```sql
select name, tname, iname, intact from rbw_segments
where intact = 'N';
```

2. If you are certain the PSU is not really damaged but was so marked for minor or transient access errors, you can use the `ALTER SEGMENT...FORCE INTACT` command for both online and offline segments to mark the segment and table or index intact.

```sql
alter segment seg_name of table table_name force intact;
alter segment seg_name of index index_name force intact;
```

**Note:** Use caution with this command because it does not verify the accessibility or integrity of the PSU. Its only benefit is that it permits you to avoid a time-consuming verification when you know there is no actual damage to a PSU.

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 multi-segment 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` command by entering the appropriate query:

```sql
alter segment seg_name of table table_name verify;
alter segment seg_name of index index_name verify;
```

**Note:** The `ALTER SEGMENT...ONLINE` command performs the same verification tasks and returns the same information as the `VERIFY` option; however, this command cannot be used on those segments already online.

5. After you have determined what the damage is, you need to repair it. Some problems such as insufficient permission on PSU files or files that are in the wrong directory are fairly 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. For information about restore operations with the SQL-BackTrack for Red Brick Warehouse option, refer to the *SQL-BackTrack for Red Brick Warehouse User’s Guide*.

6. Confirm that the damage is repaired by using the `ALTER TABLE...VERIFY` command.

7. If the segment is offline, set it to ONLINE mode with the `ALTER SEGMENT...ONLINE` command.
Managing Optical Storage

Optical storage devices provide a direct-access secondary storage that is faster than tape and less expensive than disk. The use of 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, and 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 expensive).

Because the access time is longer, you can specify whether queries and certain other 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 option).
- Specifying whether STAR indexes that reside on optical storage should be considered when an index is selected (IGNORE_OPTICAL_INDEXES option).

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. Red Brick Warehouse determines whether a segment is an optical segment by checking the filesystem type.

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 response to commands that require access to data or indexes in optical segments depends on both the setting for the OPTICAL_AVAILABILITY option and whether the command involves read or write operations. Commands affected by this option are:

- Read operations: SELECT statements and TMU UNLOAD statements
- Write operations:
  - ALTER TABLE and DROP TABLE statements
  - CREATE INDEX and ALTER INDEX statements
  - ALTER SEGMENT statements
  - INSERT, UPDATE, and DELETE statements
  - TMU LOAD DATA and REORG statements

This option can be set either in the rbw.config file or with SET commands.

You can determine or verify access behavior with respect to optical segments by checking the OPTICAL_AVAILABILITY entry in the RBW_OPTIONS table with a query similar to the following:

```sql
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
```

**Syntax**

To specify the query behavior for all sessions, enter a line in the rbw.config file using the following syntax:

```
OPTION OPTICAL_AVAILABILITY WAIT_NONE
```
To specify the query behavior for specific sessions, enter a SET command using the following syntax:

```
SET OPTICAL AVAILABILITY WAIT_NONE
```

The following settings apply as described to all the commands listed on page 8-20; other commands are not affected by this option.

**WAIT_NONE, WAIT NONE**
Specifies that an operation is to wait for row data or indexes stored in optical segments. No message regarding optical storage access is issued. The default is WAIT_NONE.

**WAIT_INFO, WAIT INFO**
Specifies that an operation is to wait for row data or indexes stored in optical segments. An informational message is issued stating that optical storage is being accessed.

**WAIT_WARN, WAIT WARN**
Specifies that an operation is to wait for row data or indexes stored in optical segments. A warning message is issued stating that optical storage is being accessed.

**SKIP_INFO, SKIP INFO**
Specifies that for read operations any optical segments containing row data or indexes are to be skipped. An informational message is issued stating that such segments were skipped.

For write operations, this option is ignored and the command is processed as if PRECHECK were specified.

**SKIP_WARN, SKIP WARN**
Specifies that for read operations any optical segments containing row data or indexes are to be skipped. A warning message is issued stating that such segments were skipped.
For write operations, this option is ignored and the command is processed as if PRECHECK were specified.

**PRECHECK**
Specifies that for any operation a check for optical segments is to be performed before the operation is performed. If any optical segments are encountered, an error message is issued and the processing terminates.

**ERROR**
Specifies that for read operations the statement is to be processed, but if any optical segments are encountered, an error message is to be issued and the processing terminates. Note that an operation might process for a significant amount of time before encountering an optical segment.

For write operations, this option is ignored and the statement is processed as if PRECHECK were specified.

**Examples**

The following examples illustrate how to specify access behavior with optical segments:

```
rbw.config file entry:  OPTION OPTICAL_AVAILABILITY INFO
SET command:          set optical availability error
```

**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. If a query is processed and an error or warning message is issued stating that an optical index was accessed, and 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.

**Note:** In most cases, frequently used indexes will not reside in optical segments: Storing an index on slower optical devices defeats the purpose of the index.
You can determine whether indexes with optical segments are considered during index selection by checking the IGNORE_OPTICAL_INDEXES entry in the RBW_OPTIONS table with a query similar to the following:

```sql
select substr(option_name, 1, 30), substr(value, 1, 12)
from rbw_options
where option_name like 'IGNORE_OPTICAL%'
    and username = CURRENT_USER;
```

**Syntax**

To specify the use of indexes in optical segments for all sessions, enter a line in the `rbw.config` file using the following syntax:

```
OPTION IGNORE_OPTICAL_INDEXES OFF
```

To specify the use of indexes that are stored in optical segments for specific sessions, enter a SET command using the following syntax:

```
SET IGNORE_OPTICAL_INDEXES OFF
```

**OFF**

Specifies that all indexes, even those in optical segments, are to be considered in selecting the best index. If an index with an optical segment is determined to be the best choice, the setting for the OPTICAL_AVAILABILITY option controls how the operation proceeds. The default is OFF.

**ON**

Specifies that only indexes stored entirely on non-optical storage are to be considered in selecting the best index. If no applicable index meets this criterion, an error message is issued and the operation fails.

**Examples**

The following examples illustrate how to specify the use of indexes stored in an optical segment:

```
rbw.config file entry: OPTION IGNORE_PARTIAL_INDEXES OFF
SET command: set ignore partial indexes on
```
Altering Tables

You can make the following changes to a table with the ALTER TABLE command:

- Add or drop a column.
- Change a column’s name.
- Change a column’s 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

You cannot alter a view.

Red Brick Systems 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, it is a good idea to update your CREATE TABLE statements to reflect the changes you made so you can re-create the table from scratch, if necessary.

For information about high performance backup and restore operations with the SQL-BackTrack for Red Brick Warehouse option, refer to the SQL-BackTrack for Red Brick Warehouse User’s Guide.

The following sections describe the tasks you can perform with the ALTER TABLE command and how to recover from an interrupted ALTER TABLE operation. For syntax descriptions and more detailed information, refer to the SQL Reference Guide.

Adding and Dropping Columns

You can add or drop one or more columns with a single ALTER TABLE statement.

When a new column is added with ALTER TABLE...ADD COLUMN, it is added at the end of the table; if multiple columns are added, they are added in the order named. Adding a column does not affect a view because column references are resolved when the view is created.
When a new column is dropped with ALTER TABLE…DROP COLUMN, its data is removed from the table; the operation is not reversible. A column cannot be dropped if it is part of a primary key, part of a foreign key, part of an index key, or referenced by a view.

When you add or drop a column, you can either specify that the changes should take place in the existing table location (IN PLACE) or provide another location (one or more segments) in which the altered table will be built. If the table is built in another location, when the changes are complete, the original table is removed. The advantage to building the table in another location is that recovery is easier if the operation is interrupted; however, it requires space for two tables. In either case, there must be enough space to hold the modified table.

**Changing a Column’s Name**

You can change a column’s name with an ALTER TABLE…ALTER COLUMN…RENAME statement. The new name must be unique in the table.

**Changing a Column’s Default Value**

You can change a column’s default value with an ALTER TABLE…ALTER COLUMN…SET DEFAULT statement. The default value is the value that is loaded into a column when the input record was empty or did not contain valid input data. This default value is also used with the TMU Automatic Row Generation option.

For more information about Automatic Row Generation, refer to the *Table Management Utility Reference Guide*. 
Maintaining a Data Warehouse
Altering Tables

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. If you need to add more rows or segments than are allowed by these parameters, you must 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, before you change MAXSEGMENTS and MAXROWS PER SEGMENT, check the MAXSIZE_ROWS value for that table in the system table RBW_TABLES. If the new MAXSEGMENTS value multiplied by the new MAXROWS PER SEGMENT value is greater than MAXSIZE_ROWS, the table will outgrow the segment before you reach the new limit. At this point, you must use ALTER SEGMENT to increase the MAXSIZE value of the segment.

Changing How Referential Integrity Is Maintained

You can change how referential integrity is maintained for a given table with an ALTER TABLE...ALTER COLUMN...ON DELETE statement. If a foreign key in a table references another table, the ON DELETE clause lets you specify whether:

• A delete action in the referenced table cascades into the referencing table.
• No delete action is performed on either table.

For more information about referential integrity and cascaded deletes, refer to “Delete Operations and Cascaded Deletes” on page 2-35.

Changing a Column’s Datatype

If you want to change a column’s datatype—for example to change a TINYINT to a SMALLINT column or to change a numeric external to a date datatype—you can do so by following these steps:

1. Use ALTER TABLE to add a new column of the desired datatype; 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 or dropped from a schema. There are many reasons why this may 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 may necessitate a new type of product that is different enough to warrant a new table. You may want to associate these new tables with another table by adding a foreign key reference from an existing table. You can do this with 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. Note that this is mandatory for a multi-column foreign key.

You must meet the following requirements in order to add a foreign key:

- The referenced table must exist.
- A primary key index must exist on the referenced table (note that 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 datatype 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.

**Note:** For large tables, the referential integrity checking may take some time.

Similarly, you may have a referenced table that is no longer needed and want to drop both the table and the foreign key reference to it. You can do this with an `ALTER TABLE...DROP CONSTRAINT` operation on the existing table. No STAR index must exist on the constraint, otherwise the `ALTER TABLE` statement will fail.

For the complete syntax of the `ALTER TABLE` command, refer to the SQL Reference Guide.
Recovering from an Interrupted ALTER TABLE Operation

Occasionally an ALTER TABLE operation is interrupted before it completes. Several possible recovery options exist. The option you choose depends both on the cause of the interruption and whether the table was being altered with the IN PLACE option.

Recovering the Table

After dealing with the cause of the interrupt, you have three choices for recovering the table, depending on the state of the table when the interrupt occurred:

- You can RESUME the alter operation and let it run to completion. This alternative is useful when much of the work has already been done.
- You can RESET the table to its original state and re-issue the original ALTER TABLE command. This alternative makes sense when very little of the work has been done. If, however, you are using the IN PLACE option, you cannot RESET the table. You must either RESUME the operation or RESTORE the table from a backup.
- You can restore the database or just the segments containing the table from a backup. If the backup was current, the table will be restored to its state before the ALTER TABLE command was issued. If the backup is not current, the restored table might not reflect its latest state.

Note: If you are altering the table IN PLACE, you should have a current database backup before beginning an ALTER TABLE command.

Interruptions: Their Causes and Prevention

Interruptions can be caused by:

- Privilege violations
  To prevent interruptions from privilege violations, make sure that the user executing the operation has the necessary file system read/write privileges, as well as the required database authority and object privileges.

- Out-of-space errors
  The ALTER TABLE command accurately 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. However, the maximum space defined is not necessarily the space
allocated, so even though the ALTER TABLE command calculates that enough space is available, in reality, 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.

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 really available before beginning an ALTER TABLE command.

• Cancel (Control-C) or kill commands, system crashes, or power failures

Preventing these types of interruptions is more difficult. Although you cannot always anticipate every situation, planning is the key to avoiding these types of interruptions, or being able to deal effectively with them when they happen. Always keep regular back-ups 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 it is absolutely necessary, especially if you have to force a cancel using an operating system utility (for example, kill -9). Such forced cancel operations can leave the data in an inconsistent state.
Copying or Moving a Database

If you want to make a copy of a database for training or testing or to move a database to a new location, you can do so using combinations of UNIX, SQL, and TMU commands. However, because pathnames can be stored as either full or relative pathnames in the system tables, simply copying the files to a new location is not always sufficient to ensure that the copy pathnames point to the copy and not the original database.

**Note:** The Enterprise Control and Coordination option includes the `rb_cm` copy management utility, which facilitates the movement of data among databases. For more information on the `rb_cm` copy management utility, refer to the *Table Management Utility Reference Guide*.

A full pathname begins with a slash (/); a relative pathname is any pathname that does not begin with a slash. The warehouse server constructs relative pathnames as 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 file (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, enter the following query:

```sql
select segname, pseq, location from rbw_storage where substr(location, 1, 1) = '/';
```

This command returns the names of all files (PSUs) that use full pathnames.
After you have determined whether the database to be copied or moved contains any full pathnames, use the following procedures to ensure a successful copy or move operation.

**Note:** A database built on a specific platform can be copied only to other locations on the same platform type; for example, a database built on a Solaris platform cannot be copied or moved to an HP 9000 platform or vice versa. To copy or move a database from one platform to another, use the TMU UNLOAD...EXTERNAL procedure. Then rebuild the database on the new platform using the data and the CREATE TABLE and LOAD DATA statements generated by the UNLOAD EXTERNAL 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 (Preferred)**

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.

Note: Users with the Enterprise Control and Coordination option can perform a variation of this procedure using the rb_cm copy management utility. The rb_cm utility pipes UNLOAD output to LOAD input, allowing you to move table data over a network without ever writing to tape or to disk. For more information about the rb_cm utility, refer to the Table Management Utility Reference Guide.

Method 2

This method is riskier because there are more opportunities for error in entering pathnames.

1. Verify that the new location contains enough space to hold the database and that the redbrick user can write to the new location.

2. As the redbrick user, make a new directory for the database in the new location.

3. Copy all files from the existing database directory to the new directory.

4. Copy all files with full pathnames to the new directory.

5. As a database user with DBA authorization or the creator of the segment, use the ALTER SEGMENT...CHANGE PATH command to change the path for each segment that uses a full pathname to the new pathname (the copy’s name).

6. Add a logical name for the copied database to the rbw.config file.

Moving a Database with Only Relative 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 and then delete the original files.

4. Change the logical database name in the rbw.config file to the new location.
Moving a Database with Full 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. 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 command to change the path to the new location.

7. Change the logical database name in the rbw.config file to the new location.
## Modifying the Configuration File

The configuration file, `rbw.config`, is created when the warehouse software is installed, using information provided during the installation procedure. This information is used by the warehouse server and by the TMU.

You, as the redbrick user, can edit the `rbw.config` file (using any standard text editor) as conditions at your site change. Because changes you make to this file have various effects on the processes that use it, refer to the following table to determine what other actions you must take.

<table>
<thead>
<tr>
<th>If you change:</th>
<th>Action required:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHMEM or MAPFILE SERVER</td>
<td>Stop and restart warehouse daemon.</td>
</tr>
<tr>
<td>MAX_SERVERS</td>
<td>1. Stop warehouse daemon.</td>
</tr>
<tr>
<td></td>
<td>2. Use <code>ipcrm</code> to remove prior memory segment or reboot.</td>
</tr>
<tr>
<td></td>
<td>3. Restart warehouse daemon.</td>
</tr>
<tr>
<td>CLEANUP_SCRIPT LOGFILE_SIZE</td>
<td>Stop and restart warehouse daemon.</td>
</tr>
<tr>
<td>QUERYPROCS TOTALQUERYPROCS</td>
<td></td>
</tr>
<tr>
<td>ADMINADVISOR_LOGGING</td>
<td></td>
</tr>
<tr>
<td>SERVER_NAME MESSAGE_DIR LOCALE</td>
<td>Do not change.</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>Stop and restart server-monitoring daemon (<code>rbw.servermon</code>).</td>
</tr>
<tr>
<td>License keys</td>
<td>No action needed. Current TMU and server processes do not recognize, but new processes will.</td>
</tr>
</tbody>
</table>
### Maintaining a Data Warehouse

#### Modifying the Configuration File

If you change:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Action required:</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE_GROUP</td>
<td>No action needed. Current server processes do not recognize, but new processes will.</td>
</tr>
<tr>
<td>ROWS_PER_SCAN_TASK</td>
<td></td>
</tr>
<tr>
<td>ROWS_PER_FETCH_TASK</td>
<td></td>
</tr>
<tr>
<td>ROWS_PER_JOIN_TASK</td>
<td></td>
</tr>
<tr>
<td>FORCE_SCAN_TASKS</td>
<td></td>
</tr>
<tr>
<td>FORCE_FETCH_TASKS</td>
<td></td>
</tr>
<tr>
<td>FORCE_JOIN_TASKS</td>
<td></td>
</tr>
<tr>
<td>FORCE_HASHJOIN_TASKS</td>
<td></td>
</tr>
<tr>
<td>CROSS_JOIN</td>
<td></td>
</tr>
<tr>
<td>ARITHABORT</td>
<td></td>
</tr>
<tr>
<td>ALLOW_POSSIBLE_DEADLOCKS</td>
<td></td>
</tr>
<tr>
<td>DEFAULT_DATA_SEGMENT</td>
<td></td>
</tr>
<tr>
<td>DEFAULT_INDEX_SEGMENT</td>
<td></td>
</tr>
<tr>
<td>TEMPORARY_DATA_SEGMENT</td>
<td></td>
</tr>
<tr>
<td>TEMPORARY_INDEX_SEGMENT</td>
<td></td>
</tr>
<tr>
<td>OPTION_ADVISOR_LOGGING</td>
<td></td>
</tr>
<tr>
<td>PRECOMPUTED_VIEW parameters</td>
<td></td>
</tr>
<tr>
<td>SEGMENTS</td>
<td></td>
</tr>
<tr>
<td>IGNORE_PARTIAL_INDEXES</td>
<td></td>
</tr>
<tr>
<td>PARTIAL_AVAILABILITY</td>
<td></td>
</tr>
<tr>
<td>QUERY_TEMPACE parameters</td>
<td></td>
</tr>
<tr>
<td>RESULT_BUFFER</td>
<td></td>
</tr>
<tr>
<td>RESULT_BUFFER_FULL_ACTION</td>
<td></td>
</tr>
<tr>
<td>TMU_BUFFERS</td>
<td>No action needed. Current TMU processes do not recognize, but new processes will.</td>
</tr>
<tr>
<td>TMU_OPTIMIZE</td>
<td></td>
</tr>
<tr>
<td>TMU_CONVERSION_TASKS</td>
<td></td>
</tr>
<tr>
<td>TMU_INDEX_TASKS</td>
<td></td>
</tr>
<tr>
<td>TMU_SERIAL_MODE</td>
<td></td>
</tr>
<tr>
<td>AUTOROWGEN</td>
<td></td>
</tr>
<tr>
<td>FILLFACTOR parameters</td>
<td>No action needed. Current server and TMU processes do not recognize, but new processes will.</td>
</tr>
<tr>
<td>INDEX_TEMPSPACE parameters</td>
<td></td>
</tr>
<tr>
<td>PASSWORD parameters</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** In the `rbw.config` file, those 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 or TMU SET commands, refer to Appendix B, “Configuration File.”

The RBW_OPTIONS system table lists current values for all parameters that you can change; it is updated whenever a SET command changes a value during a session. The values displayed are the values that apply to the current session.
Monitoring and Controlling Warehouse Processes

Red Brick Warehouse includes tools to help monitor and control the warehouse daemons and the server processes. You can also monitor queries through the USAGE ROUTINE event in Red Brick log file.

In addition to these tools, which are described in the following sections, other monitoring and control features are available with the Enterprise Control and Coordination option and are described in Chapter 7, “Managing Database Activity with the Enterprise Control and Coordination Option.”

Warehouse Daemon

You can use the following scripts to monitor and control the Red Brick Warehouse daemon processes (rbwapid, rbwlogd, and rbwadmd).

<table>
<thead>
<tr>
<th>Script</th>
<th>Description</th>
</tr>
</thead>
</table>
| rbw.start | Starts rbwapid, rbwlogd, and rbwadmd daemons as background processes. You must be logged in as the redbrick user to execute rbw.start. **Example**:  
  redbrick_dir/bin/rbw.start config_path RB_HOST |
| rbw.show  | Displays information about the active rbwapid, rbwlogd, and rbwadmd daemons and their associated server processes. **Example**: 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. **Example**: redbrick_dir/bin/rbw.stop RB_HOST |

For more information about running these scripts and automatic startup procedures, refer to the Installation and Configuration Guide.

Findserver Utility

You can use a utility named rbw.findserver (Findserver) to inquire about a specific warehouse user’s session or all active warehouse sessions. For each session it finds, the Findserver utility lists the following information:

- Warehouse username
- Database path
- Server process ID
- Date and time that a user session started
The \texttt{rbw.findserver} program works in conjunction with a server-monitoring daemon named \texttt{rbw.servermon}. If the Findserver utility is enabled, the \texttt{rbw.servermon} daemon is started automatically by the \texttt{rbw.start} script and runs whenever the \texttt{rbwapid} daemon is running. The monitoring daemon runs in the background and maintains a private record of information about active server sessions.

\textbf{Enabling Findserver}

To enable server monitoring and the Findserver utility, you must add the following line to your \texttt{rbw.config} file:

\begin{verbatim}
RBWMON INTERVAL n
\end{verbatim}

The RBWMON keyword enables server monitoring. The INTERVAL parameter specifies the time in seconds (\textit{n}) between maintenance updates of the monitor log file; a value of 120 seconds is recommended.

If the RBWMON keyword is not present in the \texttt{rbw.config} file, no server monitoring is performed and the \texttt{rbw.findserver} utility does not function. You must add the RBWMON keyword; it is not added automatically by the installation script.

\textbf{Using Findserver}

The \texttt{rbw.findserver} program must be run as the \texttt{redbrick} user, and the \texttt{RB_CONFIG} environment variable must be set correctly. The syntax is:

\begin{verbatim}
rbw.findserver [db_username]
\end{verbatim}

If \texttt{db_username} is specified, only session(s) for that user are listed. If \texttt{db_username} is omitted, all active warehouse sessions are listed.

\textbf{Stopping the rbw.servermon Daemon}

Once started, the \texttt{rbw.servermon} daemon runs as long as the \texttt{rbwapid} daemon is running. When the \texttt{rbwapid} daemon stops, the \texttt{rbw.servermon} daemon will stop at the next scheduled maintenance update of the monitor log file, as determined by the INTERVAL value.

\textbf{Note:} Use the \texttt{rbw.stop} utility to stop the \texttt{rbwapid} daemon.
Log Files

Several log files are available in the redbrick directory to allow you to monitor the various warehouse processes:

- **install.log**, a file that contains a software identification number and warehouse installation date.
- **rbwapid.log**, a file that 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.
- In addition, the redbrick_dir/logs directory contains non-text and binary rbwacct and rbwlog files that contain detailed activity and accounting information. These files, which are not text files, are part of the Enterprise Control and Coordination option; they are also used by Red Brick Customer Support.

Query Logging

The entire SQL statements for queries are logged through the USAGE ROUTINE event of the Red Brick Enterprise Control and Coordination log file. You can enable query logging by issuing the following command:

```
RISQL> alter system change logging level usage routine;
```

In order to enable query logging, the log daemon must be running. When you enable SQL logging, all queries that Red Brick Warehouse processes are written to the log file. Depending on how active your system is, this might make your log files grow in size rapidly, so if you enable query logging, be sure to provide ample disk space for your log files.

For more information on logging, refer to Chapter 7, “Managing Database Activity with the Enterprise Control and Coordination Option.” For the complete syntax of the ALTER SYSTEM command, refer to the SQL Reference Guide.
Enabling Licensed Options

Some features of Red Brick Warehouse are options that require a license key to enable their use. These features are included as part of the Red Brick Warehouse software. If an option is purchased at the same time as Red Brick Warehouse, it is installed and enabled during the normal installation process (using the license key provided on a separate information sheet). If your site purchases one of these options after you have installed the warehouse software, you can run the installation script and select the License Option menu, which will enable the option when you provide the license key.

If you have purchased the option but do not have a license key, you can contact Red Brick Customer Support Services for this information.

Determining Version Information

If you need to contact the Red Brick Customer Support Center, determine the complete version identification information for the Red Brick Warehouse software before you call.

You can obtain warehouse version information as follows:

- By viewing the rbwapid.log file. The warehouse 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:

  ```sql
  select rbw_version from rbw_tables ;
  ```

A similar query can be run on any table. It returns the version number, one line for each row in the table, so choose a table with a small number of rows.
Deleting Database Objects and Databases

As users’ requirements evolve, you might need to remove tables from an existing database or delete an entire database from a Red Brick Warehouse installation. This section contains the following information:

- How to drop database objects (tables, indexes, views, synonyms, segments, macros, and roles) from a database with DROP statements.
- How to delete databases with the warehouse utility *rb_deleter*.

**Note:** A user is removed from a database by issuing a REVOKE CONNECT statement.

Dropping Database Objects

You can drop database objects from a database by using the appropriate SQL DROP statement:

- DROP INDEX
- DROP MACRO
- DROP ROLE (only applicable 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. Columns are dropped from a table with the ALTER TABLE statement.

For complete syntax descriptions, refer to the *SQL Reference Guide*. 
Indexes

An index is dropped from the database with the DROP INDEX command.

You can specify whether to keep or drop any user-defined segments associated with the index. If the segments are kept, they are detached from the table associated with the index and are available for reuse. If the segments are dropped, all PSUs in the segments are also removed. The default behavior is to keep user-defined segments. You can override the default behavior globally by setting 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.

Macros

A macro is dropped with the DROP MACRO command; a temporary macro is also dropped when the database connection or server session is terminated. After a macro is dropped, references to it are no longer expanded.

The use of the PUBLIC and TEMPORARY keywords in a DROP MACRO statement must be the same as in the CREATE MACRO statement.

Roles

The predefined system roles—DBA, RESOURCE, and CONNECT—cannot be dropped; you can only revoke the role from specific users.

The Enterprise Control and Coordination option includes the ability to define custom, or user-defined, roles in addition to the predefined system roles. A user-defined role is dropped with the DROP ROLE command. 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 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 command. A segment must be taken offline (with the ALTER SEGMENT command) and detached before you can drop it with a DROP SEGMENT command.

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 \textit{rbw.config} file or with a SET command from the command line. For more information about setting SEGMENTS parameter, refer to “Setting Segment and Partial Availability Behavior” on page 9-15.
- 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

A synonym is dropped with the DROP SYNONYM command; this command 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 view so that they do not reference the synonym.

Tables

A table is dropped from the database with the DROP TABLE command. This command also drops any indexes on the table and removes any privileges or synonyms that reference the table. This command can be used only for base tables, not views or synonyms.

You can specify whether to keep or drop any user-defined segments associated with the table. If the segments are kept, they are detached from the table and are available for reuse. If the segments are dropped, all PSUs in the segments are also removed. The default behavior is to keep user-defined segments. You can override the default behavior globally by setting the SEGMENTS parameter in the \textit{rbw.config} file or locally for a given table in the DROP TABLE statement.
Unlike user-defined segments, default segments are always dropped when the table is dropped.

**Note:** 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 command will fail. You must first detach and drop the damaged segments before you can drop the table. However, a DROP TABLE...DROPPING SEGMENTS command will succeed 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**

A view is dropped from the database with the DROP VIEW command. Dropping a view does not affect underlying tables.

Before dropping a view, you must drop any views that reference it.

**Deleting a Database**

To delete a database, use the warehouse utility `rb_deleter`. This utility removes any default files created by a warehouse server in the database directory. This utility is typically executed only by the warehouse 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 warehouse 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 `rm` commands.
**Maintaining a Data Warehouse**

*Deleting Database Objects and Databases*

---

**Procedure**

To delete a database:

1. Optionally, access the database with DBA authority. 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 (files) not located in the database directory or the DEFAULT_DATA_SEGMENT and DEFAULT_INDEX_SEGMENT directories. Record their names for use in Step 4.

2. Exit from the database.

3. Log in as the `redbrick` user.

4. Invoke `rb_deleter`, which is located in the `$RB_CONFIG/bin` directory, by entering:
   
   ```
   $ rb_deleter pathname
   ```

   at the system prompt, where `pathname` is a full directory path specification to the database directory, such as `/disk1/db_sales_92`.

5. 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 (files) that were not in the main database directory. (See Step 1.) Check for directories referenced by CREATE SEGMENT commands 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.

7. Log out as the `redbrick` user.
Tuning a Warehouse for Performance

This chapter describes various ways in which you can customize Red Brick Warehouse to improve performance at your site. Because customizing is by definition site-specific, in most cases only general guidelines are provided. The suggested course of action is to use Red Brick Warehouse initially in its default configuration. As you gain experience with the warehouse software, the system environment, and the work load at your site, you might find areas where you would like to improve performance.

This chapter includes the following sections:

• Specifying Parameters with rbw.config File Entries or SET Commands
• Setting Temporary Space Parameters
• Setting the Result Buffer for Long-Running Queries
• Setting Segment and Partial Availability Behavior
• Setting the Index Fill Factor
• Creating Additional Indexes
• Understanding Red Brick Query Processing
• TARGETjoin Query Processing
• Using Synonyms to Control Fact-to-Fact Joins
• Making SQL-Based Improvements

You can turn on statistics reporting, which provides some information useful in analyzing query performance, with the SET STATS command.

For information about improving query performance through parallel processing, refer to Chapter 10, “Tuning a Warehouse for Parallel Query Processing.” For information about optional features, refer to Chapter 7, “Managing Database Activity with the Enterprise Control and Coordination Option.” To customize the warehouse environment for your site and user community, refer to your operating system documentation.
Specifying Parameters with rbw.config File Entries or SET Commands

Many parameters can be set either in the rbw.config file or with SET commands. Entries in the rbw.config file affect all sessions for that warehouse, whereas SET commands affect only the session during which they are executed. Most SET commands can be entered interactively anywhere you can enter SQL statements, or they can be entered in the warehouse, database, or user .rbwrc files. SET commands for the TMU can be entered 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.

Note: The syntax diagrams for SET commands in this chapter include the terminating semicolon (;) required by the TMU, the RISQL Entry Tool, and the RISQL Reporter. Not all SQL entry tools require this terminator.

Setting Temporary Space Parameters

Index-building operations and complex queries can require large amounts of temporary space to store intermediate results. With the temporary space parameters, you can define the directories to be used when temporary space is needed, the threshold at which intermediate results spill from memory to disk, and the maximum amount of disk space to be used for temporary space. Separate parameters control index-building operations and query operations.

This section describes the following:

• Parameters used to control temporary space
• Procedure by which temporary space is allocated
• Syntax for temporary space parameters
• Procedure to determine the current values of these parameters
• Removal of temporary files

For information about how to estimate temporary space requirements, refer to “Planning for Temporary Space Requirements” on page 4-32.
### Temporary Space Parameters

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

<table>
<thead>
<tr>
<th>Temporary Space Parameters</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>For Index Building Temporary Space</strong></td>
<td></td>
</tr>
<tr>
<td><strong>TUNE Parameter/SET Command</strong></td>
<td><strong>Function</strong></td>
</tr>
<tr>
<td>TUNE INDEX_TEMPSPACE_DIRECTORY, SET INDEX TEMPSPACE DIRECTORIES*</td>
<td>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.*</td>
</tr>
<tr>
<td>TUNE INDEX_TEMPSPACE_THRESHOLD, SET INDEX TEMPSPACE THRESHOLD</td>
<td>Specifies size at which index-building operations spill from memory to disk.**</td>
</tr>
<tr>
<td>TUNE INDEX_TEMPSPACE_MAXSPILLSIZE, SET INDEX TEMPSPACE MAXSPILLSIZE</td>
<td>Specifies maximum amount of temporary space that can be allocated to a spill during an index-building operation.**</td>
</tr>
</tbody>
</table>

| **TUNE Parameter/SET Command** | **Function** |
| TUNE QUERY_TEMPSPACE_DIRECTORY, SET QUERY TEMPSPACE DIRECTORIES | Specifies temporary space directory or directories to be used by query processing. Multiple rbw.config file entries can be made, one directory per entry.* |
| TUNE QUERY_MEMORY_LIMIT, SET QUERY MEMORY LIMIT | Specifies the maximum memory size for queries, after which they spill from memory to disk. |
| TUNE QUERY_TEMPSPACE_MAXSPILLSIZE, SET QUERY TEMPSPACE MAXSPILLSIZE | Specifies maximum amount of temporary space that can be allocated to service a spill during a query operation. |

*The warehouse server and TMU automatically spread temporary space usage evenly across the designated directories.

**For operations that build multiple indexes in parallel, these parameter values are split equally among each index; that is, the value is a total value for the entire operation.
The QUERY_TEMPSPACE parameters are used whenever queries are processed; they are not used for TMU operations. The INDEX_TEMPSPACE parameters are used whenever indexes are built or modified by CREATE INDEX statements or TMU LOAD DATA or REORG operations.

You can specify values for these parameters in the rbw.config file or with SET commands, which can be entered in .rbwrc files or anywhere you can enter SQL commands. You can also specify INDEX_TEMPSPACE parameters for TMU operations with SET statements in a TMU control file. Note that the values in the rbw.config file are processed before any .rbwrc files are read and before the TMU runs, so the rbw.config file settings can be overridden by .rbwrc file commands or TMU control files.

**How Temporary Space Is Allocated**

The set of directories comprising each temporary space can be thought of as a circular buffer, and the set of files created in each directory for a particular query or index-building process can be thought of as a slice of the buffer, a slice that can grow to but not exceed MAXSPILLSIZE.

**Random Directory Sequence**

When a process exceeds the threshold size and spills to disk, a random sequence of these temporary space directories is determined for that process; this sequence determines the order in which the directories are used by that process. For example, if five temporary space directories—\(d_1, d_2, d_3, d_4, d_5\)—are defined, a sequence for one spill operation might be \(d_3, d_1, d_4, d_2, d_5\), while the sequence for another concurrent spill operation might be \(d_1, d_2, d_5, d_3, d_4\).

The following figure illustrates this concept of the temporary space directories comprising a circular buffer, one for queries and one for index-building operations. Two slices based on random directory sequences are shown for each buffer.
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.
For both query and index-building spills, the directories are used in the random sequence determined for each spill. However, in directories containing multiple files, the files are used in the following sequences:

- For query temporary directories: The files are used one per directory in the random sequence, thus distributing the load among the directories, repeating the sequence as needed.
- For index-building directories: All files in a directory are used, before proceeding to the next directory in the random sequence.

**Full and Out-of-Space Conditions**

A slice of query or index temporary space is *full* when either of the following conditions is met: (1) The sum of the current sizes of all files that comprise that slice equals the MAXSPILLSIZE value for that space, or (2) the slice cannot be further extended (by extending a file) without exceeding the MAXSPILLSIZE value for that space.

A slice of query or index temporary space is *out of space* when the slice is not yet full, but all files that comprise that slice are in use and no more disk space is available to extend the last file in the sequence of files.

Full and out-of-space conditions affect the various operations as follows:

- **Queries:** If a slice of query temporary space becomes full or runs out of space, the query is aborted.

- **Indexes:**
  - If a slice of index temporary space—online or offline—runs out of space, the operation is aborted.
  - If a slice of *online* index temporary space becomes full, the contents are merged into the index, the temporary space is emptied, and the index build continues, reusing the space.
  - If a slice of *offline* index temporary space becomes full, the operation is aborted.
**TEMPSPACE Syntax**

To specify parameters that apply to all sessions, enter a line in the `rbw.config` file, using the following syntax:

```
TUNE
  QUERY_TEMPSPACE_DIRECTORY — dir_path
  QUERY_MEMORY_LIMIT — value
  QUERY_TEMPSPACE_MAXSPILLSIZE — size
  INDEX_TEMPSPACE_DIRECTORY — dir_path
  INDEX_TEMPSPACE_THRESHOLD — value
  INDEX_TEMPSPACE_MAXSPILLSIZE — size
```

To specify the parameters for specific sessions, enter a SET command using the following syntax:

```
SET
  QUERY TEMPSPACE DIRECTORIES 'dir_path'
  MAXSPILLSIZE — size
  RESET

SET
  QUERY MEMORY LIMIT — value
  DEFAULT

SET
  INDEX TEMPSPACE DIRECTORIES 'dir_path'
  THRESHOLD — value
  MAXSPILLSIZE — size
  RESET

SET
  TEMPSPACE RESET ;
```

**DIRECTORY dir_path, DIRECTORIES 'dir_path', ...**

Specifies a directory or a set of directories that are to be used for temporary files; `dir_path` must be a full pathname. To define a set of directories using entries in the `rbw.config` file, enter multiple lines. The order in which the directories are specified has no effect because the order in which they are used is random (determined internally) and no user control is possible.

If no temporary space directories are defined, the default directory is `/tmp`. 
THRESHOLD value
This parameter specifies the amount of memory used before writing the intermediate results from an index-building operation to disk.

For index-building operations involving multiple indexes, this threshold value is allocated equally among the indexes being built.

The size must be specified as either kilobytes (K) or megabytes (M) by appending K or M to the number. Note that no space is allowed between the number and the unit identifier (K, M). For example: 1024K, 500M.

The threshold value must be specified before the corresponding MAXSPILLSIZE value is specified; it must precede the MAXSPILLSIZE entry in the rbw.config file.

A value of 0 causes files to be written to disk after the first 200 rows or index entries.

The default value for INDEX_TEMPSPACE_THRESHOLD is 10 megabytes (10M).

MAXSPILLSIZE size
Specifies the total maximum amount of temporary space per operation. For an index-building operation involving multiple indexes, this space is allocated equally among the indexes being built. For query operations, however, the entire value is allocated to each query and to each of its subqueries, if any.

The size must be specified as kilobytes (K), megabytes (M), or gigabytes (G) by appending K, M, or G to the number. Note that no space is allowed between the number and the unit identifier (K, M, G). For example: 1024K, 500M, 8G.

The default MAXSPILLSIZE value is 1 gigabyte. The maximum MAXSPILLSIZE value is 2047 gigabytes.

RESET
Resets the query or index TEMPSPACE parameters to the values specified in the rbw.config file. If neither QUERY nor INDEX is specified, all TEMPSPACE parameters are reset.
QUERY MEMORY LIMIT value

The QUERY MEMORY LIMIT value must be specified as kilobytes (K), megabytes (M), or gigabytes (G) by appending K, M, or G to the number. Note that no space is allowed between the number and the unit identifier (K, M, or G). For example: 2048K, 500M, 3G.

The default value of QUERY MEMORY LIMIT is 50 megabytes (50M). The range is from 2 megabytes (2M) to 4 gigabytes (4G).

Usage

In addition, use the following guidelines when setting temporary space parameters:

- Always set the QUERY_MEMORY_LIMIT value before setting the QUERY_TEMPSPACE_MAXSPILLSIZE value.
- Never set the QUERY_MEMORY_LIMIT to a larger value than the maximum data segment size allocated to a process by the operating system or to a value larger than the QUERY_TEMPSPACE_MAXSPILLSIZE value.
- In general, smaller QUERY_MEMORY_LIMIT values are better in multi-user environments. A value that is too large can cause excessive paging or higher physical memory usage.
- Use tools such as vmstat or sar to monitor memory usage.

For additional information about how queries use temporary space, refer to “Setting QUERY_MEMORY_LIMIT” on page 9-11.

Examples

The following examples illustrate entries in the rbw.config file that apply to all sessions:

```
TUNE QUERY_TEMPSPACE_DIRECTORY /disk1/qtemp
TUNE QUERY_TEMPSPACE_DIRECTORY /disk2/qtemp
TUNE QUERY_TEMPSPACE_DIRECTORY /disk3/qtemp
TUNE QUERY_MEMORY_LIMIT 2M
TUNE QUERY_TEMPSPACE_MAXSPILLSIZE 8G
```

The following examples illustrate SET commands that can be used to change parameters for a specific session:

```
SET INDEX TEMPSPACE DIRECTORIES '/disk1/itemp', '/disk2/itemp', '/disk3/itemp'
SET INDEX TEMPSPACE THRESHOLD 2M
SET INDEX TEMPSPACE MAXSPILLSIZE 3G
```
Setting Temporary Space Parameters

The following example illustrates how to reset the INDEX_TEMPSPACE parameters to the values specified in the rbw.config file, leaving the QUERY_TEMPSPACE parameters set to their current values:

```
SET INDEX TEMPSPACE RESET
```

The following example illustrates how to reset all TEMPSPACE parameters to the values specified in the rbw.config file:

```
SET TEMPSPACE RESET
```

Determining Current Values

To determine the current values for QUERY_TEMPSPACE and INDEX_TEMPSPACE parameters, query the RBW_OPTIONS system table.

Example

To determine the QUERY_TEMPSPACE parameters in effect for the current session, enter a query similar to the following:

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

Removing Temporary Files

Spill files are usually removed by the warehouse server (rbwsvr) or the TMU as soon as possible. If, however, the 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) executes a cleanup script specified by the following entry in the rbw.config file:

```
RBWAPI CLEANUP_SCRIPT script_name
```

where `script_name` is the name of the cleanup script for your system.

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 cleanup script does not find and remove spill files from locations specified with a SET command during a server or TMU session; these files must be removed manually.
Setting QUERY_MEMORY_LIMIT

When setting the QUERY_MEMORY_LIMIT parameter, consider the following:

- Be aware of the amount of physical memory available on your system.
- Do not consistently over-commit your memory.
- Know the number of users on the system and the number of users who will be issuing queries during the same time period.
- Keep the paging rate down.

Red Brick Warehouse allocates memory to each query. The size of the memory allocation is allowed to grow from 1MB, which is the default size to which the buffer cache is initialized, to the value specified by the QUERY_MEMORY_LIMIT parameter, after which it spills to disk. The value of the QUERY_MEMORY_LIMIT parameter applies to all users on the system. Therefore, if QUERY_MEMORY_LIMIT is set to 10MB and you have 10 users on your system, the minimum amount of memory used for query processing is 10 times 1MB, or 10MB, and the potential amount of memory that can be used for query processing is 10 times 10MB, or 100 MB. Each user’s memory consumption grows above 1MB only when a query is issued that requires more than 1MB of memory.

If Red Brick Warehouse runs out of query-processing memory before each user reaches the value specified in the QUERY_MEMORY_LIMIT parameter, the operating system will start swapping memory to disk. You want to avoid this condition because it will slow down performance. When setting the QUERY_MEMORY_LIMIT parameter, choose a value high enough so that most queries can run in memory, but not so high that the operating system has to swap to disk.

There are trade-offs to consider, however. For example, setting the QUERY_MEMORY_LIMIT parameter to a higher value might avoid spilling to disk in 90% of your queries, thus making those queries run many times faster. But the remaining 10% might be large queries run simultaneously by several users, potentially over-committing your memory resources and forcing the operating system to swap to disk, and ultimately slowing everyone down. You must consider the trade-offs and set the QUERY_MEMORY_LIMIT value accordingly.
The amount of memory used also depends on the number of concurrent users executing queries; this is another factor to consider when setting the value for QUERY_MEMORY_LIMIT. The types of queries that are being executed is also a factor: queries that browse a small dimension table tend to use small amounts of memory, while queries that join two fact tables tend to use very large amounts of memory. It is sometimes difficult to predict what the users will do, but you can find trends by monitoring the database activity.
Setting the Result Buffer for Long-Running Queries

The result buffer is an area of temporary space that is used to hold query results that have completed processing on the server but are waiting for the client to request them. This 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 lock(s) on the table(s) remain until all of the results either leave the Red Brick Warehouse server or are placed in the buffer. With client tools that require user input to receive more than a certain amount of data, the read locks will 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.

There are two SQL SET commands and two corresponding rbw.config TUNE parameters to 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 Syntax

To specify the size of the buffer that holds query results until the client is ready to receive them, enter an SQL SET command with the following syntax:

```
SET RESULT BUFFER value [K|M|G] [UNLIMITED];
```

The corresponding rbw.config file syntax is as follows:

```
TUNE RESULT_BUFFER value [K|M|G] [UNLIMITED];
```
Tuning a Warehouse for Performance
Setting the Result Buffer for Long-Running Queries

value
Specifies an integer value, which must be followed by a \( K \) (kilobytes), an \( M \) (megabytes), or a \( G \) (gigabytes).

UNLIMITED
Indicates that there is no limit on the amount buffered. Note that the buffer uses the same space allocated with the QUERY TEMPSPACE MAXSPILLSIZE parameter, so when the RESULT BUFFER parameter is set to unlimited, the buffer size is still limited by the QUERY TEMPSPACE MAXSPILLSIZE value.

Setting a value of 0 for the RESULT BUFFER parameter specifies that no results will be buffered.

RESULT BUFFER FULL ACTION Parameter Syntax

To specify the behavior when the results buffer size specified with the SET RESULT BUFFER command is reached, enter a SET command with the following syntax:

\[ \text{SET} \] \text{RESULT BUFFER FULL ACTION} \text{ABORT} ; \text{PAUSE} ; \]

The corresponding rbw.config file syntax is as follows:

\[ \text{TUNE} - \text{RESULT_BUFFER_FULL_ACTION} \text{ABORT} \text{PAUSE} ; \]

The value ABORT indicates that the query will abort when the buffer size is reached. The value PAUSE indicates that when the buffer size is reached the query will pause until the client requests more data.

Example

The following SET commands specify a result buffer of 100 megabytes for the current session and force the query to abort when that buffer size is reached:

```
set result buffer 100M;
set result buffer full action abort;
```
Setting Segment and Partial Availability Behavior

Segment creation and deletion behavior, as well as query behavior against partially available tables, are determined on a global basis by option settings in the rbw.config file. These option settings can be overridden for the current session by SET commands entered on the command line.

Location of Default Segments

You can specify a directory location for all default row data segments and another for all default index segments (that is, those segments not specifically created with a CREATE SEGMENT statement).

Syntax

To set a default directory for default data or index segments for all sessions, enter a line in the rbw.config file using the following syntax:

```
 OPTION [ DEFAULT_DATA_SEGMENT ] dir_path
 [ DEFAULT_INDEX_SEGMENT ]
```

To set a default directory for default data or index segments for specific sessions, enter a SET command using the following syntax:

```
 SET DEFAULT [ DATA SEGMENT STORAGE PATH ] dir_path
 [ INDEX ]
```

`dir_path`
Pathname of the directory in which all default row data segments or all default index segments are to be stored.

If no default directory is specified, all default segments are stored in the database directory, as defined in the rbw.config file or with the RB_PATH environment variable.
**Examples**

The following examples illustrate how to specify a default location for default data and index segments.

- **rbw.config file entries:**
  ```plaintext
  OPTION DEFAULT_DATA_SEGMENT /dsk1/dsegs
  OPTION DEFAULT_INDEX_SEGMENT /dsk1/ixsegs
  ```

- **SET commands:**
  ```plaintext
  set default data segment storage path 'dsk1/dsegs';
  set default index segment storage path 'dsk1/ixsegs';
  ```

**Segment Drop Behavior**

You can specify whether a user-defined segment should be dropped or kept if the table or index in that segment is dropped. (Default segments are always dropped.)

**Syntax**

To specify segment drop behavior for user-defined segments for all sessions, enter a line in the `rbw.config` file using the following syntax:

```
OPTION SEGMENTS KEEP DROP
```

To specify segment drop behavior for user-defined segments for specific sessions, enter a SET command using the following syntax:

```
SET SEGMENTS KEEP DROP;
```

**KEEP**

Specifies the segment is to be kept for reuse even though the table or index it contains is dropped. The segment can be reused for another table or index. The default behavior is KEEP for user-defined segments.

**Note:** 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 segment(s) is detached and dropped.

**DROP**

Specifies the segment is to be dropped if the table or index it contains is dropped.
Examples

The following examples illustrate how to specify drop behavior for user-defined segments.

rbw.config file entry:  
```
OPTION SEGMENTS DROP
```

SET command:  
```
set segments drop ;
```

Query Behavior on Partially Available Tables

You can specify how queries behave against partially available tables. In this context, a partially available table is a table with either one or more offline row data segments or one or more offline index segments for the index to be used for that query.

Whether indexes with offline segments are considered when the best index for a query is selected depends on the setting of the IGNORE PARTIAL INDEXES option. If a query is processed and an error or warning message is issued stating that a partial index was used, and you know that other fully available but less optimal indexes exist, you can set the IGNORE PARTIAL INDEXES option to ON to force the query processing to use the fully available index.

Syntax

To specify the query behavior for all sessions, enter a line in the rbw.config file using the following syntax:

```
OPTION — PARTIAL_AVAILABILITY — PRECHECK
  INFO
  WARN
  ERROR
```

To specify the query behavior for specific sessions, enter a SET command using the following syntax:

```
SET — PARTIAL — AVAILABILITY — PRECHECK ;
  INFO
  WARN
  ERROR
```
Precheck
Specifies that table availability is to be checked before the query is processed. If a table is only partially available, an error message is issued and the query is not processed. The default is PRECHECK.

Info
Specifies that the query is to be processed and the results returned, even if a row data or index segment is unavailable. If the results would be different if the table were fully available (that is, if the server needs to access an offline segment to process the query), an informational message to this effect is issued along with the results.

Warn
Same as INFO, but the message is a warning, not an informational message.

Error
Specifies that the query is to be processed even if a row data or index segment is unavailable. If the results would be different if the table were fully available, no results are returned and an error message is issued. The query might process for a significant amount of time before determining that the results might be affected.

Examples
The following examples illustrate how to specify query behavior with partially available segments.

rbw.config file entry:  
OPTION PARTIAL_AVAILABILITY INFO

SET command:  
set partial availability error ;

Use of Partially Available Indexes
You can specify whether partially available indexes should be considered when the best strategy for processing a given query is selected.

Syntax
To specify use of partially available indexes for all sessions, enter a line in the rbw.config file using the following syntax:

OPTION IGNORE_PARTIAL_INDEXES [ON | OFF]
To specify use of partially available indexes for specific sessions, enter a SET command using the following syntax:

\[
\text{\texttt{SET IGNORE PARTIAL INDEXES ON}} \quad \text{OFF}
\]

**ON**
Specifies that only fully available indexes are to be considered in selecting the best index for a query. If no applicable index is fully available, an error message is issued and the query fails.

**OFF**
Specifies that all indexes, even partially available ones, are to be considered in selecting the best index for a query. If a partially available index is determined to be the best choice for a given query, the setting for the PARTIAL AVAILABILITY option controls how the query is processed.

**Examples**

The following examples illustrate how to specify the use of partially available indexes.

- **rbw.config** file entry:  
  
  \[
  \text{OPTION IGNORE\_PARTIAL\_INDEXES OFF}
  \]

- **SET command:**  
  
  \[
  \text{set ignore partial indexes off ;}
  \]
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 into the index without causing the nodes (blocks) to split; such splits slow down incremental loads. Indexes built with a fill factor of less than 100% require more storage but can provide better incremental load, update, and insert performance.

The default fill factor is 100%. 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, then 100% is the appropriate fill factor. If, however, 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. You can specify fill factors for individual indexes with the CREATE INDEX or ALTER INDEX statements. All fill factors default to 100%, 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% fill factors, ignoring any specified fill factors.

Example

The following figure illustrates an index with a fill factor of 100% and the same index with a fill factor of 66%, after an initial load.
At 100%, 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%, 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 line(s) in the `rbw.config` file using the following syntax:

```
    FILLFACTOR PI    x    
    FILLFACTOR STAR   y    
    FILLFACTOR SI     z    
```

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 command).

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.
Tuning a Warehouse for Performance
Setting the Index Fill Factor

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. This fill factor can be changed on an index-by-index basis with the ALTER INDEX command.

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.

To Find 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:

```sql
select name, fillfactor
from rbw_indexes
where name = 'index_name';
```

**Note:** Actual use of the fill factor depends, however, on whether the TMU OPTIMIZE mode is set.

To Decide Whether to Change Default Fill Factors

To decide whether to change a default fill factor (100% 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 whether it is sufficient to reduce load performance enough to justify the additional space required by filling nodes less than 100% full.

**Note:** This discussion assumes the expected growth throughout the index is uniform. If the growth is expected at the ends of the index or in new segments of a segmented index, the default fill factor is the best choice.
You can see how the fill factor affects the amount of storage required for an index by considering the following formula, used to calculate how many elements are stored in each index node:

\[
\text{Elements per index node} = \left( \frac{\text{fillfactor}}{100} \times \frac{8.172}{\text{keysize} + 6} \right)
\]

where each index node is 8,172 bytes and corresponds to one 8-kilobyte file system block minus 20 bytes overhead, and \(\text{keysize}\) 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. The number of 8-kilobyte blocks required to store the index for various fill factors is calculated using the formulas in “Estimating Index Sizes” on page 4-21; the results are shown in the following table:

<table>
<thead>
<tr>
<th>Fill Factor</th>
<th>Elements Per Node</th>
<th>Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>81</td>
<td>627</td>
</tr>
<tr>
<td>50</td>
<td>409</td>
<td>124</td>
</tr>
<tr>
<td>100</td>
<td>817</td>
<td>63</td>
</tr>
</tbody>
</table>

If this table is to be loaded once with no anticipated additions, a fill factor of 100 is the best choice.

If this table is expected 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% of what you expect it to contain, a fill factor of 10 is reasonable.

**Note:** If you choose a low fill factor, the resulting indexes can be very large and therefore have a negative effect on query performance.
Creating Additional Indexes

When you create a table containing a primary key, a B-TREE index is automatically created on the primary key column(s). Creating additional indexes often improves query performance. In determining whether to create additional indexes, keep in mind that the improvement in performance must be weighed against the additional space required to store the index and the time required to build and update the index when changes are made to the tables upon which the index is based.

For fact tables that are queried with constraints on foreign key columns, you can improve query performance by creating STAR indexes using the foreign keys that will be constrained. The order of the foreign keys in a STAR index will affect performance on particular queries; the best performance is gained from a STAR index whose leading foreign key matches the query constraint.

If all the columns in the primary key are also columns in foreign keys, then creating a STAR index that includes all the primary key columns might make the primary key B-TREE index redundant. In this case, you can save disk space and processing overhead by dropping the primary key B-TREE index after creating the primary key STAR index.

You can also improve performance for multi-table joins where the tables are related by primary key/foreign key relationships by enabling TARGETjoin processing. To enable TARGETjoin, create indexes on the foreign key columns of the referencing (fact) table. For information on TARGETjoin processing, refer to “TARGETjoin Query Processing” on page 9-40 and to Chapter 4, “Planning a Database Implementation.”

If you have columns that contain weakly selective constraints, consider creating TARGET indexes to greatly improve query performance for queries constraining on those columns. For detailed information on TARGET indexes, refer to “TARGET Indexes” on page 4-8.

Whenever a dimension table contains a foreign key that is not also a primary key (that is, the table references an outboard table), consider either creating an additional B-TREE or TARGET index on each foreign key column for better performance or creating a STAR index.

The EXPLAIN command can help you determine what indexes are being used for a given query. For information about EXPLAIN, refer to “EXPLAIN Command” on page 9-37.

For more information about using additional indexes to improve performance and their requirements, refer to “Determining When to Create Additional Indexes” on page 4-3.
Understanding Red Brick Query Processing

Red Brick Warehouse evaluates a query and automatically decides the best way to process it, based on the indexes that are available. The server goes through many phases while processing a query. The following sections list the join algorithms Red Brick Warehouse uses, illustrate the phases of query processing, and discuss how to use the EXPLAIN command. This command shows you what phases a particular query will go through, and what steps you can take to optimize query performance.

Join Algorithms

Generally, the queries that might require tuning are queries that involve joins. Red Brick Warehouse uses the following algorithms to process joins:

- STARjoin
- TARGETjoin
- B-TREE one-to-one match (nested loops join)
- Hybrid hash join
- Naive one-to-one match (Cross join)

The warehouse server’s choice of join method depends on which indexes are available. At runtime, the server evaluates the query and makes decisions about the query execution plan based on the following criteria:

1. If the appropriate STAR index exists to join the tables, the query uses the STARjoin algorithm. An appropriate STAR index contains some or all of the keys that are constrained in the query.

2. If the appropriate STAR index exists to join the tables, and if TARGET or B-TREE indexes exist on the foreign key columns in the referencing (fact) table, the query will use either STARjoin or TARGETjoin algorithm, depending on which has the best indexes available for the join operation.

3. If the appropriate STAR index does not exist but TARGET indexes exist on the foreign key columns in the referencing (fact) table, then the query will use the TARGETjoin algorithm.

4. If the appropriate STAR index does not exist but either a B-TREE or a TARGET index is present over the joining columns, that index is used with the B-TREE one-to-one match algorithm. If both a B-TREE and a TARGET index exist, the server chooses the best index for the join operation.

5. If no indexes are present over the joining columns and the join is an equijoin (the query constraints are equality conditions), the hybrid hash join algorithm is used.
6. If no indexes are present over the joining columns and the join is not an equijoin, the cross join algorithm is used. The cross join algorithm calculates all possible combinations of the joining columns (the Cartesian product). Therefore, cross joins are disallowed unless the OPTION CROSS JOIN parameter is set to ON. This requirement ensures that users do not issue cross join queries inadvertently, by omitting a join condition, for example.

Note: These criteria are simplified for the purpose of this discussion; there are many other variables that add complexity to the choices the Red Brick Warehouse server makes.
The following figure illustrates the server’s decision-making process for joining tables:

```
Start

STAR Index?

Yes

TARGET or B-TREE Indexes on foreign keys?

Yes

TARGET or TARGETjoin

End

No

No

TARGETjoin

End

No

No

No

B-TREE or TARGET Index?

Yes

Equijoin?

Yes

Hybrid Hash Join

End

No

Is this the query you meant to write?

Yes

Rewrite the query

End

No

No Indexes

End

No

No

Naive 1-1 Match (Cross Join)

End
```

End
Operator Model

Red Brick Warehouse uses an object-oriented operator model to evaluate a query. Stages of the query are broken down into operators, which are elemental pieces of the query execution plan. Each operator has its own portion of work to do before passing the execution along to the next operator.

The output of the EXPLAIN command lists the operators that are used in a given query execution plan. By understanding what each operator does and which indexes are being used in a query, you can devise ways of improving query performance by one or more of the following means:

- Re-writing the query
- Adding or dropping indexes
- Changing the schema design
- Changing the values of tuning parameters
- Modifying the physical layout of your database and database files

For more information on the EXPLAIN command, refer to the section “EXPLAIN Command” on page 9-37.

The following is an alphabetical list of the operators with a brief description of each, as well as a brief description of the fields associated with each operator that appears in the EXPLAIN output.

B-TREE 1-1 Match

Given a key value, this operator looks it up in the index.

Join Type
Lists the type of join performed by the operator.

Possible values: InnerJoin, LeftOuterJoin, RightOuterJoin, FullOuterJoin. For information on the different types of joins, refer to the SQL Reference Guide.

Index
Lists the name of the index used in the operation and the name of the table on which the index is defined.
**B-TREE Scan**

This operator scans an index or a range of keys in an index.

**Reverse Order**

For multi-column indexes, indicates if the index is being scanned in the key order (FALSE) of the index or in the reverse key order (TRUE) of the index. It is more efficient to scan an index in the order of its keys than in the reverse order. If Reverse Order = T, you might improve performance by creating an additional index with the key order reversed in the CREATE INDEX statement.

Possible values: TRUE, FALSE.

**Predicate**

A predicate is a restricted condition on the query, for example:

```...where promo_type = 400```

Predicates can have one condition or multiple conditions.

**Start-Stop Predicate**

A start-stop predicate is a predicate with a restrictive range, such as a conjunction in the WHERE clause. This allows only a restricted amount of the index to be scanned, which increases the efficiency of the index scan, thus improving query performance. For example, the following predicate:

```... where col1 > 5 and col1 < 8```

restricts the range on Col1 between 5 and 8. If the domain of Col1 is all integers and there is an index defined on Col1, this allows a very efficient, restricted scan of the index, which is good for query performance.

**Note:** The Red Brick Warehouse server uses start-stop predicates whenever possible, even if there are no conjunctions in the WHERE clause.

**Bit Vector Sort**

This operator sorts the RBW_SEGID and RBW_ROWNUM bits from the pseudo-columns containing that information.
Choose Plan

This operator represents a dynamic decision point made at runtime. Only one of the choices will be executed at runtime.

Num Prelims
Lists the number of preliminary operations that are performed before the choice is made as to which plan to use. These preliminary operations are performed regardless of which plan is selected.

Num Choices
Lists the number of choices the server will choose from at runtime.

Type
Lists the type of plan.
Possible values: Unknown, General, STARjoin.

Delete
This operator performs the DELETE operation.

Delete All
Indicates (FALSE) if there is a search condition on the DELETE operation, as specified in the WHERE clause of the DELETE command, and (TRUE) if there is no search condition.
Possible values: TRUE, FALSE.

Table
The name of the table from which the rows will be deleted.

Constraint Name
The name of the constraint that references another table involved in the DELETE operation.
**Delete Cascade**

This operator finds the data to be deleted in the referenced table for a DELETE CASCADE operation.

**Delete All**
Indicates (FALSE) if there is a search condition on the DELETE operation, as specified in the WHERE clause of the DELETE command, and (TRUE) if there is no search condition.

Possible values: TRUE, FALSE.

**Table**
The name of the table from which the rows will be deleted.

**Delete Refcheck**
This operator checks whether the DELETE operation will violate referential integrity. If it finds a referential integrity violation, the DELETE operation is disallowed.

**Delete All**
Indicates (FALSE) if there is a search condition on the DELETE operation, as specified in the WHERE clause of the DELETE command, and (TRUE) if there is no search condition.

Possible values: TRUE, FALSE.

**Table**
The name of the table from which the rows will be deleted.

**Constraint Name**
The name of the constraint that references another table involved in the DELETE operation, used to check referential integrity.

**Exchange**
This operator splits an operation for parallelism.

**Exchange Type**
Lists the type of operation to be parallelized.

Possible values: Unknown, Functional Join, STARjoin, Table Scan, Upper Hash 1-1 Match, Lower Hash 1-1 Match.
Execute

This operator coordinates the interaction between operators. Execute is the first operator in an EXPLAIN report.

Table Locks

Possible values: Read, Write.

Functional Join

Given the segment ID and row number, this operator reads the row.

Number of Tables

The number of tables involved in the operation and their names.

General Purpose

This operator is used for dynamic substitutions such as SELECT COUNT(*)

Operation

Describes the operation to be performed by the operator.

Count: Optimization for COUNT (*) processing based on table size when there is no predicate, grouping, HAVING clause, etc.

Textsize: Processing of ‘SELECT @@textsize’ variable selection, used by some query tools.

Hash 1-1 Match

This operator performs a hybrid hash join.

Join Type

Lists the type of join performed by the operator.

Possible values: InnerJoin, LeftOuterJoin, RightOuterJoin, FullOuterJoin. For information on the different types of joins, refer to the SQL Reference Guide.
**Hash AVL Aggregate**

This operator is used for aggregate and GROUP BY processing.

**Grouping**
Indicates if GROUP BY processing is used.
Possible values: TRUE, FALSE.

**Distinct**
Indicates if SELECT DISTINCT processing is used.
Possible values: TRUE, FALSE.

**Insert**
This operator is used to insert into a table.

**Table**
The name of the table being inserted into.

**Mode**
Indicates whether the INSERT operation is written to disk as soon as each row is received (*Immediate*) or whether the data is stored in a buffer and then written to disk all at once when the operation has completed (*Delayed*).

**Values** indicates an INSERT operation using the VALUES keyword, as follows:

```
insert into table1 values (a, b);
```

Possible values: Values, Delayed, Immediate.

**Merge Sort**
This operator is used to perform sorts.

**Distinct**
Indicates if SELECT DISTINCT processing is used.
Possible values: TRUE, FALSE.

**Naive 1-1 Match**
This operator is used to compute the Cartesian product (cross join) of two tables.
**RISQL Calculate**

This operator is used to process RISQL display functions.

**Simple Merge**

This operator takes two input lists and combines them.

**Sort 1-1 Match**

This operator performs a matching sort of two sorted lists.

**Match type**

Possible values: union, intersect, except.

**STARjoin**

This operator performs the STARjoin processing.

**Join Type**

Lists the type of join performed by the operator.

Possible values: InnerJoin. For information on the different types of joins, refer to the *SQL Reference Guide*.

**Num Facts**

Lists the number of fact (referencing) tables involved in the operation.

**Num Potential Dimensions**

Lists the number of potential dimension (referenced) tables involved in the operation.

**Fact Table**

Lists the name of the fact table(s) involved in the operation.

**Potential STAR Indexes**

For each fact table, lists the names of the potential STAR indexes involved in the STARJoin operation.

**Dimension Table(s)**

Lists the names of the dimension (referenced) tables that participate in the STAR index.
Subquery

This operator is used to process a subquery.

Scalar
Indicates whether the subquery is a scalar subquery.
Possible values: TRUE, FALSE.

Correlated
Indicates whether the subquery is a correlated subquery.
Possible values: TRUE, FALSE.

Table Scan

This operator scans a table.

Table
Indicates the name of the table being scanned.

Predicate
Indicates the predicate (the restrictive condition) on the table being scanned.

TARGETjoin

This operator performs the TARGETjoin processing, which efficiently joins tables related by primary key/foreign key relationships.

Table
Indicates the name of the table involved in the operation.

Predicate
Indicates the predicate (the restrictive condition) on the operation.

Num Indexes
Indicates the number of indexes involved in the operation.

Index(s)
Indicates the name of each index involved in the operation.
**TARGET Scan**

This operator performs TARGET index processing and processing of certain INTERSECT and UNION operations that use a B-TREE index.

**Table**
Indicates the name of the table involved in the operation.

**Predicate**
Indicates the predicate (the restrictive condition) on the operation.

**Num Indexes**
Indicates the number of indexes involved in the operation.

**Index(s)**
Indicates the name of each index involved in the operation.

**Update**

This operator performs an update operation.

**Mode**
Indicates whether the UPDATE operation is written to disk as soon as each row is processed (Immediate) or whether the data is stored in a buffer and then written to disk all at once when the operation has completed (Delayed).

Possible values: Delayed, Immediate.

**Table**
Indicates the name of the table being updated.

**Virtual Table Scan**
This operator manages internal results stored cached in virtual memory.
**EXPLAIN Command**

The EXPLAIN command provides output detailing a query execution path. EXPLAIN tells you which operators are used and which indexes are used. You can use EXPLAIN to understand how a query is going to execute. Based on the information you learn, you might decide to create and/or drop indexes to tune the query for better performance.

Some parts of the query execution path are determined dynamically during query execution, and in those cases, EXPLAIN tells you the possible paths. EXPLAIN also shows where parallelism is used in the part of the report on the Exchange operator. Then, when you run a query with SET STATS INFO enabled, the degree of parallelism that the query actually used for each Exchange operator is printed in informational messages.

In the EXPLAIN report, each operator is indicated by dashes (—) followed by its name in capital letters. Each operator name is followed by an ID number that is unique to the query. This ID is used to track which operator is doing which part of the query-processing work. After the ID number, there is descriptive information about the actions the operator is performing. For information on the operators, refer to the section “Operator Model” on page 9-28. The following example shows the report on the Execute operator from a typical EXPLAIN report:

```
- EXECUTE (ID: 0) 2 Table locks (table, type): (PROMOTION, Read), (SALES, Read)
```

Note: Each EXPLAIN query-processing report begins with the word “EXPLANATION” followed by the Execute operator.

**Example**

The following example shows the output from EXPLAIN for a simple join of the Sales and Promotion tables from the Aroma database.

```
RISQL> explain select sales.promokey, dollars > from promotion, sales > where sales.promokey = promotion.promokey;
EXPLANATION
 [ - EXECUTE (ID: 0) 2 Table locks (table, type): (PROMOTION, Read), (SALES, Read)
  ----- BTREE 1-1 MATCH (ID: 1) Join type: InnerJoin; Index(s): [Table: PROMOTION, Index: PROMOTION_PK_IDX]
  ----- TABLE SCAN (ID: 2) Table: SALES, Predicate: <none>
 ]
RISQL>
```
In this example, the report shows there are read locks on the Promotion and Sales tables; the tables will be joined using a B-TREE one-to-one match join with the Promotion_pk_idx B-TREE index.

**Example**

The following example adds a constraint to the previous example.

```
RISQL> explain select sales.promokey, dollars
> from promotion, sales
> where sales.promokey = promotion.promokey
> and promotion.promo_type = 400;
```

```
EXPLANATION
[ - EXECUTE (ID: 0) 2 Table locks (table, type): (PROMOTION, Read),
(SALES, Read)
--- CHOOSE PLAN (ID: 1) Num prelims: 1; Num choices: 2; Type:
  STARjoin;

  Prelim: 1; Choose Plan [id : 1] {
    BIT VECTOR SORT (ID: 2)
    -- TABLE SCAN (ID: 3) Table: PROMOTION, Predicate:
      (PROMOTION.PROMO_TYPE) = (400)
  }

  Choice: 1; Choose Plan [id : 1] {
    EXCHANGE (ID: 4) Exchange type: Functional Join
    -- FUNCTIONAL JOIN (ID: 5) 1 tables: SALES
    ---- EXCHANGE (ID: 6) Exchange type: STARjoin
    ------ STARJOIN (ID: 7) Join type: InnerJoin, Num facts: 1, Num
    potential dimensions: 4, Fact Table: SALES, Potential STAR Indexes:
    SALES_STAR_IDX; Dimension Table(s): PERIOD, PRODUCT, STORE,
    PROMOTION
  }

  Choice: 2; Choose Plan [id : 1] {
    EXCHANGE (ID: 8) Exchange type: Table Scan
    -- FUNCTIONAL JOIN (ID: 9) 1 tables: PROMOTION
    ---- BTREE 1-1 MATCH (ID: 10) Join type: InnerJoin; Index(s):
      [Table: PROMOTION, Index: PROMOTION_PK_IDX]
    ------ TABLE SCAN (ID: 11) Table: SALES, Predicate: <none>
  }
]
RISQL>
```
In addition to the read locks from the previous example, this plan shows a choice that the server will make at runtime. Choice 1 is a STARjoin using the STAR index Sales_star_idx, and choice 2 is a B-TREE one-to-one match (nested loops join) using the Promotion_pk_idx B-TREE index. There are several places where parallelism might occur (shown by the Exchange operators): in Choice 1, the STARjoin can be parallelized, and in Choice 2, the B-TREE one-to-one match can be parallelized.

If you now run the query with SET STATS INFO enabled, you get the following results:

```
RISQL> set stats info;
RISQL> select sales.promokey, dollars
> from promotion, sales
> where sales.promokey = promotion.promokey
> and promotion.promo_type = 400;
PROMOKEY   DOLLARS
172  348.00
172  128.00
...
165  79.75
** STATISTICS ** (500) Compilation = 00:00:00.12 cp time,
00:00:00.41 time, Logical IO count=93
** STATISTICS ** (1457) EXCHANGE (ID: 4) Parallelism over 1 times
High: 4 Low: 4 Average: 4.
** STATISTICS ** (1457) EXCHANGE (ID: 6) Parallelism over 1 times
High: 1 Low: 1 Average: 1.
** STATISTICS ** (1458) CHOOSE PLAN (ID: 1) Choice: 1 was chosen 1
  times.
** STATISTICS ** (1459) CHOOSE PLAN (ID: 1) STARjoin on 1 tables was
done 1 times.
** STATISTICS ** (1460) CHOOSE PLAN (ID: 1) usedIndex SALES_STAR_IDX
of Table SALES 1 times for STARjoin.
** STATISTICS ** (500) Time = 00:00:02.73 cp time, 00:00:03.01 time,
Logical IO count=145
** INFORMATION ** (256) 392 rows returned.
RISQL>
```

From this output, you can tell that Choice 1, the STARjoin, is what was actually executed, the degree of parallelism that was used on the Exchange (ID=4) was 4, and the degree of parallelism on Exchange (ID=6) was 1.
Tuning a Warehouse for Performance
TARGETjoin Query Processing

**TARGETjoin Query Processing**

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

- How to Use TARGETjoin Processing
- When to Use TARGETjoin Processing
- Examples
- Reading EXPLAIN Output on a TARGETjoin Query
- Summary and Recommendations

**How to Use TARGETjoin Processing**

This section explains what you need to do to enable TARGETjoin processing and under what conditions Red Brick Warehouse uses TARGETjoin processing to run a query.

**Create TARGET or B-TREE Indexes on Foreign Keys of Fact Table**

To enable TARGETjoin processing, you create indexes (TARGET or B-TREE) on the foreign keys of the fact table. For single-column foreign keys, create TARGET indexes on the foreign key column. For multi-column foreign keys, create B-TREE indexes on the concatenation of the foreign key columns. The reason for creating B-TREE indexes on multi-column foreign keys is that TARGET indexes are single-column indexes and therefore cannot be created on a multi-column foreign key. After you create the indexes, run queries and TARGETjoin processing is selected automatically when it is the best choice.

**Note:** For the purpose of describing the TARGETjoin functionality, consider a fact table to be any table that references other tables by foreign key references. This includes fact tables in a simple star schema that reference dimension tables by foreign key references, and it also includes dimension tables that reference outboard tables by foreign key references.
Rules for TARGETjoin Query Processing

To understand how Red Brick Warehouse chooses whether to use TARGETjoin processing, consider the following cases:

• One or more STAR index(es) are present
• No STAR index is present

When There Is At Least One STAR Index On the Fact Table

If there is at least one STAR index covering any of the keys constrained in a query, a TARGETjoin query plan is generated based on the STARjoin query plan if the following conditions are satisfied:

• At least one STAR index exists covering some or all of the constrained keys of the query.
• TARGET or B-TREE indexes exist on all the single-column fact table foreign keys that are both constrained in the query and keys of the qualifying STAR index(es).
• B-TREE indexes exist on all the multi-column fact table foreign keys that are both constrained in the query and keys of the qualifying STAR index(es).

If these conditions are not satisfied, a query cannot use TARGETjoin processing, but the query will complete using another join method. When these conditions are satisfied, the optimizer considers both STARjoin and TARGETjoin processing and automatically chooses the best index and join method for each query.

When There Are No Qualifying STAR Indexes on the Fact Table

If there is no STAR index covering the keys constrained in a query, a TARGETjoin query plan is generated when the following conditions are satisfied:

• No STAR index exists covering any of the constrained keys of the query.
• The query joins two or more dimension tables to the fact table.
• At least two of the constrained dimensions have TARGET indexes on the corresponding fact table foreign keys (B-TREE index on multi-column foreign keys).

Note: Dimensions that reference a single column foreign key must have a TARGET index on that foreign key or they will not be considered for a TARGETjoin query plan, even if a BTREE index exists on the foreign key.
If these conditions are not satisfied, the query uses another join method. When these conditions are satisfied, the query plan generated does not include STARjoin as a possible join method; it includes a choice of a table scan or TARGETjoin processing. You can use the EXPLAIN command to see whether TARGETjoin processing is an option for a particular query. For examples of EXPLAIN outputs that show TARGETjoin query plans, refer to “Reading EXPLAIN Output on a TARGETjoin Query” on page 9-47.

Turning Off TARGETjoin Query Processing

If you do not want query plans generated that include TARGETjoin processing, make sure either that no indexes exist on the foreign key columns of the fact table(s) or that the segments containing the indexes are in the OFFLINE state. You can drop the indexes with the DROP INDEX command or you can bring the indexes to the OFFLINE state with the ALTER SEGMENT...OFFLINE command. For information on these commands, refer to the SQL Reference Guide.

When to Use TARGETjoin Processing

When deciding whether to use TARGETjoin processing, consider the following questions:

- Do your queries perform well already?
- Is your schema appropriate for TARGETjoin processing?
- What are the trade-offs between creating more STAR indexes and creating indexes on the foreign key columns?

Evaluate Query Performance

If your query performance is already good, do not create the indexes to enable TARGETjoin processing. The sole purpose is to speed up the performance of queries, and if that performance is already good, then there is no reason to incur the administrative cost of creating and maintaining more indexes.

Often, however, the question, “Is performance good?” is not so easily answered. Do all queries need to complete in less than 10 seconds, or can some take 10 minutes? Is it acceptable for some queries that are not issued very often to take several hours?
The only people who can answer these questions are you and your user community. The users will always appreciate faster queries, but is absolute performance more important than, for example, longer downtime during database maintenance? Evaluate your query performance and decide whether the costs of creating the indexes to enable TARGETjoin processing are worth the benefits.

For more information about the costs associated with TARGETjoin processing, refer to “Administration Considerations for TARGETjoin Processing” on page 4-12 of Chapter 4, “Planning a Database Implementation.”

**Schema Types**

TARGETjoin processing joins tables that have primary key/foreign key relationships. These relationships occur in a wide variety of schemas, most notably in star schemas. But many star schemas expand on the simple star schema and have outboard tables. Such schemas are sometimes referred to as “snowflake” schemas. The following figure shows such a schema:

With a snowflake schema, TARGETjoin processing is possible to join the dimension tables to the fact table and to join the outboard tables to the dimension tables that reference them. To enable TARGETjoin processing in the preceding figure, you must create TARGETindexes on the foreign key columns of the Fact table and on the foreign key columns of the Dimension 2 table.
Trade-Offs Between Many STAR Indexes and TARGETjoin Processing

Creating foreign key indexes to enable TARGETjoin processing is an alternative to creating many STAR indexes. For schemas with many dimensions, there are fewer TARGET indexes to create than there are potential STAR indexes. For example, a schema with one fact table and ten dimension tables has ten factorial (10!), or 3,628,800 possible STAR indexes. The same schema has ten possible foreign key indexes.

When the number of dimensions in a schema is relatively small, the number of potential STAR indexes is much lower, and it is much easier to create a set of STAR indexes that perform well under a large variety of queries. For example, on a schema with one fact table and four dimension tables, two or three STAR indexes will provide excellent performance for virtually any query against the database. As the number of dimensions gets larger, however, it becomes more difficult to cover a large variety of queries with a relatively small number of STAR indexes.

If you know exactly what queries will be run against the database, then you can always create STAR indexes that are well-suited for those queries. As the number of STAR indexes gets large, however, this becomes more and more impractical. Exactly what “large” is depends on what is practical in your unique situation. A good compromise between ultimate performance and manageability is to create a few STAR indexes and to also create foreign key indexes to enable TARGETjoin processing. This will provide excellent performance on some queries and good performance on all queries.

Examples

The examples in this section are based on a modified version of the Aroma database, the sample database shipped with Red Brick Warehouse. To create this database, create indexes on the foreign keys of the Sales table of the Aroma database using the following CREATE INDEX statements:

```sql
CREATE INDEX sales_perkey_target_idx ON sales (perkey);
CREATE INDEX sales_promokey_target_idx ON sales (promokey);
CREATE INDEX sales_storekey_small_target_idx ON sales (storekey) INDEX domain small;
CREATE INDEX sales_classkey_prodkey_btree_idx ON sales (classkey, prodkey);
```

Note: The TARGET index on the foreign key that references the Store table is created as DOMAIN SMALL because there are only 18 unique values. The index on the foreign key that references the Product table is a B-TREE index because it is a multi-column foreign key.
Query That Chooses TARGETjoin

Suppose you want to answer the following business question:

What are the dollar values of the sales on each promotion of type 900 in Atlanta, and what are the total sales for each promotion and for all the promotions of type 900 in Atlanta?

The following query against the modified Aroma database answers this question and is executed using TARGETjoin processing:

```sql
RISQL> set stats info;
RISQL> select substr(promo_desc, 1, 20) as PROMO_DESC,
>           substr(store_name, 1, 25) as STORE_NAME, dollars
> from sales natural join store natural join promotion
> where city like 'Atlanta%' and promo_type = 900
> order by 1
> break by 1 summing 3
> ;
** STATISTICS ** (500) Compilation = 00:00:00.17 cp time,
00:00:00.20 time, Logical IO count=133
PROMO_DESC   STORE_NAME DOLLARS
Christmas special Olympic Coffee Company 210.00
Christmas special NULL 210.00
Easter special Olympic Coffee Company 420.00
Easter special Olympic Coffee Company 30.00
Easter special Olympic Coffee Company 150.00
Easter special NULL 600.00
NULL       NULL     810.00
** STATISTICS ** (1457) EXCHANGE (ID: 19) Parallelism over 1 times
High: 1 Low: 1 Average: 1.
** STATISTICS ** (1457) EXCHANGE (ID: 25) Parallelism over 1 times
High: 1 Low: 1 Average: 1.
** STATISTICS ** (1458) CHOOSE PLAN (ID: 3) Choice: 3 was chosen 1 times.
** STATISTICS ** (1461) CHOOSE PLAN (ID: 3) TARGETjoin was done 1 times.
** STATISTICS ** (500) Time = 00:00:00.20 cp time, 00:00:00.74
  time, Logical IO count=174
** INFORMATION ** (256) 7 rows returned.
RISQL>
```

This query constrains on the Store table (city like 'Atlanta%') and on the Promotion table (promo_type = 900). These two tables are referenced by foreign keys from the Sales table. The STARindex on the Sales table was created with the following CREATE INDEX statement:

```sql
create star index sales_star_idx
   on sales (sales_date_fkc, sales_product_fkc,
               sales_store_fkc, sales_promo_fkc);
```
Notice the order of the keys in the STARindex. The foreign key constraints that reference the Store and Promotion tables are the last two keys of this index. A STARjoin query performs best when the leading key(s) of the index are constrained in the query. Adding a constraint on the Period table to the previous query causes it to use STARjoin processing to join the tables as follows:

```sql
RISQL> set stats info;
RISQL> select substr(promo_desc, 1, 20) as PROMO_DESC,
>                               substr(store_name, 1, 25) as STORE_NAME, dollars
> from sales natural join store natural join promotion natural
join period
> where city like 'Atlanta%'
> and promo_type = 900
> and year = 1995
> order by 1
> break by 1 summing 3
> ;
** STATISTICS ** (500) Compilation = 00:00:00.17 cp time,
00:00:00.31 time, Logical IO count=135
PROMO_DESC STORE_NAME DOLLARS
Easter special Olympic Coffee Company 30.00
Easter special Olympic Coffee Company 150.00
Easter special NULL 180.00
NULL NULL 180.00
** STATISTICS ** (1458) CHOOSE PLAN (ID: 3) Choice: 1 was chosen 1 times.
** STATISTICS ** (1459) CHOOSE PLAN (ID: 3) STARjoin on 1 tables
was done 1 times.
** STATISTICS ** (1460) CHOOSE PLAN (ID: 3) used Index
SALES_STAR_IDX of Table SALES 1 times for STARjoin.
** STATISTICS ** (500) Time = 00:00:00.39 cp time, 00:00:00.39
time, Logical IO count=310
** INFORMATION ** (256) 4 rows returned.
RISQL>
```

This query chooses STARjoin processing instead of TARGETjoin processing because it constrains on the Period table. The `sales_date_fkc` constraint from the Period table is the leading key of the STARindex; therefore, the STARindex offers optimal performance in this case.
Reading EXPLAIN Output on a TARGETjoin Query

When TARGETjoin processing is a possible choice for a query execution path, the Choose Plan operator in the EXPLAIN output shows as many as three choices:

• Table Scan (may include B-TREE 1-1 match, depending on the indexes available)
• STARjoin
• TARGETjoin

The join method is chosen at runtime and can be displayed with the statistics messages issued when SET STATS INFO is enabled.

STAR and TARGET Plan

The following is an example of the EXPLAIN output for a query that has STARjoin processing, table scan, and TARGETjoin processing as execution options. This query is run against the modified Aroma database (with TARGET indexes).

```
RISQL> explain select count(*)
> from sales natural join period natural join store
> where year = 1995
> and store_name like 'C%';
** STATISTICS ** (500) Compilation = 00:00:00.16 cp time,
00:00:00.88 time, Logical IO count=133
EXPLANATION
[  
- EXECUTE (ID: 0) 5 Table locks (table, type): (PERIOD, Read),
(STORE, Read), (SALES, Read), (PRODUCT, Read), (PROMOTION, Read)
--- CHOOSE PLAN (ID: 1) Num prelims: 2; Num choices: 3; Type:
StarJoin;
  Prelim: 1; Choose Plan [id : 1] {
    BIT VECTOR SORT (ID: 2)
    -- TABLE SCAN (ID: 3) Table: PERIOD, Predicate: (PERIOD.YEAR)
    = (1995)
  }
  Prelim: 2; Choose Plan [id : 1] {
    BIT VECTOR SORT (ID: 4)
    -- TABLE SCAN (ID: 5) Table: STORE, Predicate:
    {((STORE.STORE_NAME) <= ('C') ) && ((STORE.STORE_NAME) >= ('C')) } && ((STORE.STORE_NAME) <like> ('C%')
    }
  }
  Choice: 1; Choose Plan [id : 1] {
```
Notice that there are three choices under the Choose Plan operator: STARjoin, Table Scan with B-TREE 1-1 match, and TARGETjoin.
TARGET Only Plan

The following example illustrates the EXPLAIN output for a query that does not have a STARjoin execution option. This is the same query as the previous example, but this query is run against an Aroma database with TARGET indexes on the Sales table foreign keys and no STAR indexes.

Note: Red Brick Systems does not recommend dropping your STAR indexes when using TARGETjoin processing. This example is just to illustrate what the EXPLAIN output looks like in those situations where there are no STAR indexes.

RISQL> explain select count(*)
> from sales natural join period natural join store
> where year = 1995
> and store_name like 'C%';
** STATISTICS ** (500) Compilation = 00:00:00.06 cp time,
00:00:00.10 time, Logical IO count=90
EXPLANATION
[ - EXECUTE (ID: 0) 3 Table locks (table, type): (PERIOD, Read),
(STORE, Read), (SALES, Read)
--- CHOOSE PLAN (ID: 1) Num prelims: 2; Num choices: 2; Type:
StarJoin;

Prelim: 1; Choose Plan [id : 1] {
   BIT VECTOR SORT (ID: 2)
   -- TABLE SCAN (ID: 3) Table: PERIOD, Predicate: (PERIOD.YEAR) = (1995)
}

Prelim: 2; Choose Plan [id : 1] {
   BIT VECTOR SORT (ID: 4)
   -- TABLE SCAN (ID: 5) Table: STORE, Predicate:
   (((STORE.STORE_NAME) <= ('C' 

   )) && ((STORE.STORE_NAME) >= ('C'))
} && ((STORE.STORE_NAME) <like> ('C%'))

Choice: 1; Choose Plan [id : 1] {
   HASH AVL AGGR (ID: 6) Grouping: FALSE, Distinct: FALSE;
   -- FUNCTIONAL JOIN (ID: 7) 1 tables: PERIOD 
   ----- BTREE 1-1 MATCH (ID: 8) Join type: InnerJoin; Index(s):
   [Table: PERIOD
   D, Index: PERIOD_PK_ID]
   ------ FUNCTIONAL JOIN (ID: 9) 1 tables: STORE
   -------- BTREE 1-1 MATCH (ID: 10) Join type: InnerJoin;
   Index(s): [Table: STORE, Index: STORE_PK_ID]
   ---------- TABLE SCAN (ID: 11) Table: SALES, Predicate:<none>
Choice: 2; Choose Plan [id : 1] {  
| HASH AVL AGGR (ID: 12) Grouping: FALSE, Distinct: FALSE; |
| -- EXCHANGE (ID: 13) Exchange type: Functional Join |
| ---- HASH AVL AGGR (ID: 14) Grouping: FALSE, Distinct: FALSE; |
| ------ EXCHANGE (ID: 15) Exchange type: TARGETjoin |
| -------- TARGET JOIN (ID: 16) Table: SALES, Predicate: <none> |
| ; Num indexes: 2 Index(s): Index: SALES_PERKEY_TARGET_IDX ,Index: SALES_STOREKEY_SMALL_TARGET_IDX |
| ---------- FUNCTIONAL JOIN (ID: 17) 1 tables: PERIOD |
| -------------- VIRTAB SCAN (ID: 18) |
| ---------- FUNCTIONAL JOIN (ID: 19) 1 tables: STORE |
| -------------- VIRTAB SCAN (ID: 20) |
| {  
| ** STATISTICS ** (500) Time = 00:00:00.00 cp time, 00:00:00.04 time, Logical IO count=90  
| ** INFORMATION ** (256) 43 rows returned. 
RISQL> |

Notice that this query has two choices of join methods: table scan and TARGETjoin. If you run this query with SET STATS INFO enabled, you can see that it runs using TARGETjoin, which is Choice 2 in the EXPLAIN output:

RISQL> select count(*)  
> from sales natural join period natural join store  
> where year = 1995  
> and store_name like 'C%';  
** STATISTICS ** (500) Compilation = 00:00:00.11 cp time, 00:00:00.23 time, Logical IO count=90  
4561  
** STATISTICS ** (1457) EXCHANGE (ID: 13) Parallelism over 1 times  
High: 1 Low: 1 Average: 1.  
** STATISTICS ** (1457) EXCHANGE (ID: 15) Parallelism over 1 times  
High: 1 Low: 1 Average: 1.  
** STATISTICS ** (1458) CHOOSE PLAN (ID: 1) Choice: 2 was chosen 1 times.  
** STATISTICS ** (1461) CHOOSE PLAN (ID: 1) TARGETjoin was done 1 times.  
** STATISTICS ** (500) Time = 00:00:00.36 cp time, 00:00:00.68 time, Logical IO count=130  
** INFORMATION ** (256) 1 rows returned. 
RISQL> 

If you want to know which TARGET index(es) were used in the query, the names of the indexes are shown in the EXPLAIN output of the query, shown in the previous example.

9-50 Warehouse Administrator’s Guide for UNIX Platforms
Summary and Recommendations

TARGETjoin processing is a complementary technology to STARjoin processing. It works well when you do not have an optimal STARindex available. The following diagram shows a schema that is an ideal candidate for TARGETjoin processing to complement STARjoin processing:

This simple star schema has a single fact table and ten dimension tables referenced by foreign key/primary key relationships.
Such a schema can be found in a large variety of applications. For example, it could be a retail schema where the fact table is a Sales table with dimensions such as Period, Product, Market, Customer, and so on. It could also be a health insurance claims database where the fact table is a Claims table with dimensions such as Member, Provider, Occupation, Physician, and so on.

With this type of schema—one with a large number of dimension tables—you might need to create many STAR indexes in order for STARjoin processing to work well over a large variety of ad-hoc queries.

With TARGETjoin processing, instead of creating many STAR indexes, you can create one or two STAR indexes and then create TARGET indexes on the foreign keys of the fact table.

**Indexes to Create**

For this example, assume you know that about 70% of your queries constrain on dimensions 1—4, and the rest are ad-hoc queries that constrain on different combinations of all ten dimensions. In this case, a good strategy is to create the following indexes:

- One STAR index covering all of the dimensions, with the leading key the one the fact table is segmented on (for example, the time dimension).
- A second STAR index covering dimensions 1—4, which are used in 70% of your queries. You might also want the leading key of this STAR index to be the time dimension, or whatever dimension the fact table is segmented on.
- Ten TARGET indexes, one on each of the fact table foreign keys.

The STAR index that covers all of the dimension table keys has two main purposes:

- It provides good query performance over a large variety of queries.
- It enables STARjoin/TARGETjoin query plans to be generated for any query against this database, as described in “When There Is At Least One STAR Index On the Fact Table” on page 9-41.

This strategy gives optimal performance on the core 70% queries, and it also gives good performance on virtually any ad-hoc query that users might ask.
Large Dimension Table

Suppose further that Dimension 10 is a large table named Customer with 1,000,000 rows. In this situation, TARGETjoin processing might not offer the best performance on queries that loosely constrain on the large Customer table. But these queries should perform well with STARjoin processing, particularly if the Customer dimension is the last key in the STARindex and other keys in that STARindex are constrained as well. Therefore, the Customer table (or whatever your large table is) should be the last key in the STARindex that covers all your dimensions.

Experiment

All schemas and data are different. It is not possible to know exactly what the best index implementation is without some experimentation and evaluation of your query performance. If query performance is already good with one or two STAR indexes, then you probably do not need to add any additional indexes.

This example is a simplified case; obviously your situation will be different. But if you refer to these examples as guidelines and if your schema is a good candidate for TARGETjoin processing, it will nicely complement the STARjoin processing that works very well in those “70%” queries and be a big performance boost to your data warehouse.
Using Synonyms to Control Fact-to-Fact Joins

You can use synonyms to perform a hash join or a B-TREE 1-1 match join on queries that would normally use STARjoin processing. This is particularly useful with fact-to-fact joins in complex schemas when the fact-to-fact STARjoin operation is not performing well.

To use STARjoin processing on a multiple fact table join, each fact table must have at least one foreign key reference to a common dimension. Additionally, there must exist STAR indexes on each fact table where the shared foreign keys are in the same relative order in the STAR indexes.

In the multiple fact table schema in the preceding figure, assume indexes exist with the following definitions:

```sql
create star index STAR_FACT1
    on fact1(CommonDimensionKey, DimAKey) ;
create star index STAR_FACT2
    on fact2(CommonDimensionKey, DimBKey) ;
```

The following query uses the STAR indexes for the fact-to-fact join operation:

```sql
select dimA.column, fact1.column, fact2.column, dimB.column
from dimA, fact1, CommonDimension, fact2, dimB
where DimA.column = <value1>
    and DimB.column = <value2>
    and DimA.DimAKey = fact1.DimAKey
    and fact1.CommonDimensionKey =
        CommonDimension.CommonDimensionKey
    and fact2.CommonDimensionKey =
        CommonDimension.CommonDimensionKey
    and fact2.DimBKey = DimB.DimBKey ;
```
If this fact-to-fact STARjoin query is not performing well, you can use the ALTER TABLE...ALTER CONSTRAINT command to move a foreign key constraint to reference a synonym instead of the base table to which the synonym refers. Any STAR indexes that reference the base table are not considered in the query plan when the synonym (instead of the base table) is specified in the query. Also, any STAR indexes that reference the synonym are not considered when the base table is specified in the query. This allows Red Brick Warehouse to effectively consider a different set of indexes depending on whether the base table or the synonym is included in the query.

The following ALTER TABLE...ALTER CONSTRAINT command changes the constraint on one of the fact tables to reference the synonym instead of the CommonDimension table:

```
alter table fact1 alter constraint col1
    references synonym_of_commondimension;
```

The foreign key constraint from table Fact1 now points towards the synonym instead toward the CommonDimension table, as shown in the following figure:

Now if you specify the synonym instead of the CommonDimension table in your query, a fact-to-fact STARjoin does not occur:

```
select dimA.column, fact1.column, fact2.column, dimB.column
from dimA, fact1, Synonym_of_Commondimension, fact2, dimB
where DimA.column = <value1>
    and DimB.column = <value2>
    and DimA.DimAKey = fact1.DimAKey
    and fact1.CommonDimensionKey =
        Synonym_of_Commondimension.CommonDimensionKey
    and fact2.CommonDimensionKey =
        Synonym_of_Commondimension.CommonDimensionKey
    and fact2.DimBKey = DimB.DimBKey ;
```
This query uses a hash join to join to the results of two single-fact table STARjoin operations instead of the multiple fact table STARjoin. For some queries, this might improve performance. Keep in mind, however, that this is extremely dependent on many factors included schema definition, available indexes, how tightly constrained the query is, uniformity of the data, etc. This does provide an additional tuning tool for multiple fact table joins, but it requires a certain amount of experimentation to see how it works on your database.

Note: The synonym and the base table share the same physical data; they are only different logically. Query results are the same selecting from a synonym or from the table to which the synonym refers.

For the syntax of the ALTER TABLE and CREATE SYNONYM commands, refer to the SQL Reference Guide.
Making SQL-Based Improvements

In some cases, changing the way queries are constructed results in improved performance. Determining when and how to make these changes requires an in-depth knowledge of SQL, but the following examples provide some limited suggestions. You will also find some ideas about alternative methods of accomplishing tasks in the SQL Self-Study Guide and the SQL Reference Guide.

**UNION Versus Interdimensional ORs**

A query that uses the union of two (or more) queries can potentially run faster than a query that constrains the referenced (dimension) tables with an OR operator. The results returned are the same.

**Example**

The following SQL fragments illustrate a case where performance can be improved by splitting a query with OR constraints on referenced tables into two queries whose results are combined with a UNION operation. Tables T2 and T3 are independent referenced tables and reflect different dimensions of the data—that is, the two tables do not reference each other.

<table>
<thead>
<tr>
<th>Interdimensional OR:</th>
<th>UNION:</th>
</tr>
</thead>
<tbody>
<tr>
<td>select ... from t1, t2, t3</td>
<td>select ... from t1, t2</td>
</tr>
<tr>
<td>where t2.col1 = 'x'</td>
<td>where t2.col1 = 'x'</td>
</tr>
<tr>
<td>or</td>
<td>union</td>
</tr>
<tr>
<td>t3.col1 = 'y'</td>
<td>select ... from t1, t3</td>
</tr>
<tr>
<td></td>
<td>where t3.col1 = 'y'</td>
</tr>
</tbody>
</table>

**Subquery in the FROM Clause Versus Correlated Subquery**

A query that uses a subquery in the FROM clause often runs faster than an equivalent query (a query that answers the same question and returns the same results) with a correlated subquery in the SELECT list. Therefore, re-writing a correlated subquery as a subquery in the FROM clause might provide a substantial performance improvement.
Example

The following query, from the Aroma database, contains a correlated subquery in the select list:

```sql
RISQL> select outer_product.prod_name as aroma_product,
       sum(outer_sales.dollars) as dollars_jan_94,
       (select sum(inner_sales.dollars)
        from sales as inner_sales, product as inner_product,
        period as inner_period
        where inner_sales.prodkey = inner_product.prodkey
        and inner_sales.classkey = inner_product.classkey
        and inner_sales.perkey = inner_period.perkey
        and inner_period.year = outer_period.year + 1
        and inner_period.month = outer_period.month
        and inner_product.prod_name = outer_product.prod_name
        and inner_product.pkg_type like 'No pkg%'
        ) as dollars_jan_95
       from sales as outer_sales, product as outer_product,
       period as outer_period
       where outer_sales.prodkey = outer_product.prodkey
       and outer_sales.classkey = outer_product.classkey
       and outer_sales.perkey = outer_period.perkey
       and outer_period.year = 1994
       and outer_period.month = 'JAN'
       and outer_product.pkg_type like 'No pkg%'
       group by outer_product.prod_name, outer_period.year,
       outer_period.month
       order by outer_product.prod_name;
```

**STATISTICS** *(500) Time = 00:00:39.38 cp time, 00:00:40.04 time,
Logical IO count=27651, IO count=6
**INFORMATION** *(256)* 25 rows returned.

<table>
<thead>
<tr>
<th>AROMA_PRODUCT</th>
<th>DOLLARS_JAN_94</th>
<th>DOLLARS_JAN_95</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aroma Roma</td>
<td>10460.75</td>
<td>15055.00</td>
</tr>
<tr>
<td>Aroma baseball cap</td>
<td>1113.00</td>
<td>1049.40</td>
</tr>
</tbody>
</table>
...
The following query, from the Aroma database, uses a subquery in the FROM clause:

```
RISQL> select aroma_product, dollars_jan_94, dollars_jan_95
from
  (select product.prod_name, sum(dollars)
   from sales, product, period
   where sales.prodkey = product.prodkey
     and sales.classkey = product.classkey
     and sales.perkey = period.perkey
     and period.year = 1994
     and period.month = 'JAN'
     and product.pkg_type like 'No pkg'
   group by product.prod_name, period.year,
           period.month )
  as sales_alias1 (aroma_product, dollars_jan_94)
natural join
  (select product.prod_name, sum(dollars)
   from sales, product, period
   where sales.prodkey = product.prodkey
     and sales.classkey = product.classkey
     and sales.perkey = period.perkey
     and period.year = 1995
     and period.month = 'JAN'
     and product.pkg_type like 'No pkg'
   group by product.prod_name, period.year,
           period.month )
  as sales_alias2 (aroma_product, dollars_jan_95)
order by aroma_product;
```

<table>
<thead>
<tr>
<th>AROMA_PRODUCT</th>
<th>DOLLARS_JAN_94</th>
<th>DOLLARS_JAN_95</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aroma Roma</td>
<td>10460.75</td>
<td>15055.00</td>
</tr>
<tr>
<td>Aroma baseball cap</td>
<td>1113.00</td>
<td>1049.40</td>
</tr>
</tbody>
</table>

**STATISTICS** *(500) Time = 00:00:01.20 cp time, 00:00:02.11 time,
Logical IO count=169, IO count=260
** INFORMATION ** *(256) 25 rows returned.*

This query completes in 2.11 seconds, while the previous query (with the correlated subquery) completes in 40.04 seconds. Both queries return the same results. The actual difference in the response time of a correlated subquery versus a subquery in the FROM clause will differ depending on the query, the data, and other system-dependent factors.

**Note:** Actual query response times will vary from system to system.
Query performance can be improved by using multiple processes to process queries against large tables. Because multiple processes consume more system resources—CPU, disk, processes, and memory—than a single process, several parameters in the `rbw.config` file allow you to control the amount of parallel processing, balancing query response time against the available system resources and demands of other users. Queries that involve a relation scan of a large table or a STAR index are candidates for parallel processing.

This chapter describes query performance improvements that can be achieved by parallel query processing and is organized as follows:

- Parallel Query Tuning Parameters
- Enabling Parallel Query Processing
- Limiting I/O Contention with the FILEGROUP Parameter
- Increasing Disk Group Processes with the GROUP Parameter
- Limiting Available Processes
- Setting Minimum Row Requirements with ROWS_PER_TASK Parameters
- Forcing the Number of Parallel Tasks with the FORCE_TASKS Parameters
- Enabling Partitioned Parallelism for Aggregation
- Other Limitations to Parallelism
- Analysis of System Resources and Workload
- Tuning for Specific Query Types
- Summary: Basic Guidelines
## Parallel Query Tuning Parameters

The extent to which parallel query processing is used is based on the tuning parameters listed in the following table. After these parameters are set, parallelism on demand is enabled whenever the specified conditions are met; without any further tuning, the end user will benefit from parallel processing.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TUNE FILE_GROUP</td>
<td>Reduces seek contention on disk devices.</td>
</tr>
<tr>
<td>TUNE GROUP</td>
<td>Sets number of parallel processes per file group (disk group).</td>
</tr>
<tr>
<td>TUNE TOTALQUERYPROCS</td>
<td>Sets maximum number of processes available for parallel query processing at one time by all warehouse server processes.</td>
</tr>
<tr>
<td>TUNE QUERYPROCS*</td>
<td>Sets maximum number of processes available for parallel query processing at one time by a single warehouse server process.</td>
</tr>
<tr>
<td>TUNE ROWS_PER_SCAN_TASK*</td>
<td>Limits number of parallel processes applied to a query that performs a relation scan (that is, does not use an index).</td>
</tr>
<tr>
<td>TUNE ROWS_PER_JOIN_TASK*</td>
<td>Limits number of parallel processes applied to the index-probing portion of a query.</td>
</tr>
<tr>
<td>TUNE ROWS_PER_FETCH_TASK*</td>
<td>Limits number of parallel processes applied to the row-data-fetching phase of a query.</td>
</tr>
<tr>
<td>TUNE FORCE_FETCH_TASKS*</td>
<td>Sets the number of parallel tasks for fetching rows in queries that use a STAR index.</td>
</tr>
<tr>
<td>TUNE FORCE_JOIN_TASKS*</td>
<td>Sets the number of parallel tasks for joining tables in queries that use a STAR index.</td>
</tr>
<tr>
<td>TUNE FORCE_SCAN_TASKS*</td>
<td>Sets the number of parallel tasks for relation scans of tables.</td>
</tr>
<tr>
<td>TUNE FORCE_HASHJOIN_TASKS*</td>
<td>Sets the number of parallel tasks for hybrid hash joins.</td>
</tr>
<tr>
<td>TUNE FORCE_AGREGATION_TASKS**</td>
<td>Sets the number of parallel tasks for partitioned parallel aggregation.</td>
</tr>
<tr>
<td>TUNE PARTITIONED_PARALLEL_AGGREGATION*</td>
<td>Enables or disables parallelism for aggregation operations.</td>
</tr>
</tbody>
</table>

*These parameters can also be set with a SET command entered anywhere SQL statements can be entered.
Enabling Parallel Query Processing

To enable parallel processing, you must set the QUERYPROCS and TOTALQUERYPROCS parameters in the configuration file (rbw.config) to a value greater than zero.

You can control the extent of parallelism for query processing in either of two ways:

- Specify the minimum number of rows you want a task to process with the ROWS_PER_TASK parameters, which prevent incurring the overhead of parallel processing for trivial queries. These parameters are intended to handle the general purpose day-to-day processing.

- Specify the number of parallel tasks you want to use to process a query, regardless of the number of rows per task, with the FORCE_TASKS parameters. These parameters, which allow you more control over CPU resources, are designed for use on specific queries on which you want to specify the number of parallel processes to be used in order to get the query processed quickly, regardless of the resources used. These parameters override the ROWS_PER_TASK parameters.

All parameters can be entered as TUNE parameters in the rbw.config file so they affect all server sessions; the order of these parameters in the rbw.config file is not significant. Alternatively, you can enter them as SQL SET statements, in which case they affect the current session only.

The parallel tuning parameters are discussed in detail in the following sections.
Limiting I/O Contention with the FILE_GROUP Parameter

The FILE_GROUP parameter definitions tell a warehouse server what PSUs are on the same disk in order to reduce seek contention on individual disk devices; these groups of PSUs are referred to as disk groups. This parameter limits the amount of parallelism: In general, at most one process will be allocated per disk group for each operation (such as a scan), unless the TUNE GROUP parameter is used to specify that more than one process can be used for a specific group.

Note: Because of the SuperScan technology used for disk I/O, processes from multiple servers performing relation scans on tables can access the same data with a single read operation, which can reduce seek contention across server processes.

These entries are read at server startup so changes will be in effect for all sessions started after a change to the rbw.config file.

Syntax

To specify disk groups, enter a line for each disk group in the rbw.config file using the following syntax:

```
TUNE FILE_GROUP disk_group_id pathname_prefix
```

*disk_group_id*

Identifier used to group files into the same disk group; this must be an integer in the range of 1 to 32767.

*pathname_prefix*

Any string of characters delimited by whitespace or end-of-line.

Usage Notes

To determine what group a file belongs to, the filename (as specified in CREATE SEGMENT statements) is converted to an absolute filename (corresponding to either a link or an actual file). Then the longest pathname prefix that is a left substring of the filename is located. The disk_group_id associated with this pathname prefix is the ID of the group to which the file belongs.
If no matching pathname prefix exists for a particular filename, the file is considered to be in its own private group with a limit of one process applied to that group. You can change this behavior by including an entry in the rbw.config file with a pathname prefix of ‘/’ and a unique disk group ID; this entry matches any file not matched by any other entry.

Any number of pathname prefixes can be mapped to the same disk group; each mapping requires a separate TUNE FILE_GROUP statement.

If multiple TUNE FILE_GROUP statements contain identical pathname prefixes, the last such entry is used and all others are discarded. With the exception of duplicate elimination, the order of the entries has no impact.

If you want to allow more than one process per disk group, you can increase this limit with the TUNE GROUP parameter.

**Example**

A table has its data segmented as follows:

<table>
<thead>
<tr>
<th>seg1</th>
<th>seg2</th>
<th>seg3</th>
</tr>
</thead>
<tbody>
<tr>
<td>/disk1/psu11</td>
<td>/disk3/psu21</td>
<td>/disk4/psu31</td>
</tr>
<tr>
<td>/disk2/psu12</td>
<td>/disk4/psu22</td>
<td>/disk5/psu32</td>
</tr>
</tbody>
</table>

To reduce seek contention by placing PSUs on the same physical disk in the same file group, enter the following lines in the rbw.config file:

```
TUNE FILE_GROUP 1 /disk1
TUNE FILE_GROUP 2 /disk2
TUNE FILE_GROUP 3 /disk3
TUNE FILE_GROUP 4 /disk4
TUNE FILE_GROUP 5 /disk5
```

Note that FILE_GROUP 4 contains both psu22 and psu31, which are on the same disk (/disk4). Putting these PSUs in the same disk group prevents a query process from assigning two processes to work simultaneously on the two PSUs.
Increasing Disk Group Processes with the GROUP Parameter

The GROUP parameter provides a way to allow some parallel processing to occur within a disk group. In those cases where PSUs are striped across multiple disks, such as disk arrays or multiple disks grouped together as logical volumes, you can use this parameter to allow parallel processing for a specific disk group.

Syntax

To specify parallelism for a disk group, enter a line for each disk group in the rbw.config file using the following syntax:

```
TUNE GROUP disk_group_id num_processes
```

`disk_group_id`
Identifier used to define a disk group with the FILE GROUP parameter; this must be an integer value.

`num_processes`
Maximum number of outstanding processes against the specified disk group. If you do not specify a GROUP parameter for a disk group, the default is one process at a time for that disk group.

Usage Notes

Even though the FILE_GROUP parameter is designed specifically to limit I/O contention within a specific disk group, multiple disks that are striped to appear as one logical volume to the operating system can support more I/O activity. You can increase parallel I/O activity to disk groups of this type (RAID disks or striped logical volumes) with the GROUP parameter.

In cases where queries are CPU-intensive rather than I/O-bound and there is excess CPU capacity, you can use the GROUP parameter to allow additional parallelism to take advantage of all the CPU capacity.

You need to enter a separate GROUP parameter for each disk group you want to modify.
Example

Suppose in the example on page 10-5 that /disk1 and /disk2 are actually logical volumes that are striped across five physical disks each. You could adjust the parallelism per disk group to accommodate CPU-intensive queries that are heavily concentrated in seg1 by entering the following lines in the rbw.config file:

```
TUNE GROUP 1 5
TUNE GROUP 2 5
```

Any query that accessed seg1 now could have as many as five I/O requests outstanding against /disk1 and /disk2.

Limiting Available Processes

You can control the number of processes available for parallel query processing and the allocation of those processes in a multi-user environment. The TOTALQUERYPROCS parameter specifies the maximum number of processes available for parallel queries at one time on all the servers controlled by a single daemon, providing a mechanism to control the system load imposed by parallel queries. The QUERYPROCS parameter specifies the maximum number of concurrent parallel processes to be used in processing a single query, providing a mechanism to control the resources allocated to a single server (user).

The algorithm that allocates processes to queries employs a “graceful decrease” mechanism to ration remaining processes when the demand is high. After 50% of the total processes available for processing queries have been allocated, subsequent queries are allocated fewer processes per query.

Example

Assume TOTALQUERYPROCS is 1,000, QUERYPROCS is 100, and five queries have each been allocated 100 processes so that 500 processes out of the 1,000 total processes have been allocated. In this case, a “graceful decrease” sets in and subsequent queries each receive fewer than 100 processes apiece (even if they request 100). Once the ratio of allocated processes to total processes drops below 50%, queries again receive the requested number of processes, up to the limit imposed by the QUERYPROCS value.
Tuning a Warehouse for Parallel Query Processing

Limiting Available Processes

**TOTALQUERYPROCS**

To specify a limit on the total number of processes available for parallel queries to all servers controlled by a single warehouse daemon, enter a line in the `rbw.config` file using the following syntax:

```bash
TUNE TOTALQUERYPROCS max_parallel_tasks
```

*max_parallel_tasks*

A non-negative integer in the range of 0 to 32767. (A value of 0 or 1 effectively disables parallel query execution.)

The count specified by `max_parallel_tasks` does not include those processes allocated to the base servers (the number limited by the MAX_SERVERS parameter). Changing this parameter might also require operating-system parameters to be changed, as described in the *Installation and Configuration Guide*. Individual operating systems have limits or operating-system parameters that limit the allowable range of TOTALQUERYPROCS.

**QUERYPROCS**

To specify a limit on the total number of parallel processes available for a single query for all sessions, enter a line in the `rbw.config` file using the following syntax:

```bash
TUNE QUERYPROCS num_per_query
```

To specify a limit on the total number of parallel processes available for a single query for specific sessions, enter a SET command using the following syntax:

```bash
SET QUERYPROCS num_per_query
```

*num_per_query*

A non-negative integer in the range of 0 to 32767. Specifying a value of 0 effectively disables parallel query processing. This number is an upper bound; other factors such as the ROWS_PER_TASK and FORCE_TASKS parameters and the TOTALQUERYPROCS parameter can also limit the number of processes available for a single query. Also note that the distribution of data might result in some processes completing before others, which might reduce the amount of parallelism to less than expected.
Using TOTALQUERYPROCS and QUERYPROCS

If either the TOTALQUERYPROCS parameter or the QUERYPROCS parameter is 0 or not present, no parallel processing is done.

If multiple TOTALQUERYPROCS or QUERYPROCS statements are present, the last such entry is used and all others are discarded. With the exception of duplicate elimination, the order of the entries has no impact.

The TOTALQUERYPROCS entry in the rbw.config file is read at daemon startup so changes are not effective until the rbwapid daemon is restarted.

The QUERYPROCS entry in the rbw.config file is read at server startup so changes are not effective until a new server is started. If the SET command attempts to set QUERYPROCS to a value greater than the value in the rbw.config file, the value in the rbw.config file is used.
Setting Minimum Row Requirements with ROWS_PER_TASK Parameters

You can provide some guidelines to be used by the server in determining how much parallelism to use. If the number of rows to be processed is small, the overhead of parallel processes outweighs the benefits. To prevent this, you can provide a row-per-process limit, which in effect says, “Do not start a parallel process unless the rows-per-process is greater than $x$.” The following three parameters are used for different types of queries and at different points in query processing:

- **ROWS_PER_SCAN_TASK** is the minimum estimated number of rows to be scanned by a relation scan before a parallel relation scan is performed. If the expected number of rows to be scanned exceeds this number, parallel scan processes are used. This number does not affect queries that use an index, only those that do a relation scan of the table.

- **ROWS_PER_FETCH_TASK** is the minimum estimated number of data rows returned during the fetch portion of a STARjoin before parallel fetch processes are used.

- **ROWS_PER_JOIN_TASK** is the minimum estimated number of index entries returned during the join processing (index-probing) portion of a STARjoin before parallel join processes are used.

These parameters can be set for all sessions with entries in the `rbw.config` file, which are read at server startup; therefore, changes are effective only for new server sessions started after the change is made. If multiple values for a given parameter are specified, the last entry of each type is used and all others are discarded. With the exception of duplicate elimination, the order of the entries has no impact. These parameters can also be set for a specific session with a SET command.

The following sections describe how to select values for these parameters. In general, the default values supplied with Red Brick Warehouse inhibit parallelism. If you are not satisfied with the resource consumption and/or query response time and feel that more or less parallelism would improve performance, adjust the values accordingly.
**ROWS_PER_SCAN_TASK**

The ROWS_PER_SCAN_TASK parameter sets a lower limit to the number of rows each scan process must return in order to justify its existence, thus limiting the number of parallel processes initiated for a relation scan. This limit affects queries that use no index but scan an entire table.

The server uses this parameter value as follows to determine the maximum number of processes to use for a query of this type:

1. Each disk group with at least the number of rows specified by ROWS_PER_SCAN_TASK is assigned processes based on the following formula:

   \[
   \text{MIN} \left( \frac{\text{rows\_in\_group}}{\text{rows\_per\_process}}, \text{max\_processes\_per\_group} \right)
   \]

   - Specified by ROWS_PER_SCAN_TASK
   - Specified by GROUP or 1 if no corresponding GROUP entry

2. The number of rows in each disk group containing fewer than the specified number of rows are added together. This total row number is then divided by the number of rows specified by ROWS_PER_SCAN_TASK to determine how many additional processes to allocate:

   \[
   \text{Number of additional processes} = \frac{\text{total\_rows}}{\text{rows\_per\_process}}
   \]

   - Specified by ROWS_PER_SCAN_TASK

**Note:** The number of rows (in a disk group) refers to the number of rows for which space has been allocated in a PSU, and this number might exceed the number of rows visible to a query; for example, space might be allocated for rows that have since been deleted.

**Syntax**

To specify the value used for rows_per_process in the preceding equation for all sessions, enter a line in the rbw.config file using the following syntax:

```
TUNE ROWS_PER_SCAN_TASK rows_per_process
```
To specify the value used for `rows_per_process` in the preceding equation for specific sessions, enter a SET command using the following syntax:

```
SET ROWS_PER_SCAN_TASK rows_per_process  ;
```

**rows_per_process**

An integer in the range of 1 to $2^{31}$. A higher value provides less parallelism in returning rows from the queried table, and a lower value, more parallelism. (Red Brick Systems recommends this number be at least 5,000.)

**Example**

Assume a table has space for 18,000,000 rows allocated across three segments, and each segment contains two PSUs. Each PSU is in its own disk group. The first PSU in each segment has been allocated up to its maximum size, which is sufficient to hold 4,500,000 rows, and the second PSU in each segment has sufficient space allocated to hold 1,500,000 rows. The following figure illustrates this table:

```
<table>
<thead>
<tr>
<th>Segments</th>
<th>PSUs, each in its own disk group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment1</td>
<td><img src="4,500,000" alt="4,500,000" /> <img src="1,500,000" alt="1,500,000" /></td>
</tr>
<tr>
<td>Segment2</td>
<td><img src="4,500,000" alt="4,500,000" /> <img src="1,500,000" alt="1,500,000" /></td>
</tr>
<tr>
<td>Segment3</td>
<td><img src="4,500,000" alt="4,500,000" /> <img src="1,500,000" alt="1,500,000" /></td>
</tr>
</tbody>
</table>
```

Assuming a maximum of one process per disk group and a `ROWS_PER_SCAN_TASK` value of 2,000,000, the maximum number of processes that could be allocated to a relation scan of the table is computed as follows:

1. There are three disk groups, one for each of the large PSUs in each segment, that are each large enough to hold over 2,000,000 rows. These disk groups contribute three processes to the maximum process count, even though each PSU is more than twice as large as the value for the `ROWS_PER_SCAN_TASK` parameter.
Tuning a Warehouse for Parallel Query Processing

Setting Minimum Row Requirements with ROWS_PER_TASK Parameters

2. There are three disk groups, one for each of the small PSUs in each segment, whose allocated space is insufficient to hold 2,000,000 rows. Thus the total number of rows that could be held in the allocated space in all these groups is added up to yield 4,500,000 rows. This number is divided by 2,000,000 and rounded down to yield two processes.

3. The resulting maximum process count is $3 + 2 = 5$. So a relation scan of this table would be processed by at most 5 parallel processes.

Note: The number of processes actually used is also bounded by the values set for the TOTALQUERYPROCS and QUERYPROCS parameters.

ROWS_PER_FETCH_TASK and ROWS_PER_JOIN_TASK

The ROWS_PER_FETCH_TASK and ROWS_PER_JOIN_TASK parameters determine how many parallel processes are used to process queries that use a STAR index. Because queries vary in the amount of work done during the index-probing phase and the row-data-processing phase, you can set different limits for each phase. For example, if your queries tend to require a lot of processing after each row is fetched (GROUP BY, SUM, MIN, and so on), you should assign fewer rows per process for the fetch phase than for the join phase so that more processes are used for the fetch phase.

These parameters allow you to control parallel processing for queries that use a STAR index based on the following guidelines:

- The more tightly constrained a query is on the columns that participate in the STAR index, the smaller the number of parallel processes needed to probe the index efficiently during the join phase. The ROWS_PER_JOIN_TASK parameter defines what “tight” is and how many processes to use during the join phase.
- The more rows to be returned and/or the more processing of row data to be done, the larger the number of parallel processes that can be used effectively during the fetch phase. The ROWS_PER_FETCH_TASK parameter determines how many processes to use during the fetch phase.
Both parameters use the following equations:

\[
\text{Estimated rows} = \frac{(\text{count(*) from fact_table}) \times \prod(\text{number of rows from dimension table})}{\prod(\text{count(*) from dimension tables})}
\]

\[
\text{Number of processes} = \frac{\text{estimated rows}}{\text{rows per process}}
\]

\[\text{Specified by ROWS_PER_FETCH_TASK and ROWS_PER_JOIN_TASK}\]

**Estimated rows**

This equation is applied after the constraints on the dimension tables are processed and the number of rows in each dimension table that satisfy the constraints is known. These numbers are multiplied together and then multiplied by the ratio of rows in the referencing table to all possible values in the STAR index—a type of density or sparseness, to determine an estimated number of rows to return from the STAR index.

**Number of processes**

This equation uses the estimated number of rows and the rows-per-process parameter to calculate how many processes to use during each phase of query processing.

**Syntax**

To specify the minimum number of rows per process used in the preceding equation for each phase for all sessions, enter lines in the `rbw.config` file using the following syntax:

```
TUNE ROWS_PER_JOIN_TASK rows_per_process
TUNE ROWS_PER_FETCH_TASK rows_per_process
```

To specify the minimum number of rows per process used in the preceding equation for each phase for specific sessions, enter a SET command using the following syntax:

```
SET ROWS_PER_JOIN_TASK rows_per_process
SET ROWS_PER_FETCH_TASK rows_per_process
```
**Tuning a Warehouse for Parallel Query Processing**

Setting Minimum Row Requirements with ROWS_PER_TASK Parameters

**rows_per_process**
Integers in the range of 1 to \(2^{31}\). A higher value provides less parallelism, and a lower value, more parallelism. In no case will the system run a query in parallel if the number of processes given by the preceding equations is less than 2.

**Tip:** As a general rule, the values of ROWS_PER_JOIN_TASK and ROWS_PER_FETCH_TASK should each be at least 5,000 to justify the use of parallel processes.

**Example**
This example illustrates how the equations on page 10-14 are used. A referencing (fact) table Fact has 3,000,000 rows. Three referenced (dimension) tables are used in its STAR index: Product with 2,000 rows, Market with 50 rows, and Period with 156 rows. The ROWS_PER_JOIN_TASK parameter is specified to be 90,000 rows per process, and the ROWS_PER_FETCH_TASK parameter is specified to be 50,000.

After the constraints are processed, all the rows from the Product table, 10 rows from the Market table, and 52 rows from the Period table satisfy the constraints:

\[
\text{Estimated rows} = \frac{(3,000,000) \times (2,000 \times 10 \times 52)}{(2,000 \times 50 \times 156)} = 200,000
\]

Based on ROWS_PER_JOIN_TASK, which is specified as 90,000 rows per process, apply the equation for the number of processes that will be used to process the STAR index as follows:

Number of join processes = \(\text{floor}(200,000 / 90,000) = 2\) processes

Based on ROWS_PER_FETCH_TASK, which is specified as 50,000 rows per process, the number of processes used to fetch the data and perform any result pre-aggregation is calculated as follows:

Number of fetch processes = \(\text{floor}(200,000 / 50,000) = 4\)

These figures indicate that \(2 + 4 = 6\) processes could be used to process the query based on the values specified for ROWS_PER_JOIN_TASK and ROWS_PER_FETCH_TASK.
Forcing the Number of Parallel Tasks with the FORCE_TASKS Parameters

The FORCE_TASKS parameters allow you to explicitly specify the number of parallel tasks that are used to process a query. The ROWS_PER_TASK parameters require that you determine a minimum number of rows needed to justify starting a parallel process, an implicit limit. These parameters, except for the FORCE_HASHJOIN_TASKS parameter, are analogous to the ROWS_PER_TASK parameters in their target queries:

- **FORCE_SCAN_TASKS**, which specifies a maximum number of tasks that can be used for relation-scan operations. This parameter does not affect queries that use an index, only those that perform a relation scan.
- **FORCE_FETCH_TASKS**, which specifies the maximum number of parallel tasks that can be used for the fetch portion of a query using a STAR index.
- **FORCE_JOIN_TASKS**, which specifies the maximum number of parallel tasks that can be used for the join portion of a query using a STAR index.
- **FORCE_HASHJOIN_TASKS**, which specifies the maximum number of parallel tasks that can be used for each hybrid hash join in a query.

These parameters are designed to be used only in cases where you want to override the general allocation of parallel processes. For example, if you are running a query to build an aggregate table, no one else is using the system, and you want the query to complete as quickly as possible even if it greatly increases resource consumption.

**FORCE_TASKS Syntax**

To specify the number of tasks for all sessions, enter a line in the rbw.config file using the following syntax:

```
TUNE FORCE_SCAN_TASKS OFF value
TUNE FORCE_FETCH_TASKS OFF value
TUNE FORCE_JOIN_TASKS OFF value
TUNE FORCE_HASHJOIN_TASKS OFF value
```
To specify the number of tasks for a single session, enter a SET statement using the following syntax:

```plaintext
>>> SET --- FORCE_SCAN_TASKS --- OFF
    value

>>> SET --- FORCE_FETCH_TASKS --- OFF
    value

>>> SET --- FORCE_JOIN_TASKS --- OFF
    value

>>> SET --- FORCE_HASHJOIN_TASKS --- OFF
    value
```

**OFF**

Explicit control of parallelism by the specified task limit is not enabled. The default value is OFF.

**value**

An integer value that explicitly limits the maximum number of tasks; however, this value does not guarantee the specified number. The actual number of tasks used will be the lowest of these three values:

- The specified FORCE_TASKS value.
- The number of PSUs over which the table is distributed.
- The number of processes that can be allocated from the QUERYPROCS/TOTALQUERYPROCS pool.

**no argument**

If OFF or value is not specified, the SET command returns the current setting for that parameter. For example:

```plaintext
set force_scan_tasks;
** INFORMATION ** (1433) FORCE_SCAN_TASKS is currently set to 6.
```


**FORCE_SCAN_TASKS**

The value set for FORCE_SCAN_TASKS controls the number of parallel tasks for relation scans of tables.

However, the FORCE_SCAN_TASKS value does not guarantee that a certain number of parallel processes will be used. The actual number of processes used will be the *lowest* of these three values:

- The FORCE_SCAN_TASKS value.
- The number of PSUs over which the table is distributed.
- The number of processes that can be allocated from the QUERYPROCS/TOTALQUERYPROCS pool.

**Example**

Assume the following settings:

<table>
<thead>
<tr>
<th>FORCE_SCAN_TASKS</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSUs in table</td>
<td>18</td>
</tr>
<tr>
<td>QUERYPROCS</td>
<td>18</td>
</tr>
<tr>
<td>TOTALQUERYPROCS</td>
<td>24</td>
</tr>
</tbody>
</table>

Whether the FORCE_SCAN_TASKS value is used in this case depends on the number of processes *available* from the TOTALQUERYPROCS pool. If only 6 processes are already allocated, 18 processes will be available so the FORCE_SCAN_TASKS value of 16 will be used.

**Note:** After 50% of the TOTALQUERYPROCS pool has been allocated, subsequent queries are allocated fewer processes per query.

**Usage Notes**

Also note the following points regarding task allocation for relation scans:

- If FORCE_SCAN_TASKS is set, the ROWS_PER_SCAN_TASK value is ignored.
- If FORCE_SCAN_TASKS is set to a value that is greater than the number of disk groups, some disk groups will simply be allocated more than one process. When FORCE_SCAN_TASKS is set, the number of disk groups does not influence the behavior of parallel processing.
**FORCE_FETCH_TASKS and FORCE_JOIN_TASKS**

The values set for FORCE_FETCH_TASKS and FORCE_JOIN_TASKS control the number of parallel tasks for fetching rows and joining tables in queries that use a STAR index. If either of these values is greater than or equal to 1, it will override the corresponding value set for ROWS_PER_FETCH_TASK or ROWS_PER_JOIN_TASK.

However, the FORCE_FETCH_TASKS and FORCE_JOIN_TASKS values do not guarantee that a certain number of parallel processes will be used. The actual number of processes used to fetch rows will be the lowest of these three values:

- The FORCE_FETCH_TASKS value.
- The number of PSUs over which the table is distributed.
- The number of processes available from the QUERYPROCS/TOTALQUERYPROCS pool.

The actual number of processes used to join tables will usually be the lowest of these two values:

- The FORCE_JOIN_TASKS value.
- The number of processes available from the QUERYPROCS/TOTALQUERYPROCS pool.

**Note:** In rare cases, the FORCE_JOIN_TASKS value might be greater than the number of STAR index rows that match the constraints in the query; therefore, it will not be possible to logically divide and process the query by the specified number of tasks. Instead, the number of matching rows will be used to set the limit on parallel join tasks.

**Examples**

**FORCE_FETCH_TASKS**

Assume the following settings:

- FORCE_FETCH_TASKS: 16
- PSUs in table: 18
- QUERYPROCS: 18
- TOTALQUERYPROCS: 24

In this case, whether the FORCE_FETCH_TASKS value will be used depends on the number of processes available from the TOTALQUERYPROCS pool. If only 6 processes are already allocated, 18 processes will be available and the FORCE_FETCH_TASKS value of 16 will be used.
Tuning a Warehouse for Parallel Query Processing
Forcing the Number of Parallel Tasks with the FORCE_TASKS Parameters

**FORCE_JOIN_TASKS**

Assume the following settings:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORCE_JOIN_TASKS</td>
<td>8</td>
</tr>
<tr>
<td>QUERYPROCS</td>
<td>12</td>
</tr>
<tr>
<td>TOTALQUERYPROCS</td>
<td>30</td>
</tr>
</tbody>
</table>

If there are 9 or more processes available from the TOTALQUERYPROCS pool, the FORCE_JOIN_TASKS value will be used.

**Usage Notes**

Also note the following points regarding task allocation for fetching rows and joining tables:

- You do not have to force both fetch and join tasks. For example, you can force join tasks but allow fetch tasks to be computed dynamically.
- If FORCE_FETCH_TASKS is set, the ROWS_PER_FETCH_TASK value is not used; similarly, if FORCE_JOIN_TASKS is set, the ROWS_PER_JOIN_TASK value is not used.
- Although the number of PSUs over which the table is distributed affects the allocation of parallel fetch tasks, the number of disk groups does not.
- The number of PSUs used to partition the STAR index does not affect the allocation of parallel join tasks.
- For multi-fact table joins, one fact table is selected to control the partitioning. If 10 PSUs are used to distribute the chosen fact table, 10 processes are available for fetch-task partitioning.
- If there are fewer than the requested number of processes available from the QUERYPROCS/TOTALQUERYPROCS pool and both FORCE options are set, the system tries to preserve the ratio of FORCE_JOIN_TASKS to FORCE_FETCH_TASKS values.
FORCE_HASHJOIN_TASKS

The value set for FORCE_HASHJOIN_TASKS controls the number of parallel tasks for hybrid hash joins.

However, the FORCE_HASHJOIN_TASKS value does not guarantee that a certain number of parallel processes will be used. The actual number of processes used will be the lowest of these following values:

• The FORCE_HASHJOIN_TASKS value.
• The number of processes that can be allocated from the QUERYPROCS/TOTALQUERYPROCS pool.

Example

Assume the following settings:

<table>
<thead>
<tr>
<th>FORCE_HASHJOIN_TASKS</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUERYPROCS</td>
<td>12</td>
</tr>
<tr>
<td>TOTALQUERYPROCS</td>
<td>30</td>
</tr>
</tbody>
</table>

If there are 10 or more processes available from the TOTALQUERYPROCS pool, the FORCE_HASHJOIN_TASKS value will be used.

Usage Notes

Also note the following points regarding task allocation for parallel hybrid hash joins:

• The PARALLEL_HASHJOIN option must be set to ON, either with a SET PARALLEL_HASHJOIN ON command or a TUNE PARALLEL_HASHJOIN ON parameter, in order to get any parallelism from hybrid hash joins.
• You must have at least 2 more than the value you specify in FORCE_HASHJOIN_TASKS available from the QUERYPROCS/TOTALQUERYPROCS pool in order to achieve that level of parallelism. For example, in order to get 8 parallel hash join processes, you must specify FORCE_HASHJOIN_TASKS to 8 and have at least 10 tasks available from the QUERYPROCS/TOTALQUERYPROCS pool.
Enabling Partitioned Parallelism for Aggregation

Queries that involve aggregation (SUM, MIN, MAX, COUNT) over groups specified in the GROUP BY clause can benefit from parallelism partitioned on the grouping columns, especially if there are a large number of groups (for example, hundreds of thousands or millions).

Partitioned Parallelism Syntax

To specify the number of aggregation tasks for all sessions and to enable or disable partitioned parallel aggregation, enter a line in the `rbw.config` file using the following syntax:

```plaintext
TUNE -- FORCE_AGGREGATION_TASKS value OFF
```

To specify the number of aggregation tasks for a single session and to enable or disable partitioned parallel aggregation, enter a SET statement using the following syntax:

```plaintext
SET -- FORCE_AGGREGATION_TASKS value OFF
```

OFF

Disables explicit control of the partitioned aggregation parallelism tasks (FORCE_AGGREGATION_TASKS parameter) and disables partitioned aggregation parallelism (PARTITIONED_PARALLEL_AGGREGATION parameter). The default value is OFF.

value

An integer value that explicitly limits the maximum number of additional tasks used for partitioned parallel aggregation; however, this value does not guarantee the specified number. The actual number of tasks used will have an upper bound of the lower of these two values:

- The specified FORCE_AGGREGATION_TASKS value.
- The number of processes that can be allocated from the QUERYPROCS/TOTALQUERYPROCS pool.
no argument
If OFF or value is not specified, the SET command returns the current setting for that parameter. For example:

```sql
set force_aggregation_tasks;
```

** INFORMATION ** (1433) FORCE_AGGREGATION_TASKS is currently set to 6.

** Usage Notes **

When PARTITIONED PARALLEL AGGREGATION is set to OFF, there is still a potential performance gain with parallel aggregation, but where the parallelism is not partitioned by the grouping columns. For relatively small numbers of groups, this should provide better performance than the partitioned parallelism.

When you set partitioned parallel aggregation on, the amount of processes used on your machine potentially doubles—the amount of processes needed for the parallel aggregation plus the amount of processes needed for the rest of the query processing; therefore, to ensure the same resources to the other parts of the parallel query, you should increase the values of the QUERYPROCS and TOTALQUERYPROCS parameters.

Partitioned parallelism is also effective when populating a table with an INSERT INTO...SELECT...GROUP BY operation, and is particularly effective when there are a large number of groups. This might improve the performance of these types of operations when populating aggregate tables for use with the Red Brick Vista option. For details about the Red Brick Vista option, refer to the Red Brick Vista User's Guide.


The equations on page 10-14 determine upper limits on the number of parallel processes applied for specific tasks; however, other system considerations also limit these numbers.

The number of processes applied to index join tasks is also limited by the number of file groups in which the segments for the selected STAR index reside. In the example on page 10-15, in order for two processes to be used to process the join tasks, the index must be spread over at least two file groups. If it is fewer than two file groups, only one process will be applied (unless you force parallelism with the FORCE_JOIN_TASKS parameter or specify more than 1 disk group process with the GROUP parameter). In general, if you examine the segments in which the STAR index resides, the limit on the amount of parallelism used in the join tasks is the number of file groups covered by those segments.

The number of processes assigned to fetch tasks is also limited in the same manner by the number of groups in which the data resides.

Another limitation is the possibility that the system might be unable to allocate the number of processes desired. In the previous example, the equations indicated that five processes should be allocated to process the query. If, however, the system is able to allocate only four processes (other users are using some of the processes allocated for processing parallel queries), the system must allocate the available processes between the join and fetch phases. The allocation of limited processes to the fetch phases is based on the following equation:

\[
\text{Number of fetch processes} = \text{MAX} \left( 1, \left\lfloor \frac{\text{requested for fetch}}{\text{total available}} \times \text{total requested} \right\rfloor \right)
\]

The remaining available processes are then allocated for join processing.

In the example, with only four processes available, the number of processes allocated to fetch processing is 2:

\[
\text{Number of fetch processes} = \text{MAX} \left( 1, \left\lfloor \frac{3}{5} \times 4 \right\rfloor \right) = 2
\]

The remaining two processes are allocated to join processing.

The most important thing to note is that parallelism is bounded by the number of groups in which the index and data reside. In general, the system does not allocate more processes than there are groups affected by the query.
Example

This example illustrates how the number of processes allocated is affected by the distribution of the index and data across file groups. Assume a table is implemented as follows:

```sql
create segment idx1 ... (two PSUs);
create segment idx2 ... (two PSUs);
create segment data1 ... (three PSUs);
create segment data2 ... (three PSUs);
create table fact...
data in (data1, data2) segment by ...
primary index in (idx1, idx2) segment by references of (prodkey)
ranges (min:1000, 1000:max)
```

Assume that each PSU is in a disk group by itself.

The maximum number of processes that will ever be allocated for join processing is four, because the index covers four file groups. If the constraints cover only one segment, the maximum number of join processes actually would be two, because the segment contains two PSUs, each in a separate disk group.

The row data accesses are unpredictable; however, the calculated maximum number of fetch processes that might be allocated to fetch processing is six because the data is distributed across six PSUs, each in a separate file group.
**Analysis of System Resources and Workload**

Parallelism takes advantage of available system resources by scheduling more work concurrently to speed up query processing. Parallelism also introduces additional factors that affect performance gains. Getting the best performance on parallel queries depends on the degree to which you can exploit concurrency and load balancing across a system, a task that requires careful planning, even before you load the database. The main questions concern system resource allocation and usage.

**Disk Usage**

To reduce disk contention and increase I/O activity, I/O load should be distributed evenly across as many physical disks as there are parallel processes. The object is to improve disk service time and improve I/O concurrency in order to reduce the time a query waits because its processes are blocked on I/O.

Ideally, no more than one disk group, as defined by the FILE_GROUP parameter, should be assigned to one physical disk. Disk arrays or disks grouped together using a logical volume manager facility are exceptions; in these cases, there are always multiple physical disks grouped together as one logical disk. In the following discussions, a disk group is treated as interchangeable with a physical disk. If there are multiple files (PSUs) per physical disk, these files should normally be organized into a single disk group using the FILE_GROUP parameter. Such an organization ensures that I/O can be scheduled evenly and reduces excessive head movements on the disk, especially important for parallel scans where it is desirable to have an orderly sequential access on the file and to take advantage of any available read-ahead capability as much as possible.

In some cases where disk arrays or disks are grouped together, such as striped disks or RAID systems, or where the query workload is CPU-intensive rather than I/O-bound and there is excess CPU capacity, better performance might result from allowing more than one process per disk group. If you want to allow parallel processes within a disk group for a query, use the TUNE GROUP parameter to specify the number of processes.
For Data

If you are not sure how uniformly the data will be distributed across segments, or if you want to spread the data across more physical disks because you anticipate the queries will tend to cluster on a single disk or only a few disks, you can use the SEGMENT BY HASH option in creating tables. Hashing segments helps to ensure that data will be evenly distributed across all segments and thereby helps to balance processing across parallel processes. However, before deciding to use the hash option, consider the space management issues associated with hashing; for example, hashed segments cannot be dropped individually or taken offline for loads.

For STAR Indexes

It is not always necessary or desirable to segment the STAR index. Consider whether the benefit merits the extra administration overhead. Parallelism for join tasks is derived based on PSUs and disk groups. As long as there are multiple PSUs and they are not grouped into a disk group, parallelism can be invoked even if the STAR index is in a single segment.

If the STAR index is to be loaded across multiple segments for parallel join processing, load the index on separate disks if possible. As a minimum, allow as many index PSUs or segments as the planned number of join processes. If the CPUs are very fast, you might need to allow more index and join processes to keep the CPUs busy. If not enough disks are available, the next best option is to load the index segments on the same disks as the table segments, as long as the disks are not too busy.

Memory Usage

Memory requirements increase with parallel processing, even if the number of users remains the same. Because parallel processes are spawned from the “parent” server, the child processes inherit many of the characteristics of the parent, including memory requirements. So be aware of the additional memory demands in parallel processing, particularly important in a multi-user environment.

To determine memory requirements at your site, you should first determine how many users are actively executing queries at the same time (concurrently active users). The number of concurrently active users will determine the memory demand and paging/swapping activities in the system at any one time. For example, if 100 users are connected, but 80 of them only submit a query (10 minutes long) once a day, 18 of them only twice a day (5 minutes long), and 2 of them all the time, then you can estimate about 3 concurrently
active users (assuming the 98 users submit their queries evenly throughout the day). Three users probably will not introduce heavy paging/swapping activities in the system. However, if there are 50 concurrently active users, examine carefully whether there is enough memory to support them. If there is not enough memory, thrashing will occur, resulting in very heavy paging/swapping activities.

If the number of concurrently active users is high, consider either adding more memory or reducing the amount of parallelism. You can reduce parallelism either by reducing the number of query processes per user (QUERYPROCS) or by ensuring that only large queries use parallel processing. You can prevent trivial or small queries from using parallel processing by choosing a large value (thousands to tens-of-thousands for STARjoin queries and tens- to hundreds-of-thousands for scans) in the ROWS_PER_JOIN/FETCH/SCAN_TASK parameters.

When the system is up and running, monitor the paging/swapping activities. Acceptable values for these two activities vary depending on system size and speed. Rather than using generic values that may or may not apply to your system, a better approach is to monitor the system when it is not performing optimally; the paging/swapping disks will indicate when things are not right. The two indicators to monitor are the waiting for I/O percentage (WIO) in the System Activity Report (SAR), or an equivalent performance monitoring tool, and the disk service time. You can also evaluate disk usage based on busy percentage, disk request queue, and disk waiting time. If all these indicators are high, users are probably kept waiting while the system is busy paging or thrashing in memory. To decrease the wait time, you can add more paging/swapping devices, reduce parallelism, or add memory.
CPU Allocation

The degree of concurrency in parallel processing largely depends on how many CPUs are available. Although allocating all the CPUs in the system might yield the best results, this option is often not practical because other work in the system could be competing for CPU resources. If all the CPUs are used for parallel query processing, other users would experience a slow-down because CPU resources are less available to them. Therefore, each administrator must decide how the CPU resources are to be distributed. Of course, if all the CPUs are already saturated (that is, 100% busy), there will be no gain and perhaps even a slight degradation from parallel processing. After you have determined the number of CPUs to use for parallel query processing, you can derive the number of parallel processes (QUERYPROCS) as follows:

- For a query that is quite CPU-intensive, set the QUERYPROCS parameter to the “number” of CPUs divided by the number of concurrently active users.
- For a query that is not very CPU-intensive but is I/O-intensive, set the QUERYPROCS parameter to two or three times the value derived for CPU-intensive queries. Monitor the CPU-busy statistic to determine whether more parallel processes are needed.

For example, assume you have a 12-CPU system and you want to allocate about 65% of CPU resources (8 CPUs) to parallel processing. If the system is 45% busy overall, it is reasonable to add more parallel processes until the system reaches at least 65% busy. And if other tasks are executing in the system at the same time, it is reasonable to add even more parallel processes, depending on the distribution of resource consumption. (If there are other tasks, parallel processing probably was not consuming 65% of the CPU resources.)

The FILE_GROUPS setup also affects concurrency. For I/O-intensive queries, specify as many disk groups (physical disks) as there are parallel processes (QUERYPROCS). One factor that the server uses to determine how many parallel processes to create is the number of disk groups: If the number of disk groups is less than the QUERYPROCS value, the number of parallel processes to create will be reduced to the number of disk groups.

If there are more CPUs than disk groups and there is excessive CPU capacity, use the TUNE GROUP parameter to provide more parallelism than is normally allocated per disk group.
Tuning for Specific Query Types

This section describes how you can fine-tune parallel processing for specific query types. Before tuning for parallel processing, however, make sure that the queries themselves are fine-tuned with well-written SQL.

Note: The FORCE_TASKS parameters override the corresponding ROWS_PER_TASK parameters and are not intended for use as general tuning parameters. The following discussion focuses on tuning with the ROWS_PER_TASK parameters.

For information on how Red Brick Warehouse processes queries, refer to “Understanding Red Brick Query Processing” on page 9-25.

Parallel STARjoin Queries

The speedup expected from parallel processing of STARjoin queries (those queries that use a STAR index) depends primarily on:

- Density of the query (number of rows selected from the referencing table).
- Number of parallel processes.
- Mix of parallel (join and fetch) processes.
- Number of disk groups, or file groups, as defined by the TUNE FILE_GROUPS entries in the rbw.config file.
- Amount of parallelism per disk group, as defined by the TUNE GROUP entries in the rbw.config file. Keep in mind that allowing parallelism within disk groups might increase disk contention and degrade performance.

Density

In general, the higher the density of a query, the better the potential for speedup.

Number of Parallel Processes

The number of processes can depend on the distribution of the data. If data for the query is clustered in the referencing table, it is possible that not all parallel processes will get work. In this case, hashing the referencing table segments helps distribute the data more evenly across disks. For some cases where STARjoin queries are slow, alternate STAR indexes might provide more speedup than parallelism; depending on which columns are constrained, consider alternate STAR indexes before resorting to parallel processing.
Mix of Parallel Processes and File Groups

Choosing the right mix between join and fetch processes is also important in speedup gains. For example, assume a query requires relatively much more post processing than join processing but has only one fetch process allocated; the parallel speedup is much less than if more fetch processes were allocated.

Use the following guidelines to finely tune the mix for specific queries:

- For queries dominated by post processing, allocate more fetch than join processes.
- For queries dominated by join processing, allocate more join than fetch processes.
- If there is a mix of join and post processing, start by specifying an equal number of join and fetch processes. Then alter the ratio in both directions by 50% to determine which ratio is best. Usually, ratios of n/1 or 1/n are not the best choices, although there are exceptions.

Use the ROWS_PER_JOIN/FETCH_TASK parameters to control the ratio between join and fetch processes. Note that the values for these two parameters are inversely proportional to the ratio desired. For example, for six join and two fetch processes, specify:

- rows per join process = 2000
- rows per fetch process = 6000

The actual allocation of processes is more complicated:

1. The number of processes is calculated by dividing the rows per join/fetch task into the estimated number of rows. If the join or fetch value is less than the estimated number of rows, no parallel processes are created.

2. The number of processes is evaluated against the number of file groups and group limits; the smaller number becomes the new number of processes. For join processes, the number of file groups for index segments is used. For fetch processes, the number of file groups for the table is used.

3. The QUERYPROCS value is compared with the sum of the join and fetch processes; the smaller number becomes the new number of processes. If the QUERYPROCS value is smaller, the ratio between the join and fetch processes (as specified by the ROWS_PER_JOIN/FETCH_TASK values) is preserved. If, however, the number of file groups for either joins or fetches is less than the number of processes allocated by the ratio, the ratio is not preserved: Maximum parallelism is offered by allocating join and fetch processes up to the QUERYPROCS value.
4. If the remaining number of processes specified by the QUERYPROCS value is less than 50% of TOTALQUERYPROCS, a graceful degradation process begins, assigning only a portion of the remaining QUERYPROCS.

Considerations for Multi-User Environments

In a multi-user environment, it is often not worthwhile to offer parallelism for trivial queries (queries that take less than one minute). Parallel queries consume much more memory and the user-perceivable speedup improvement is relatively small. You must consider the corresponding resource trade-off to decide which query types (trivial, medium, or large) merit parallel processing. To prevent trivial queries from using parallel processing, specify high values for the ROWS_PER_JOIN/FETCH_TASK parameters.

Additionally, use the TOTALQUERYPROCS parameter to limit the number of users of parallel processing; a reasonable value for the TOTALQUERYPROCS parameter is two to three times the QUERYPROCS value. If there is enough memory, you might find values of five times or higher acceptable. Fairly high values for the ROWS_PER_JOIN/FETCH_TASK parameters ensure that only large queries use parallel processing.

Parallel Table Scans

The speedup provided by parallel processing of table-scan operations is determined by the number of parallel scan processes, and the number of file (disk) groups available, and the TUNE GROUP parameter value.

Depending on the speed of the CPUs, several parallel scan processes might be required to keep the “number” of CPUs busy. The recommended procedure is to assign no more than one disk group to one physical disk and to allow parallel scan processing to assign no more than one process per file (disk) group, thus facilitating sequential contiguous disk access whenever no more than one user is accessing the disk at a time. Otherwise, disk service times per block might increase by orders of magnitude. You can, however, use the TUNE GROUP parameter to allow more than one process per group in cases where the a query is CPU-bound on a multi-processor system with additional CPU capacity.

As described for parallel STARjoin queries, the recommendations for limiting trivial queries (those that are small enough to return in one minute or less) and multi-user environments apply to ROWS_PER_SCAN_TASK as well.
SuperScan Technology

SuperScan technology is used whenever multiple users are scanning the same table at the same time or overlap some of the time. The perceived improvement varies greatly from time to time depending on system activities.

SuperScan technology takes advantage of the operating system’s file buffer cache to reduce physical I/O activity. If the scanning processes find most of the I/O blocks in the file buffer cache, there will be a large decrease in I/O activity. If few blocks are found in the file buffer cache, there will be a smaller decrease. Because the file buffer cache is used by everyone in the system, its state (that is, the hit rate) depends on the system activities. The interval between additional users beginning to scan the table also affects the decrease in I/O activity. Because users probably do not have the same time arrival pattern from day to day, improvement from SuperScan technology might vary from day to day.

About Reasonable Values

No one set of tuning values applies to all customers across all queries. As discussed in the previous sections, many considerations affect parallelism and they are different for each customer, depending on the hardware platform, configuration, query mix, and number of users. The information presented here highlights general areas of concern and provides guidelines for tuning query performance for your environment.

The ROWS_PER_JOIN/FETCH/SCAN_TASK parameters have two main uses. First, they control parallelism and determine when it is invoked; use these parameters to set the minimum estimated number of rows required before parallel processing is used. Secondly, the ROWS_PER_JOIN/FETCH_TASK parameters determine the ratio between join and fetch processes for STARjoin queries.

Previous sections discuss factors relevant to the number of parallel processes; the primary one is the “number” of CPUs. Then disk groups, memory, and users are considered, with TOTALQUERYPROCS limiting the number of users for parallel query processing.
Summary: Basic Guidelines

The following suggestions provide some basic guidelines for setting “reasonable” values; refer back to the previous sections for more detailed information. Remember, however, each system and workload is unique, so you must experiment to determine what works best at your site.

1. Set the ROWS_PER_JOIN/FETCH/SCAN_TASK values fairly high. For more information, refer to “Memory Usage” on page 10-27 and “Setting Minimum Row Requirements with ROWS_PER_TASK Parameters” on page 10-10.

2. Set the values for ROWS_PER_JOIN_TASK and ROWS_PER_FETCH_TASK to the same value. For more information, refer to “Tuning for Specific Query Types” on page 10-30.

3. Set QUERYPROCS as discussed in “CPU Allocation” on page 10-29.

4. Set TOTALQUERYPROCS to at least two to three times QUERYPROCS, or even higher if memory is available. For more information, refer to “Limiting Available Processes” on page 10-7.

5. If multiple PSUs for the same table are on the same disk device, define a disk group (file group) for those PSUs to limit disk contention. For more information, refer to “Disk Usage” on page 10-26.

6. In most cases, you do not need to set the TUNE GROUP parameter: generally, only one process per disk group is best. For more information, refer to “Increasing Disk Group Processes with the GROUP Parameter” on page 10-6.

These basic settings will provide users with a reasonable environment for parallelism on large queries, but they will not process every query optimally. If you want to fine-tune parallel processing for more optimal processing of specific queries, refer to the suggestions on page 10-30.
This appendix illustrates how to build a database using Aroma, the sample database included with Red Brick Warehouse, and 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. This example uses the sample database included with Red Brick Warehouse, Aroma. Because Aroma is a relatively small database, default segments are used for the dimension (referenced) tables, and named segments for the referencing (fact) table Sales; however, before you build production databases, you should 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 illustrated in the following figure:

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 were included on the media containing Red Brick Warehouse and should be installed on your system in the directory redbrick_dir/sample_inputs, where redbrick_dir is the 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 directory to a directory named aroma_inputs that you create in your own directory. However, you can also use the files directly from the redbrick_dir/sample_input directory or you can use any editor to write your own files from the examples shown in this appendix.
**Steps for Building a Database**

Building the Aroma database consists of the following steps; the commands and tools used for each step are shown in parentheses.

1. Log in as the `redbrick` user.
2. Make a directory for the new database. (UNIX command `mkdir`)
3. Create the database. (`rb_creator`)
4. Change the default password for the database. (RISQL Entry Tool, GRANT statement)
5. Create the user tables for the database. (RISQL Entry Tool, CREATE TABLE statements)
6. Write the LOAD DATA statements and control files for the Table Management Utility (TMU). (any editor)
7. Load the data into the database. (`rb_tmu`, LOAD DATA control files)
8. Verify that the database was built and loaded successfully.

Each step is described in the following sections.

**Logging in as redbrick**

The `redbrick` user is the administrative account for Red Brick Warehouse. You must be logged in as `redbrick` for the administrative tasks involved in creating a database. The `redbrick_dir/bin` directory must also be in the `redbrick` user’s path.

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 warehouse database administrator.
2. Verify that the `redbrick_dir/bin` directory is in your path by entering:
   ```bash
   $ env
   ```
   In the list of environment variables displayed, check the entry for PATH: Its definition should include `redbrick_dir/bin`. 
Making the Database Directory

Determine where you want to build the database. Then create two directories, one for the database and one for the input files, and copy the Aroma input files. The Aroma scripts, inputs, and loaded database require about 16 megabytes of storage.

**Procedure**

1. As the `redbrick` user, change to the directory you have selected (`my_directory` in this example) by entering:
   
   ```
   $ 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` by entering:
   
   ```
   $ mkdir aroma_db
   ```

3. Verify that permissions on the `aroma_db` directory are set correctly by entering:
   
   ```
   $ ls -dl aroma_db
   
   The permissions, as a minimum, should be:
   
   drwx------ aroma_db
   ```

   which indicate that the `redbrick` user has read, write, and execute permissions. Permissions for group and other will depend upon your environment and are not important for this exercise.

4. Create a directory named `aroma_inputs` for Aroma’s input files by entering:
   
   ```
   $ 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 by entering:
   
   ```
   $ cp redbrick_dir/sample_input/aroma* aroma_inputs
   ```
Creating the Database

Create the new database using the `rb_creator` script; this step creates the system tables. You must be the `redbrick` user to execute `rb_creator`.

Procedure

1. Verify that you are in the correct directory (the directory containing the database directory):

   ```
   $ cd my_directory
   ```

2. Run `rb_creator` by entering:

   ```
   $ rb_creator aroma_db
   ```

3. Enter a logical database name for the new database in the `rbw.config` file, using your favorite text editor. In this example, the new database is named Newaroma.

   ```
   DB NEWAROMA /my_directory/aroma_db
   ```

   Users can access Newaroma by using the `-d` option on the command line for the RISQL Entry Tool, RISQL Reporter, and the TMU. They can omit the `-d` option if the RB_PATH environment variable is set to the logical database name for the desired database.

   **Note:** Newaroma counts as one of the two databases allowed by the Red Brick Warehouse for Workgroups two-database limit.

4. Log out of the `redbrick` account.

The `rb_creator` 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.
Example: Building a Database
Changing the Default Password

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.

Note: 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.

Procedure

1. Verify that the redbrick_dir/bin directory is in your path.
2. Invoke the RISQL Entry Tool by entering:

   $ risql -d NEWAROMA system manager

   providing the logical database name and the default username and password.
3. Change the password for the system account from manager to cryptic by entering:

   RISQL> grant connect to system with cryptic;
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.

The 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 look like this:

```sql
create table market (
    mktkey integer not null,
    hq_city char(20),
    hq_state char(20),
    district char(20),
    region char(20),
    constraint mkt_pkc primary key (mktkey));

create table store (
    storekey integer not null,
    mktkey integer not null,
    store_type char(10),
    store_name char(30),
    street char(30),
    city char(20),
    state char(5),
    zip char (10),
    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),
    class_desc char(60),
    primary key (classkey));

create table product (
    classkey integer not null,
    prodkey integer not null,
    prod_name char(30),
    pkg_type char(20),
    constraint prod_pkc primary key (classkey, prodkey),
    constraint prod_fkc foreign key (classkey)
        references class (classkey))
    maxrows per segment 2500;
```
Example: Building a Database

Creating the User Tables

```sql
create table promotion (
    promokey integer not null,
    promo_type integer not null,
    promo_desc char(40),
    value dec(7,2),
    start_date date,
    end_date date,
    primary key (promokey))
maxrows per segment 2500;
create table period (
    perkey integer not null,
    date date,
    day character(3),
    week integer,
    month character(5),
    qtr character(5),
    year integer,
    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,
    dollars dec(7,2),
    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))
maxrows per segment 60000;
data in (daily_data1, daily_data2)
segment by values of (perkey)
ranges (min:415, 415:max)
maxsegments 2
maxrows per segment 60000;
```

Note the following about the 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.
Example: Building a Database
Creating the User Tables

- Each column in a table has a declared datatype that corresponds to the data to be stored. Newaroma uses the following datatypes:
  - CHARACTER, which contains the specified number of characters
  - INTEGER (4 bytes)
  - DECIMAL (7,2), with a precision of 7 and a scale of 2 (4 bytes)
  - DATE (3 bytes)
- The Sales table is created using named segments.
- The other tables are all created using default segments.

For more information about planning and creating databases, refer to Chapter 4, “Planning a Database Implementation,” and Chapter 5, “Creating a Database.”

For a detailed description of the CREATE TABLE syntax and more information about datatypes, refer to the SQL Reference Guide.

Procedure

To create the user tables for Newaroma:

1. Invoke the RISQL Entry Tool by entering:

   ```
   % risql -d NEWAROMA system
   ```
   providing the logical database name and username.

2. Supply your password in response to the prompt:

   ```
   (C) Copyright 1991 - 1998, Red Brick Systems, Los Gatos, California, USA
   All rights reserved
   RISQL Entry Tool Version 5.1 (xxxx)
   Please type password: ********
   RISQL>
   ```
   **Note:** 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 by entering:

   ```
   RISQL> run /my_directory/aroma_inputs/aroma_create.risql;
   ```
   **Note:** You can also enter the statements interactively from the command line, but it is much easier and subject to fewer errors to run them from a file that you can edit.
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 you how to verify that they were built and how you can find additional information about the tables and indexes that you might need for various administrative tasks later.

**Note:** The RISQL Entry Tool and RISQL Reporter both require a semicolon to terminate SQL statements, as shown in this appendix. Other tools might not require nor accept them.

4. To verify that the tables were created, query the system table RBW_TABLES by entering a SQL statement similar to the following:

   ```sql
   RISQL> select * from rbw_tables where id > 0;
   ```

If the tables were created, the response is similar to the following (although more columns—DATETIME, SEGMENT_BY, PARTIAL and COMMENT—will be displayed and the lines will wrap):

   ```sql
   RISQL> select * from rbw_tables where id > 0;
   ```

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>CREATOR</th>
<th>ID</th>
<th>MAXSEGMENTS</th>
<th>MAXROWS_PER</th>
<th>MAXSIZE_ROW</th>
<th>INTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEAL</td>
<td>TABLE</td>
<td>SYSTEM</td>
<td>6</td>
<td>1</td>
<td>NULL</td>
<td>40370022</td>
<td>Y</td>
</tr>
<tr>
<td>ORDERS</td>
<td>TABLE</td>
<td>SYSTEM</td>
<td>11</td>
<td>1</td>
<td>2000</td>
<td>25165728</td>
<td>Y</td>
</tr>
<tr>
<td>SALES</td>
<td>TABLE</td>
<td>SYSTEM</td>
<td>13</td>
<td>2</td>
<td>60000</td>
<td>1444012</td>
<td>Y</td>
</tr>
<tr>
<td>PERIOD</td>
<td>TABLE</td>
<td>SYSTEM</td>
<td>8</td>
<td>1</td>
<td>2500</td>
<td>79429329</td>
<td>Y</td>
</tr>
<tr>
<td>SUPPLIER</td>
<td>TABLE</td>
<td>SYSTEM</td>
<td>7</td>
<td>1</td>
<td>NULL</td>
<td>17825724</td>
<td>Y</td>
</tr>
<tr>
<td>STORE</td>
<td>TABLE</td>
<td>SYSTEM</td>
<td>2</td>
<td>1</td>
<td>2500</td>
<td>18612153</td>
<td>Y</td>
</tr>
<tr>
<td>PROMOTION</td>
<td>TABLE</td>
<td>SYSTEM</td>
<td>5</td>
<td>1</td>
<td>2500</td>
<td>36175734</td>
<td>Y</td>
</tr>
<tr>
<td>PRODUCT</td>
<td>TABLE</td>
<td>SYSTEM</td>
<td>4</td>
<td>1</td>
<td>2500</td>
<td>36175734</td>
<td>Y</td>
</tr>
<tr>
<td>LINE_ITEMS</td>
<td>TABLE</td>
<td>SYSTEM</td>
<td>12</td>
<td>1</td>
<td>2000</td>
<td>65011464</td>
<td>Y</td>
</tr>
<tr>
<td>CLASS</td>
<td>TABLE</td>
<td>SYSTEM</td>
<td>3</td>
<td>1</td>
<td>NULL</td>
<td>27787158</td>
<td>Y</td>
</tr>
<tr>
<td>MARKET</td>
<td>TABLE</td>
<td>SYSTEM</td>
<td>1</td>
<td>1</td>
<td>NULL</td>
<td>25165728</td>
<td>Y</td>
</tr>
</tbody>
</table>

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:

```
RISQL> select * from rbw_indexes;
```

If the indexes were created, the response is similar to the following (although more columns—DATETIME, INTACT, PARTIAL, and COMMENT—will be displayed and the lines will wrap):

```
RISQL> select * 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_PK_IDX STORE BTREE MKTKEY SYSTEM 100 VALID
PRODUCT_PK_IDX PRODUCT BTREE CLASSKEY SYSTEM 100 VALID
ORDERS_PK1_IDX ORDERS BTREE PERKEY SYSTEM 100 VALID
ORDERS_PK2_IDX ORDERS BTREE SUPKEY SYSTEM 100 VALID
ORDERS_PK3_IDX ORDERS BTREE DEALKEY SYSTEM 100 VALID
LINE_ITEMS_STAR_IDX LINE_ITEMS STAR ORDER_NO SYSTEM 100 VALID
```

Note the following points about 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.
- Index types: Each table has a B-TREE index on its primary key, except for the Sales table (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. Look at the database directory to see what files were created by entering:

```
RISQL> !ls /my_directory/aroma_db ;
```

The exclamation mark (!) is an escape to the system shell. 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
```

If you do not use default segments for the tables and indexes, the PSUs will be found in the locations specified by the CREATE SEGMENT statements (or in default directories specified in the `rbw.config` file). Note that the PSUs for the Sales table are also in this directory. This is because 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 what table or index each segment belongs by entering:

```
RISQL> select name, tname, iname, NPSUS from rbw_segments;
```
Example: Building a Database
Creating the User Tables

The response is similar to the following (although the lines will wrap):

```sql
RISQL> select name, tname, iname, npsus from rbw_segments;

<table>
<thead>
<tr>
<th>NAME</th>
<th>TNAME</th>
<th>INAME</th>
<th>NPSUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBW_SYSTEM</td>
<td>NULL</td>
<td>NULL</td>
<td>5</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_1</td>
<td>MARKET</td>
<td>NULL</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_2</td>
<td>MARKET</td>
<td>MARKET_PK_IDX</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_3</td>
<td>STORE</td>
<td>NULL</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_4</td>
<td>STORE</td>
<td>STORE_PK_IDX</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_5</td>
<td>CLASS</td>
<td>NULL</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_6</td>
<td>CLASS</td>
<td>CLASS_PK_IDX</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_7</td>
<td>PRODUCT</td>
<td>NULL</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_8</td>
<td>PRODUCT</td>
<td>PRODUCT_PK_IDX</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_9</td>
<td>PROMOTION</td>
<td>NULL</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_10</td>
<td>PROMOTION</td>
<td>PROMOTION_PK_IDX</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_11</td>
<td>DEAL</td>
<td>NULL</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_12</td>
<td>DEAL</td>
<td>DEAL_PK_IDX</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_13</td>
<td>SUPPLIER</td>
<td>NULL</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_14</td>
<td>SUPPLIER</td>
<td>SUPPLIER_PK_IDX</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_15</td>
<td>PERIOD</td>
<td>NULL</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_16</td>
<td>PERIOD</td>
<td>PERIOD_PK_IDX</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_17</td>
<td>ORDERS</td>
<td>NULL</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_18</td>
<td>ORDERS</td>
<td>ORDERS_PK_IDX</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_19</td>
<td>LINE_ITEMS</td>
<td>NULL</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_20</td>
<td>LINE_ITEMS</td>
<td>LINE_ITEMS_PK_IDX</td>
<td>1</td>
</tr>
<tr>
<td>DAILY_DATA1</td>
<td>SALES</td>
<td>NULL</td>
<td>2</td>
</tr>
<tr>
<td>DAILY_DATA2</td>
<td>SALES</td>
<td>NULL</td>
<td>2</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_23</td>
<td>SALES</td>
<td>SALES_STAR_IDX</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_24</td>
<td>STORE</td>
<td>STORE_TGT_IDX</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_25</td>
<td>STORE</td>
<td>STORE_PK_IDX</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_26</td>
<td>PRODUCT</td>
<td>PRODUCT_PK_IDX</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_27</td>
<td>ORDERS</td>
<td>ORDERS_FK1_IDX</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_28</td>
<td>ORDERS</td>
<td>ORDERS_FK2_IDX</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_29</td>
<td>ORDERS</td>
<td>ORDERS_FK3_IDX</td>
<td>1</td>
</tr>
<tr>
<td>DEFAULT_SEGMENT_30</td>
<td>LINE_ITEMS</td>
<td>LINE_ITEMS_STAR_IDX</td>
<td>1</td>
</tr>
</tbody>
</table>

RISQL> 
```

Note the following points about segments:

- **Segment names:** Default segments are used for all the tables except the Sales table. The numeric suffixes on these names are automatically assigned by the system. The segments for the Sales table are specified in the CREATE SEGMENT and CREATE TABLE statements.
- **Additional information about each PSU is contained in the RBW_STORAGE system table.**
Writing the LOAD DATA Statements

The Table Management Utility (TMU) uses LOAD DATA statements to map the input data from the input record fields into 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 UNIX 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*1994-01-01*SA*1*JAN*Q1_94*1994
2*1994-01-02*SU*2*JAN*Q1_94*1994
3*1994-01-03*MO*2*JAN*Q1_94*1994
4*1994-01-04*TU*2*JAN*Q1_94*1994
5*1994-01-05*WE*2*JAN*Q1_94*1994
6*1994-01-06*TH*2*JAN*Q1_94*1994
7*1994-01-07*FR*2*JAN*Q1_94*1994
8*1994-01-08*SA*2*JAN*Q1_94*1994
9*1994-01-09*SU*3*JAN*Q1_94*1994
10*1994-01-10*MO*3*JAN*Q1_94*1994
11*1994-01-11*TU*3*JAN*Q1_94*1994
12*1994-01-12*WE*3*JAN*Q1_94*1994
13*1994-01-13*TH*3*JAN*Q1_94*1994
14*1994-01-14*FR*3*JAN*Q1_94*1994
15*1994-01-15*SA*3*JAN*Q1_94*1994
16*1994-01-16*SU*4*JAN*Q1_94*1994
...  
```

Note: This is only a partial list of the input records.
This input data will be stored in the Period table defined by the following
CREATE TABLE statement:

```
create table period (  
    perkey integer not null,  
    date date,  
    day character(3),  
    week integer,  
    month character(5),  
    qtr character(5),  
    year integer,  
    primary key (perkey))  
maxrows per segment 2500;
```

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 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);
```

Note the following about the LOAD DATA statement for the Period table:

- The records use a separated format with a star (*) separator.
- The load operation is done in REPLACE mode: Any existing data in the table
  is destroyed.
- The first field has the fieldtype “integer external,” is 4 characters long, and
  is to be stored in the Period table column named Perkey.
- The second field has the fieldtype “date” and has a format mask that
  specifies four digits for year and two digits for month and day; subfields are
  separated by a dash (—).
- The third field has the fieldtype “character,” is 3 characters long, and is to
  be stored in the Period table column named Day.
**Example: Building a Database**

**Writing the LOAD DATA Statements**

### 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
```

**Note:** This is only a partial list of the input records.

This data will be stored in the Product table defined by the following `CREATE TABLE` statement:

```
create table product (  
classkey integer not null,  
prodkey integer not null,  
prod_name char(30),  
pkg_type char(20),  
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 into 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)) ;
```
Example: Building a Database
Writing the LOAD DATA Statements

Note the following about 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 datatypes 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
13*San Jose*CA*San Francisco*West
14*San Francisco*CA*San Francisco*West
15*Oakland*CA*San Francisco*West
16*Los Angeles*CA*Los Angeles*West
17*Phoenix*AZ*Los Angeles*West
19*San Jose*CA*San Francisco*West
16*Oakland*CA*San Francisco*West
17*Los Angeles*CA*Los Angeles*West
19*Phoenix*AZ*Los Angeles*West

Note: This is only a partial list of the input records.

This data will be stored in the Market table defined by the following CREATE TABLE statement:

```sql
create table market (  
mktkey integer not null,  
hq_city char(20),  
hq_state char(20),  
district char(20),  
region char(20),  
constraint mkt_pkc primary key (mktkey));
```
Example: Building a Database
Writing the LOAD DATA Statements

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:

```sql
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)) ;
```

Note the following about 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:

```plaintext
0000000002 0000000002 0000000016 0000000008 0000000043.00  
0000000002 0000000004 0000000012 0000000001 0000000016 0000000009 0000000060.75  
0000000002 0000000001 0000000011 0000000001 0000000016 0000000040 00000000270.00  
0000000002 0000000005 0000000004 0000000005 0000000016 0000000004 0000000036.00  
0000000002 0000000001 0000000001 0000000001 0000000016 0000000011 0000000030.25  
0000000002 0000000001 0000000001 0000000001 0000000016 0000000010 00000000187.50  
0000000002 0000000001 0000000001 0000000001 0000000016 0000000005 00000000143.75  
0000000002 0000000001 0000000001 0000000001 0000000016 0000000002 00000000067.00  
0000000002 0000000001 0000000001 0000000001 0000000016 0000000011 00000000087.00  
0000000002 0000000001 0000000001 0000000001 0000000016 0000000012 00000000115.50  
0000000002 0000000001 0000000001 0000000001 0000000016 0000000018 00000000058.50  
0000000002 0000000001 0000000001 0000000001 0000000016 0000000017 00000000136.00  
0000000002 0000000001 0000000001 0000000001 0000000016 0000000018 00000000074.75  
0000000002 0000000001 0000000001 0000000001 0000000016 0000000014 00000000101.50  
0000000002 0000000001 0000000001 0000000001 0000000016 0000000018 00000000063.00  
0000000002 0000000001 0000000001 0000000001 0000000016 0000000012 00000000089.00  
0000000002 0000000001 0000000001 0000000001 0000000016 0000000016 00000000036.00  
0000000002 0000000001 0000000001 0000000001 0000000016 0000000010 00000000040.00
```

Note: This is only a partial list of the input records.
This input data will be stored in the Sales table defined by the following CREATE TABLE statement:

```sql
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,
   dollars dec(7,2),
   primary key (perkey, classkey, prodkey, storekey, promokey),
   foreign key (perkey) references period (perkey),
   foreign key (classkey, prodkey) references product (classkey, prodkey),
   foreign key (storekey) references store (storekey),
   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 into a column in the corresponding row in the Sales table looks like this:

```sql
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)='%');
```

Note the following about 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+11) plus 1 character for each NULLIF (1+1+1+1+1+1+1) plus the newline character equals the record length (86).
Example: Building a Database

Loading the Data

Load the Newaroma data by running the TMU with the control file containing 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:

```sql
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)) ;
```
Example: Building a Database
Loading the Data

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)= ' %',
storekeyposition(38) integerexternal(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)= ' %');

Note the following about the LOAD DATA statements and the control file:

• The input file name must be relative to the directory from which you invoke the TMU, or it must be a full pathname.
• The referenced (dimension) tables (Period, Product, and Market) must be loaded before the referencing (fact) table (Sales).

Procedure

To load the data into the aroma_db database:

1. Log in as the redbrick user.
2. Verify that the redbrick_dir/bin directory is in your path.
3. Change to the aroma_inputs directory, which contains the aroma.tmu file that contains the LOAD DATA statements for all the tables in the Aroma database.
4. Run the TMU by entering:
   $ rb_tmu -d /my_directory/aroma_db aroma.tmu system cryptic

The TMU responds with messages similar to the following:

(C) Copyright 1991-1998 Red Brick Systems, Inc., Los Gatos, California, USA
All rights reserved.
Version 5.1.1 (8003)
** 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 ** (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
Example: Building a Database

Loading the Data

invalid.
** INFORMATION ** (352) Row 102 of index PROMOTION_PK_IDX is out of sequence. Switching to standard optimized index building. Loading continues...
** INFORMATION ** (315) Finished file aroma_promo.txt. 194 rows read from this file.
** INFORMATION ** (513) Starting merge phase of index building PROMOTION_PK_IDX.
** INFORMATION ** (367) Rows: 194 inserted. 0 updated. 0 discarded. 0 skipped.
** INFORMATION ** (500) Time = 00:00:00.37 cp time, 00:00:00.90 time, Logical IO count=76, Blk Reads=3, Blk Writes=36
** INFORMATION ** (366) Loading table PERIOD.
** WARNING ** (8023) Any existing rows in tables that reference table PERIOD may now be invalid.
** INFORMATION ** (315) Finished file aroma_period.txt. 821 rows read from this file.
** INFORMATION ** (367) Rows: 821 inserted. 0 updated. 0 discarded. 0 skipped.
** INFORMATION ** (500) Time = 00:00:00.05 cp time, 00:00:00.63 time, Logical IO count=87, Blk Reads=4, Blk Writes=38
** INFORMATION ** (366) Loading table SALES.
** INFORMATION ** (352) Row 3 of index SALES_STAR_IDX is out of sequence. Switching to standard optimized index building. Loading continues...
** INFORMATION ** (315) Finished file aroma_sales.txt. 69941 rows read from this file.
** INFORMATION ** (513) Starting merge phase of index building SALES_STAR_IDX.
** INFORMATION ** (367) Rows: 69941 inserted. 0 updated. 0 discarded. 0 skipped.
** INFORMATION ** (500) Time = 00:00:03.97 cp time, 00:00:08.63 time, Logical IO count=755, Blk Reads=736, Blk Writes=699

Note the following about 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 table 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 data with respect to the STAR index that was created on the Sales table, based only on the order of its leading foreign key reference; but it is unordered data with respect to the other tables based on the order of their foreign key references.

If you want other users to access this database, you must provide them database access with the GRANT command. You should also define the database with a logical name in the rbw.config file to simplify database selection.
Example: Building a Database

Verifying the Database

Verify that the tables were built by entering 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;

<table>
<thead>
<tr>
<th>CLASSKEY</th>
<th>PRODKEY</th>
<th>PROD_NAME</th>
<th>PKG_TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>Veracruzano</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Xalapa Lapa</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>Colombiano</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>Expresso XO</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>La Antigua</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>Lotta Latte</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>Cafe Au Lait</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>22</td>
<td>NA Lite</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
<td>Aroma Roma</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>31</td>
<td>Demitasse Ms</td>
<td>No pkg</td>
</tr>
</tbody>
</table>

RISQL>

Summary

To summarize the steps in building a database:

1. Determine the tables needed for the database and the columns and datatypes for each table.
2. Log in as the redbrick user and create the database with rb_creator. 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 warehouse format. You can then combine these LOAD DATA statements into a single TMU control file or run them individually.
6. As the redbrick user, load the data with the rb_tmu and the control file(s).
7. Grant access to users.
Example: Building a Database

Summary
When 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
- License keys for warehouse options (keyword LICENSE_KEY)
- Tuning parameters that affect warehouse performance (keyword TUNE)
- Option parameters that affect warehouse behavior (keyword OPTION)
- Other system configuration settings (keywords LOCALE, RBWMON, FILLFACTOR, QUERY_TEMPSPACE, INDEX_TEMPSPACE, ADMIN, and PASSWORD)
- Logical database names and database locations (keyword DB)

Many of these parameters can also be set with SQL SET commands or with TMU control statements.

The following example shows an rbw.config file. 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 can be modified by the warehouse administrator. Values indicated as `<value>` will be replaced in the rbw.config file at your site by the actual value.

```bash
# This file contains various parameters for the Red Brick Warehouse Daemon
# (rbwapid), Server (rbwsvr), and Table Management Utilities (rb_ptmu)
# and rb_tmu
# The notation {ON | OFF} means use either ON, or OFF, etc.
# The default is the first option shown
# This config file was created by user: redbrick

# The following is used for IPC key values. Note that for shared memory
# and semaphores, the key values will range from the IPC key number to
# IPC key + MAX_SERVERS.
RB_HOST SHMEM 100

# The following is used to specify the communication port for connections
# to the Red Brick Warehouse rbwapid daemon.
RB_HOST SERVER <host>:<port_number>
```
# The following value controls the maximum number of concurrent Red Brick Warehouse sessions that can be started by the rbwapid daemon.
RBWAPI MAX_SERVERS 50

# Daemon Start-up user exit script for cleaning up spill files.
RBWAPI CLEANUP_SCRIPT <redbrick_dir>/bin/rb_sample.cleanup

# The following value can be used to specify the location of the Red Brick Warehouse server image
RBWAPI SERVER_NAME <redbrick_dir>/bin/rbwsvr

# The following value indicates the maximum size of the logfile.
# Once the daemon writes this many lines into the file, it will rename it to rbwapid.log_old and a new logfile will be started.
RBWAPI LOGFILE_SIZE 1000

# Message base details (location and language)
NLS_LOCALE MESSAGE_DIR <redbrick_dir>/messages
NLS_LOCALE LOCALE English_UnitedStates.US-ASCII@Binary

# Server Monitor daemon sampling interval (secs)
RBWMON INTERVAL 120

# Automatic referenced table row generation
#OPTION AUTOROWGEN {OFF | ON}

# Divide by zero control
#OPTION ARITHABORT {ON | OFF}

# Deadlock control
#OPTION ALLOW_POSSIBLE_DEADLOCK {OFF | ON}

# Enable cross join
#OPTION CROSS_JOIN {OFF | ON}

# Datatype for COUNT function
#OPTION COUNT_RESULT {INTEGER | DECIMAL | DEC | INT}

# Temporary Table creation authorization
#OPTION GRANT_TEMP_RESOURCE_TO_ALL {ON | OFF}

# Generate log records for Advisor?
#OPTION ADVISOR_LOGGING { ON | ON_WITH_CORR_SUB | OFF }

# Automatically rewrite queries using available precomputed views?
#OPTION PRECOMPUTED_VIEW_QUERY_REWRITE { OFF | ON }

# Automatically invalidate precomputed views based on table data change?
#OPTION AUTO_INVALIDATE_PRECOMPUTED_VIEWS { OFF | ON }

# Use invalid precomputed views when rewriting queries?
#OPTION USE_INVALID_PRECOMPUTED_VIEWS { OFF | ON }
Sample rbw.config File

B-4  Warehouse Administrator’s Guide for UNIX Platforms
# Parallel Table Management Utility License
#LICENSE_KEY PTMU_OPTION xxxxxxxxxx

# RISQL Reporter License
#LICENSE_KEY RISQL_REPORTER xxxxxxxxxx

# Workgroups Parallel Option License
#LICENSE_KEY WORKGROUPS_PARALLEL_OPTION xxxxxxxxxx

# Enterprise Control and Coordination License
#LICENSE_KEY ENTERPRISE_CONTROL_AND_COORDINATION xxxxxxxxxx

# Enterprise Copy Management License
#LICENSE_KEY ENTERPRISE_COPY_MANAGEMENT xxxxxxxxxx

# Red Brick Data Mine License
#LICENSE_KEY RED_BRICK_DATA_MINE xxxxxxxxxx

# Unlimited SQL BackTrack license
#LICENSE_KEY SQL_BACKTRACK_UNLIMITED xxxxxxxxxx

# SQL BackTrack license for Workgroups
#LICENSE_KEY SQL_BACKTRACK_FOR_WORKGROUPS xxxxxxxxxx

# Red Brick Vista License
#LICENSE_KEY RED_BRICK_VISTA xxxxxxxxxx

###########################################################
#
# TUNE section: Optional tuning & performance parameters
#
###########################################################
# Number of TMU Buffer cache pages
#TUNE TMU_BUFFERS 128

# Tuning parameter for parallel query
#TUNE ROWS_PER_SCAN_TASK 2147483647
#TUNE ROWS_PER_FETCH_TASK 2147483647
#TUNE ROWS_PER_JOIN_TASK 2147483647
#TUNE QUERYPROCS 0
#TUNE TOTALQUERYPROCS 0

#TUNE FORCE_SCAN_TASKS {OFF | <num_tasks>}
#TUNE FORCE_FETCH_TASKS {OFF | <num_tasks>}
#TUNE FORCE_JOIN_TASKS {OFF | <num_tasks>}
#TUNE FORCE_AGGREGATION_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>
Configuration File
Sample rbw.config File

#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}

# Enable parallelism for aggregation partitioned by the GROUP BY columns
#TUNE PARTITIONED_PARALLEL_AGGREGATION {ON | OFF}

# Result buffer configuration
# TUNE RESULT_BUFFER UNLIMITED
# TUNE RESULT_BUFFER_FULL_ACTION PAUSE

# PTMU Parallel Processing Tasks
#TUNE TMU_CONVERSION_TASKS <num_tasks>
#TUNE TMU_INDEX_TASKS <num_tasks>

# PTMU Serial Mode operation
#TUNE TMU_SERIAL_MODE {ON | OFF}

# Index Fillfactor parameters
#FILLFACTOR PI 100
#FILLFACTOR STAR 100
#FILLFACTOR SI 100

# Optimized index build parameter
#OPTION TMU_OPTSIMIZE {OFF | ON}

# Index Temporary Space parameters
# Always specify THRESHOLD before MAXSPILLSIZE in this
# configuration file
# Specify multiple DIRECTORYs by having multiple
# TUNE INDEX_TEMPSPACE_DIRECTORY entries
#TUNE INDEX_TEMPSPACE_THRESHOLD 10M
#TUNE INDEX_TEMPSPACE_MAXSPILLSIZE 1G
#TUNE INDEX_TEMPSPACE_DIRECTORY /tmp

# Query Temporary Space parameters
# Always specify QUERY_MEMORY_LIMIT before QUERY_TEMPSPACE_MAXSPILLSIZE
# in this configuration file
# Specify multiple DIRECTORYs by having multiple
# TUNE QUERY_TEMPSPACE_DIRECTORY entries
#TUNE QUERY_MEMORY_LIMIT 50M
#TUNE QUERY_TEMPSPACE_MAXSPILLSIZE 1G
#TUNE QUERY_TEMPSPACE_DIRECTORY /tmp
# 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
#Number of load operations stored in RBW_LOADINFO system table
#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>

# 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}
# Configuration File
Sample rbw.config File

# 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 REPORT_INTERVAL 1
# ADMIN RENICE_COMMAND <full_path_of_a_renice_executable_file>

# Create Advisor log files at system startup, and log Advisor records?
# ADMIN ADVISOR_LOGGING { OFF | ON }

# Advisor logging directory
# ADMIN ADVISOR_DIRECTORY <RB_CONFIG>/logs

# Advisor log maximum size control
# ADMIN ADVISOR_MAXSIZE 0

# PASSWORD section
#
# Number of days allowed between password changes
#PASSWORD EXPIRATION_DAYS 0
# Number of days before password expires that user will
# begin to be warned that password is about to expire
#PASSWORD EXPIRATION_WARNING_DAYS 0
# Minimum number of days that must pass between
# password changes
#PASSWORD CHANGE_MINIMUM_DAYS 0
# Number of previously used passwords on each account
# against which new passwords will be compared for
# uniqueness
#PASSWORD RESTRICT_PREVIOUS 0
# The following three parameters control the requirements
# for complex passwords. These parameters specify the
# number of characters of the three types that must be
# present in each new password.
#PASSWORD COMPLEX_NUM_ALPHA 0
#PASSWORD COMPLEX_NUM_NUMERICS 0
#PASSWORD COMPLEX_NUM_PUNCTUATION 0
# Minimum required length for new passwords
#PASSWORD MINIMUM_LENGTH 0
# Number of failed login attempts that will result in a
# user account being locked
#PASSWORD LOCK_FAILED_ATTEMPTS 0
# Number of hours an account will remain locked
#PASSWORD LOCK_PERIOD_HOURS 0

###########################################################
# DATABASE section: Add additional entries as databases
# are created
#
###########################################################
# Logical database name mappings
DB AROMA  <redbrick_dir>/aroma_db
DB ADMIN   <redbrick_dir>/admin_db
**Configuration File**

**Description of File Elements**

This section provides a brief description of the rbw.config file entries present when Red Brick Warehouse is installed with the standard installation procedure.

**Configuration Information**

**RB_HOST SHMEM**

(Not present on all platforms) Defines the base number for the IPC key range. IPC key values range from SHMEM to SHMEM plus MAX_SERVERS.

Default: 100 (base 16 integer)

This range of numbers must be unique and should be assigned to Red Brick Warehouse by the system administrator or person in charge of maintaining IPC key numbers.

**RB_HOST MAPFILE**

(Not present on all platforms) Specifies the file used as a shared memory map file.

Default: ./R.B_HOST.mapfile

**RB_HOST SERVER**

Defines the host name and port number for all connections made to Red Brick Warehouse.

Default: The host name and port number used for the warehouse installation.

**RBWAPI MAX_SERVERS**

Defines the maximum concurrent connections (users) supported by the warehouse daemon.

Default: 50 (base 10 integer)

**RBWAPI CLEANUP_SCRIPT**

Defines a spill file cleanup script to be executed by the daemon upon startup. A sample script named rb_sample.cleanup is shipped with the warehouse in the redbrick/bin directory.

**RBWAPI SERVER_NAME**

Specifies the location of the warehouse server image (rbwsvr).
RBWAPI LOGFILE_SIZE
Defines the maximum number of lines of the warehouse logfile, rbwapid.log. When this limit is reached, the warehouse daemon renames the logfile rbwapid.log_old, and starts a new logfile.
Default: 1000

NLS_LOCALE MESSAGE_DIR
Specifies the directory used for the error message files.
Defaults: ./messages

NLS_LOCALE LOCALE
Specifies the language, territory, and sort order for the Red Brick Warehouse server.
Default: English_UnitedStates.US-ASCII@Binary

RBWMON INTERVAL
Specifies the interval at which the server monitoring daemon (rbw.servermon) checks for server processes.
Default: 120 seconds

OPTION AUTOROWGEN
Turns automatic row generation for referenced tables on or off during TMU load operations.
Default: OFF

OPTION ARITHABORT
Specifies that arithmetic operations should abort on divide-by-zero errors.

OPTION ALLOW_POSSIBLE_DEADLOCKS
Specifies that 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 CROSS_JOIN
Specifies whether cross joins are allowed.
Default: OFF

OPTION COUNT_RESULT
Specifies INTEGER or DECIMAL datatype values for the COUNT function. If tables have more than $2^{32}$ rows, COUNT_RESULT should be set to DECIMAL.
Default: INTEGER
**License Keys**

**LICENSE_KEY RED_BRICK_WAREHOUSE_X**
Specifies your license key for Red Brick Warehouse, where X represents a license for a particular number of users.

**LICENSE_KEY RED_BRICK_WAREHOUSE_FOR_WORKGROUPS_X**
Specifies your license key for Red Brick Warehouse, where X represents a license for 1, 5, 10, 20, or 30 users.

**LICENSE_KEY AUTO_AGGREGATE**
Specifies your license key for the Auto Aggregate Option.

**LICENSE_KEY BACKUP_RESTORE**
Specifies your license key for the TMU Incremental Backup and Restore option.

**LICENSE_KEY PTMU_OPTION**
Specifies your license key for the Parallel Table Management Utility option.

**LICENSE_KEY RISQL_REPORTER**
Specifies your license key for the RISQL Reporter option.

**LICENSE_KEY WORKGROUPS_PARALLEL_OPTION**
Specifies your license key for the Workgroups Parallel option for Red Brick Warehouse for Workgroups.

**LICENSE_KEY ENTERPRISE_CONTROL_ANDCOORDINATION**
Specifies your license key for the Enterprise Control and Coordination option.

**LICENSE_KEY ENTERPRISE_COPY_MANAGEMENT**
Specifies your license key for the Enterprise Control and Coordination option.

**LICENSE_KEY SQL_BACKTRACK_X**
Specifies your license key for SQL-BackTrack for Red Brick Warehouse, where X represents an unlimited or a workgroups license.

**LICENSE_KEY RED_BRICK_VISTA**
Specifies your license key for the Red Brick Vista option.
Tuning and Performance Parameters

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 command, in order for the OPTION ADVISOR_LOGGING command 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 the Red Brick Vista option.
Possible values: ON, OFF, ON_WITH_CORR_SUB
Default: ON

TUNE TMU_BUFFERS
Specifies the buffer cache size in 8-kilobyte blocks for the TMU; values range from 128 blocks to 8208 blocks.
Default: 128 blocks

TUNE ROWS_PER_SCAN_TASK
Specifies the minimum estimated number of rows to be scanned by a relation scan before a parallel relation scan is performed.
Default: 2,147,483,647 (Do not enter commas.)

TUNE ROWS_PER_FETCH_TASK
Specifies the minimum estimated number of data rows returned during the fetch portion of STARjoin processing before parallel fetch processes are used.
Default: 2,147,483,647 (Do not enter commas.)

TUNE ROWS_PER_JOIN_TASK
Specifies the minimum estimated number of index entries returned during the join processing (index-probing) portion of STARjoin processing before parallel join processes are used.
Default: 2,147,483,647 (Do not enter commas.)

TUNE QUERYPROCS
Specifies upper limit on the number of processes used to process a single query.
Default: 0
**TUNE TOTALQUERYPROCS**
Specifies upper limit on the number of processes used at one time for parallel query processing across all servers under a single warehouse daemon (in addition to number specified as MAX_SERVERS parameter).
Default: 0

**TUNE FORCE_SCAN_TASKS**
Specifies the number of parallel processes to use in a relation scan.
Default: OFF

**TUNE FORCE_FETCH_TASKS**
Specifies the number of parallel processes to use fetching row data for a single query.
Default: OFF

**TUNE FORCE_JOIN_TASKS**
Specifies the number of parallel processes to use processing joins for a single query.
Default: OFF

**TUNE FORCE_HASHJOIN_TASKS**
Specifies the number of parallel processes to use processing hybrid hash joins.
Default: OFF

**TUNE FORCE_AGGREGATION_TASKS**
Specifies the number of parallel processes to use processing aggregation functions.
Default: OFF

**TUNE PARTITIONED_PARALLEL_AGGREGATION**
Specifies if parallel processes are used to process aggregation functions.
Default: OFF

**TUNE PARALLEL_HASHJOIN**
Specifies if parallel processes are used to process hybrid hash joins.
Default: ON

**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

TUNE TMU_CONVERSION_TASKS
Specifies the number of parallel tasks available for conversion of data using the TMU.
Default: none

TUNE TMU_INDEX_TASKS
Specifies the number of parallel tasks available for building primary indexes using the TMU.
Default: none

TUNE TMU_SERIAL_MODE
Specifies whether the PTMU runs in serial or parallel mode.
Default: OFF

TUNE FILE_GROUP
Defines disk groups for purposes of reducing disk seek contention.
Default: none

TUNE GROUP
Defines the maximum number of parallel processes per query that access the named disk group concurrently.
Default: 1 process per disk group per query

FILLFACTOR PI, FILLFACTOR STAR, FILLFACTOR SI
Specifies a fillfactor percentage to use when creating new index nodes for the primary, STAR, and secondary (user-defined) indexes, respectively.
Default: 100

OPTION TMU_OPTIMIZE
Turns optimized index-building on or off for the TMU.
Default: OFF
**Configuration File**

**Description of File Elements**

- **TUNE INDEX_TEMPSPACE_THRESHOLD**
  Specifies the file size (in kilobytes or megabytes) at which spill files for index building are created.
  Default size: 1 megabyte
  Default units: kilobytes

- **TUNE INDEX_TEMPSPACE_MAXSPILLSIZE**
  Specifies maximum size in kilobytes (K), megabytes (M), or gigabytes (G) to which a file for index building can grow.
  Default: 1 gigabyte

- **TUNE INDEX_TEMPSPACE_DIRECTORY**
  Specifies the directory in which spill files for index building are created.
  Specify multiple entries for multiple directories with one directory per TUNE INDEX_TEMPSPACE_DIRECTORY parameter.
  Default: /tmp

- **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 megabyte
  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

- **OPTION PRECOMPUTED_VIEW_QUERY_REWRITE**
  Turns the aggregate query rewrite system ON or OFF. Only valid with the Red Brick Vista option.
  Default: ON
OPTION AUTO_INVALIDATE_PRECOMPUTED_VIEWS
Automatically invalidates all the precomputed views that reference any detail table whose contents are modified with inserts, updates, and deletes or LOAD DATA operations after the views are created. If set to OFF, precomputed views must be marked invalid manually with the SET PRECOMPUTED VIEW view_name INVALID command. Only valid with the Red Brick Vista option. 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 into the detail table), to rewrite queries with the Red Brick Vista query rewrite engine. Only valid with the Red Brick Vista option. Default: OFF

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 physical storage units (PSUs) of the default temporary index segments for temporary tables. Default: current database directory

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. This parameter does not require the Enterprise Control and Coordination option. Default: ON

**Default Parameters**

DEFAULT ROWCOUNT
Specifies the maximum number of rows returned before the server stops the execution of a query. A value of zero (0) turns off the restriction on row retrieval. Default: 0


**Configuration File**

**Description of File Elements**

**DEFAULT INFO_MESSAGE_LIMIT**
Specifies the maximum number of informational messages ("STASTISTICS" and "INFORMATION") returned per query.
Default: 1,000

**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**

**OPTION DEFAULT_DATA_SEGMENT**
Specifies a pathname to a directory in which to place default data segments. (Do not use if warehouse contains multiple databases with default segments.)
Default: directory defined by RB_PATH

**OPTION DEFAULT_INDEX_SEGMENT**
Specifies a pathname to a directory in which to place default index segments. (Do not use if warehouse contains multiple databases with default segments.)
Default: directory defined by RB_PATH

**OPTION SEGMENTS**
Specifies whether user-defined segments should be dropped or kept when the table or index associated with them is dropped. (Default segments are always dropped.)
Default: KEEP

**OPTION IGNORE_PARTIAL_INDEXES**
Specifies that a query should ignore partial indexes and consider only fully-available indexes.
Default: OFF

**OPTION PARTIAL_AVAILABILITY**
Specifies behavior when a query attempts to access a partially available table or index.
Default: PRECHECK
**OPTION IGNORE_OPTICAL_INDEXES**  
Specifies whether to use indexes stored on optical segments or not.  
Default: OFF

**OPTION OPTICAL_AVAILABILITY**  
Specifies the query behavior with respect to optical segments.  
Default: WAIT_NONE

**ADMIN Parameters**

**Note:** The ADMIN parameters are used only by tools and features in the Enterprise Control and Coordination option.

**ADMIN ACCOUNTING**  
Specifies whether the log daemon turns on the accounting feature upon daemon startup.  
Default: OFF

**ADMIN ACCT_DIRECTORY**  
Specifies the location of the file containing the accounting records.  
Default: "$RB_CONFIG>/logs"

**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 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. Only valid with the Red Brick Vista option.  
Default: OFF
**ADMIN ADVISOR_LOG_DIRECTORY**
Specifies the location of the file containing the Advisor log records. Only valid with the Red Brick Vista option.
Default: <$RB_CONFIG>/logs

**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.)

**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

**ADMIN LOG_MAXSIZE**
Sets maximum logging file size; the minimum value is 10,240 kilobytes.
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

**ADMIN REPORT_INTERVAL**
Sets the maximum interval (in minutes) between dynamic system table refreshes.
Default: 1 minute

**ADMIN RENICE_COMMAND**
Specifies the full pathname of a UNIX renice executable file that changes user priorities.
Default: none

**PASSWORD Parameters**

*Note:* The PASSWORD parameters are used only by the password security feature in the Enterprise Control and Coordination option.

**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 RESTRICT_PREVIOUS**
Sets minimum number of unique passwords that must be used before an expired password can be reused.
Default: unlimited

**PASSWORD COMPLEX_NUM_ALPHA**
Specifies minimum number of alphabetic characters that must be used in each new password.
Default: 0
Configuration File
Description of File Elements

**PASSWORD COMPLEX_NUM_NUMERICS**
Specifies minimum number of numeric characters that must be used in each new password.
Default: 0

**PASSWORD COMPLEX_NUM_PUNCTUATION**
Specifies minimum number of punctuation characters that must be used in each new password.
Default: 0

**PASSWORD MINIMUM_LENGTH**
Specifies minimum number of total characters that must be used in each new password.
Default: 0

**PASSWORD LOCK_FAILED_ATTEMPTS**
Sets maximum number of failed database access attempts before a user is locked out of the database.
Default: 0

**PASSWORD LOCK_PERIOD_HOURS**
Sets number of hours a locked account remains locked.
Default: 0

Database Entries

**DB AROMA**
Specifies the mapping between the AROMA logical database name and its physical location. If the Aroma database is installed, then this line is present in the configuration file.
Default: DB AROMA <redbrick_dir>/aroma_db

**DB ADMIN**
Specifies the mapping between the ADMIN logical database name and its physical location. This database must be present in order to take full advantage of the Enterprise Control and Coordination option features.
Default: DB ADMIN <redbrick_dir>/admin_db

Note: The ADMIN and AROMA databases do not count against the two-database limit imposed on Red Brick Warehouse for Workgroups installations.
**Summary of Configuration Parameters**

The following table lists the various parameters that affect the warehouse 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.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Set with</th>
<th>Affects</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>rbw.config</code></td>
<td><code>sql</code></td>
</tr>
<tr>
<td>RB_HOST SHMEM</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RB_HOST MAPFILE</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RB_HOST SERVER</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>RBWAPI parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAX_SERVERS</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>SERVER_NAME</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>LOGFILE_SIZE</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>NLS_LOCALE parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MESSAGE_DIR</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LOCALE</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RBMON INTERVAL</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>LICENSE_KEY values</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>TUNE parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FORCE_SCAN_TASK</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FORCE_FETCH_TASK</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FORCE_JOIN_TASK</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FORCE_HASHJOIN_TASK</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FORCE_AGGREGATION_TASK</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ROWS_PER_SCAN_TASK</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ROWS_PER_FETCH_TASK</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ROWS_PER_JOIN_TASK</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PARALLEL_HASHJOIN</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PARTITIONED_PARALLEL_AGREGATION</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>QUERYPROCS</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
## Configuration File

### Summary of Configuration Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Set with</th>
<th>Affects</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTALQUERYPROCS</td>
<td>rbw.config</td>
<td>✓</td>
</tr>
<tr>
<td>FILE_GROUP</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GROUP</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FILLFACTOR SI, PI, STAR</td>
<td>✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>OPTION parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEFAULT_DATA_SEGMENT</td>
<td>✓ ✓</td>
<td>✓</td>
</tr>
<tr>
<td>DEFAULT_INDEX_SEGMENT</td>
<td>✓ ✓</td>
<td>✓</td>
</tr>
<tr>
<td>IGNORE_PARTIAL_INDEXES</td>
<td>✓ ✓</td>
<td>✓</td>
</tr>
<tr>
<td>PARTIAL_AVAILABILITY</td>
<td>✓ ✓</td>
<td>✓</td>
</tr>
<tr>
<td>SEGMENTS (DROP, KEEP)</td>
<td>✓ ✓</td>
<td>✓</td>
</tr>
<tr>
<td>ARITHABORT</td>
<td>✓ ✓</td>
<td>✓</td>
</tr>
<tr>
<td>AUTOROWGEN</td>
<td>✓ ✓</td>
<td>✓</td>
</tr>
<tr>
<td>COUNT_RESULT</td>
<td>✓ ✓</td>
<td>✓</td>
</tr>
<tr>
<td>CROSS_JOIN</td>
<td>✓ ✓</td>
<td>✓</td>
</tr>
<tr>
<td>ADVISOR_LOGGING</td>
<td>✓ ✓</td>
<td>✓</td>
</tr>
<tr>
<td>PRECOMPUTED_VIEW_QUERY_REWRITE</td>
<td>✓ ✓</td>
<td>✓</td>
</tr>
<tr>
<td>AUTO_INVALIDATE_PRECOMPUTED_VIEWS</td>
<td>✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>USE_INVALID_PRECOMPUTED_VIEWS</td>
<td>✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>QUERY_MEMORY_LIMIT</td>
<td>✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>QUERY_TEMPSPACE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAXSPILLSIZE, DIRECTORY</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

B-24   Warehouse Administrator’s Guide for UNIX Platforms
### Summary of Configuration Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>ADMIN parameters</th>
<th>ACCOUNTING*</th>
<th>ACCT_DIRECTORY</th>
<th>ACCT_LEVEL*</th>
<th>ACCT_MAXSIZE*</th>
<th>ADVISOR_LOGGING*</th>
<th>ADVISOR_LOG_DIRECTORY</th>
<th>ADVISOR_LOG_MAXSIZE</th>
<th>LOGGING*</th>
<th>LOG_Audit_LEVEL*</th>
<th>LOG_DIRECTORY</th>
<th>LOG_ERROR_LEVEL*</th>
<th>LOG_MAXSIZE*</th>
<th>LOG_OPERATIONAL_LEVEL*</th>
<th>LOG_SCHEMA_LEVEL*</th>
<th>LOG_USAGE_LEVEL*</th>
<th>REPORT_INTERVAL</th>
<th>RENICE_COMMAND</th>
<th>PASSWORD parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set with</td>
<td>rbw.config SQL SET TMU statement</td>
<td>Server</td>
<td>TMU</td>
<td>Daemon</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACCOUNTING*</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ACCT_DIRECTORY</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ACCT_LEVEL*</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ACCT_MAXSIZE*</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ADVISOR_LOGGING*</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ADVISOR_LOG_DIRECTORY</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ADVISOR_LOG_MAXSIZE</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LOGGING*</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LOG_Audit_LEVEL*</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LOG_DIRECTORY</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LOG_ERROR_LEVEL*</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LOG_MAXSIZE*</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LOG_OPERATIONAL_LEVEL*</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LOG_SCHEMA_LEVEL*</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LOG_USAGE_LEVEL*</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>REPORT_INTERVAL</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RENICE_COMMAND</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PASSWORD parameters</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CHANGE_MINIMUM_DAYS</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>COMPLEX_NUM_ALPHA</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>EXPIRATION_DAYS</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LOCK_FAILED_ATTEMPTS</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RESTRICT_PREVIOUS</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

* These parameters can be set with an SQL ALTER SYSTEM command.
## Configuration File

### Summary of Configuration Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Set with</th>
<th>Affects</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rbw.config</td>
<td>SQL</td>
</tr>
<tr>
<td>DB logical database names</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LOCK (WAIT,NO WAIT)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LOCK (table, database)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ORDER_BY_ASC_NULL, ORDER_BY_DESC_NULL</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
This appendix describes the system catalog, the system tables, and the Dynamic Statistic Tables (DSTs) for Red Brick Warehouse. The column name, datatype, and a description of the column is provided for each table.

This appendix contains the following sections:

- System Catalog
- Dynamic Statistic Tables
- Datatypes and Their Sizes
Red Brick Warehouse maintains system tables that describe the data stored in user databases. Contents of the system tables can be viewed with a SELECT statement, which may include system-table joins. However, the data in the system tables is read-only; it cannot be inserted, updated, or deleted.

The following command 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 a user displays information stored in a system table, the user sees only tables and information about those tables that the user created or has permission to access. The RBW_TABAUTH and RBW_USERAUTH system tables define the tables that each user creates and has permission to access.

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Table Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBW_COLUMNS</td>
<td>Describes the columns from all objects in RBW_TABLES.</td>
</tr>
<tr>
<td>RBW_CONSTRAINTS</td>
<td>Records the names of primary key and foreign key constraints defined in CREATE TABLE statements.</td>
</tr>
<tr>
<td>RBW_CONSTRAINT_COLUMNS</td>
<td>Records the names of columns that comprise the primary key and foreign key constraints defined in CREATE TABLE statements.</td>
</tr>
<tr>
<td>RBW_HIERARCHIES</td>
<td>Shows the relationships within defined hierarchies.</td>
</tr>
<tr>
<td>RBW_INDEXCOLUMNS</td>
<td>Records the columns that form the key of the indexes listed in RBW_INDEXES.</td>
</tr>
<tr>
<td>RBW_INDEXES</td>
<td>Describes the indexes on all tables listed in RBW_TABLES.</td>
</tr>
<tr>
<td>RBW_LOADINFO¹</td>
<td>Provides statistics on recent load operations.</td>
</tr>
<tr>
<td>RBW_MACROS</td>
<td>Describes all macros defined in the database.</td>
</tr>
<tr>
<td>RBW_OPTIONS</td>
<td>Displays current settings for database parameters that can be tuned.</td>
</tr>
</tbody>
</table>

¹ RBW_LOADINFO contains statistics only if the load is executed with the RBW_LOAD parameter.
<table>
<thead>
<tr>
<th>Table Name</th>
<th>Table Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBW_PRECOMPVIEW_UTILIZATION</td>
<td>Advisor system table for views that are defined in the database (refer to the Red Brick Vista User’s Guide).</td>
</tr>
<tr>
<td>RBW_PRECOMPVIEW_COLUMNS</td>
<td>Shows the relationships between columns in aggregate tables and precomputed views.</td>
</tr>
<tr>
<td>RBW_RELATIONSHIPS</td>
<td>Lists tables that share primary key–foreign key relationships and shows the constraint names applied to those relationships.</td>
</tr>
<tr>
<td>RBW_ROLE_MEMBERS</td>
<td>Describes the relationship of all user-created roles to the users and roles that have been granted to them.</td>
</tr>
<tr>
<td>RBW_ROLES</td>
<td>Describes all user-created roles defined in the database.</td>
</tr>
<tr>
<td>RBW_SEGMENTS</td>
<td>Describes all segments in the system; only users with the DBA or RESOURCE system role or ACCESS_ANY authorization can access this table.</td>
</tr>
<tr>
<td>RBW_STORAGE</td>
<td>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.</td>
</tr>
<tr>
<td>RBW_SYNONYMS</td>
<td>Describes all table synonyms for tables listed in RBW_TABLES.</td>
</tr>
<tr>
<td>RBW_TABAUTH</td>
<td>Describes the access rights granted on objects in RBW_TABLES.</td>
</tr>
<tr>
<td>RBW_TABLES</td>
<td>Describes all tables (including system tables), views, and synonyms in the database.</td>
</tr>
<tr>
<td>RBW_USERAUTH</td>
<td>Describes access rights granted to users authorized to use the database.</td>
</tr>
<tr>
<td>RBW_VIEWS</td>
<td>Describes the views in RBW_TABLES and provides information about precomputed views.</td>
</tr>
<tr>
<td>RBW_VIEWTEXT</td>
<td>Describes the text of all views in RBW_VIEWS.</td>
</tr>
</tbody>
</table>

1 Only returns rows if Enterprise Control and Coordination option is enabled with a license key.
The RBW_COLUMNS table describes the columns from all database objects listed in RBW_TABLES table. When a user displays information stored in the RBW_COLUMNS table, the user sees only those columns in objects that the user created or has permission to access. This table is updated by the ALTER, CREATE and DROP TABLE, VIEW, and SYNONYM commands. It contains the following columns:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>CHAR(128)</td>
<td>Name of column.</td>
</tr>
<tr>
<td>TNAME</td>
<td>CHAR(128)</td>
<td>Name of table.</td>
</tr>
<tr>
<td>SEQ</td>
<td>SMALLINT</td>
<td>Column sequence number.</td>
</tr>
<tr>
<td>TYPE</td>
<td>CHAR(12)</td>
<td>Column datatype.</td>
</tr>
<tr>
<td>LENGTH</td>
<td>SMALLINT</td>
<td>Actual length in bytes of column.</td>
</tr>
<tr>
<td>PRECISION</td>
<td>SMALLINT</td>
<td>Specified or implied numeric precision; for TIME and TIMESTAMP columns, the digits used to display fractional seconds.</td>
</tr>
<tr>
<td>SCALE</td>
<td>SMALLINT</td>
<td>Specified scaling factor.</td>
</tr>
<tr>
<td>NULLS</td>
<td>CHAR(1)</td>
<td>Flag indicating whether NULLs are allowed (Y or N).</td>
</tr>
<tr>
<td>UNIQ</td>
<td>CHAR(1)</td>
<td>Flag indicating whether column values are unique (Y or N).</td>
</tr>
<tr>
<td>PKSEQ</td>
<td>SMALLINT</td>
<td>Sequence of column in primary key; 0 if not a column in primary key.</td>
</tr>
<tr>
<td>TID</td>
<td>SMALLINT</td>
<td>Table identifier.</td>
</tr>
<tr>
<td>DEFAULTVALUE</td>
<td>CHAR(256)</td>
<td>Default for column.</td>
</tr>
</tbody>
</table>
The RBW_CONSTRAINTCOLUMNS table identifies the columns on which primary and foreign key constraints are defined in CREATE TABLE statements:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTRAINT_NAME</td>
<td>CHAR(128)</td>
<td>The name of the constraint.</td>
</tr>
<tr>
<td>TNAME</td>
<td>CHAR(128)</td>
<td>The name of the table containing the constraint.</td>
</tr>
<tr>
<td>CNAME</td>
<td>CHAR(128)</td>
<td>The name of the column on which the constraint is defined.</td>
</tr>
<tr>
<td>COLSEQ</td>
<td>INTEGER</td>
<td>The sequence of the column in the constraint definition.</td>
</tr>
</tbody>
</table>
**RBW_CONSTRAINTS Table**

The RBW_CONSTRAINTS table describes the primary and foreign key constraints defined in CREATE TABLE statements:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>CHAR(128)</td>
<td>The name of the constraint.</td>
</tr>
<tr>
<td>ID</td>
<td>INTEGER</td>
<td>The internal identification number of the constraint.</td>
</tr>
<tr>
<td>TYPE</td>
<td>CHAR(11)</td>
<td>The type of constraint: PRIMARY KEY or FOREIGN KEY.</td>
</tr>
<tr>
<td>TNAME</td>
<td>CHAR(128)</td>
<td>The name of the table containing the constraint.</td>
</tr>
<tr>
<td>CREATOR</td>
<td>CHAR(128)</td>
<td>The user who created or last altered the table definition.</td>
</tr>
</tbody>
</table>

**RBW_HIERARCHIES Table**

The RBW_HIERARCHIES shows the table and column relationships within a hierarchy:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>CHAR(128)</td>
<td>Name of the hierarchy.</td>
</tr>
<tr>
<td>FROM_TABLE</td>
<td>CHAR(128)</td>
<td>Table from which values are mapped.</td>
</tr>
<tr>
<td>FROM_COLUMN</td>
<td>CHAR(128)</td>
<td>Column from which values are mapped.</td>
</tr>
<tr>
<td>TO_TABLE</td>
<td>CHAR(128)</td>
<td>Table to which values are mapped.</td>
</tr>
<tr>
<td>TO_COLUMN</td>
<td>CHAR(128)</td>
<td>Column to which values are mapped.</td>
</tr>
<tr>
<td>CONSTRAINT_NAME</td>
<td>CHAR(128)</td>
<td>Names the foreign key constraint through which a rollup relationship is defined. Indicates NULL if the rollup relationship is within the same table.</td>
</tr>
</tbody>
</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 commands and contains the following columns:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INAME</td>
<td>CHAR(128)</td>
<td>Name of index.</td>
</tr>
<tr>
<td>TNAME</td>
<td>CHAR(128)</td>
<td>Name of table.</td>
</tr>
<tr>
<td>CNAME</td>
<td>CHAR(128)</td>
<td>Name of column in key.</td>
</tr>
<tr>
<td>SEQ</td>
<td>SMALLINT</td>
<td>Sequence number of column in key.</td>
</tr>
<tr>
<td>FKNAME</td>
<td>CHAR(128)</td>
<td>Name of the foreign key constraint for STAR indexes. Returns NULL for non-STAR indexes.</td>
</tr>
</tbody>
</table>

**RBW_INDEXES Table**

The RBW_INDEXES table describes the indexes on all objects listed in 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 commands and contains the following columns:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>CHAR(128)</td>
<td>Name of index; default primary key indexes are named as: <code>&lt;table_name&gt;_PK_IDX</code>.</td>
</tr>
<tr>
<td>TNAME</td>
<td>CHAR(128)</td>
<td>Name of table.</td>
</tr>
<tr>
<td>TYPE</td>
<td>CHAR(7)</td>
<td>Index type: BTREE, STAR, TARGET, TARGETS, TARGETM, or TARGETL.</td>
</tr>
<tr>
<td>CNAME</td>
<td>CHAR(128)</td>
<td>Indexed column name; first column name on multi-column indexes.</td>
</tr>
</tbody>
</table>
**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 is part of the Enterprise Control and Coordination option, which must be enabled with a license key. It contains the following columns:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TNAME</td>
<td>CHAR(128)</td>
<td>Name of table.</td>
</tr>
<tr>
<td>SEGNAME</td>
<td>CHAR(128)</td>
<td>Name of segment if load performed into an offline segment; NULL if not an offline load.</td>
</tr>
<tr>
<td>USERNAME</td>
<td>CHAR(128)</td>
<td>Name of user that performed load.</td>
</tr>
<tr>
<td>STARTED</td>
<td>TIMESTAMP</td>
<td>Date and time load started.</td>
</tr>
<tr>
<td>FINISHED</td>
<td>TIMESTAMP</td>
<td>Date and time load completed.</td>
</tr>
</tbody>
</table>
### 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 user 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 commands and contains the following columns:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>CHAR(128)</td>
<td>Name of macro.</td>
</tr>
<tr>
<td>TYPE</td>
<td>CHAR(9)</td>
<td>Macro type: PUBLIC, PRIVATE, or TEMPORARY.</td>
</tr>
<tr>
<td>NARGS</td>
<td>SMALLINT</td>
<td>Number of macro arguments expected.</td>
</tr>
<tr>
<td>CREATOR</td>
<td>CHAR(128)</td>
<td>Creator of macro.</td>
</tr>
<tr>
<td>DATETIME</td>
<td>TIMESTAMP</td>
<td>Date and time of macro creation.</td>
</tr>
</tbody>
</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:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USERNAME</td>
<td>CHAR(128)</td>
<td>Name of user running the current session.</td>
</tr>
<tr>
<td>OPTION_NAME</td>
<td>CHAR(128)</td>
<td>Name of parameter.</td>
</tr>
<tr>
<td>VALUE</td>
<td>CHAR(1024)</td>
<td>Current value of parameter.</td>
</tr>
</tbody>
</table>

RBW_PRECOMPVIEW_CANDIDATES Table

For a description of the RBW_PRECOMPVIEW_CANDIDATES Advisor system table, refer to the Red Brick Vista User's Guide.

RBW_PRECOMPVIEW_UTILIZATION Table

For a description of the RBW_PRECOMPVIEW_UTILIZATION Advisor system table, refer to the Red Brick Vista User's Guide.
The RBW_PRECOMPVIEWCOLMUNS table shows the relationship between columns in aggregate tables and precomputed views.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TNAME</td>
<td>CHAR(128)</td>
<td>Name of the aggregate table associated with the precomputed view.</td>
</tr>
<tr>
<td>TCOLUMN</td>
<td>CHAR(128)</td>
<td>Name of the column in the aggregate table.</td>
</tr>
<tr>
<td>VNAME</td>
<td>CHAR</td>
<td>Name of the precomputed view.</td>
</tr>
<tr>
<td>VCOLUMN</td>
<td>CHAR</td>
<td>Name of the column in the precomputed view.</td>
</tr>
</tbody>
</table>
**RBW_RELATIONSHIPS Table**

The RBW_RELATIONSHIPS table describes the primary key-foreign key relationships between tables in a schema. It contains the following columns:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PKTABLE</td>
<td>CHAR(128)</td>
<td>The name of the referenced (dimension) table.</td>
</tr>
<tr>
<td>FKTABLE</td>
<td>CHAR(128)</td>
<td>The name of the referencing (fact) table.</td>
</tr>
<tr>
<td>PKCONSTRAINT</td>
<td>CHAR(128)</td>
<td>The name of the primary key constraint in the referenced table. If the CREATE TABLE statement does not name the constraint, a name will be generated by appending the string _PKEYCONSTRAINT to the name of the table.</td>
</tr>
<tr>
<td>FKCONSTRAINT</td>
<td>CHAR(128)</td>
<td>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 will be generated by appending the string _FKKEYnCONSTRAINT 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.</td>
</tr>
<tr>
<td>DELACTION</td>
<td>CHAR(9)</td>
<td>Action triggered by DELETE: CASCADE, NO_ACTION.</td>
</tr>
<tr>
<td>CREATOR</td>
<td>CHAR(128)</td>
<td>The user who created the table.</td>
</tr>
</tbody>
</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 RBW_ROLE_MEMBERS table, the user sees all user-created roles that have members. This table is updated by GRANT and REVOKE commands.

This table is part of the Enterprise Control and Coordination option. In order to query RBW_ROLE_MEMBERS, this option must be enabled with a license key. It contains the following columns:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROLENAME</td>
<td>CHAR(128)</td>
<td>Name of role.</td>
</tr>
<tr>
<td>USERNAME</td>
<td>CHAR(128)</td>
<td>Name of user or user-created role that has been granted ROLENAME.</td>
</tr>
<tr>
<td>INDIRECT</td>
<td>CHAR(1)</td>
<td>Whether the user or user-created role is an indirect member of ROLENAME (Y if an indirect member; N if a direct member).</td>
</tr>
<tr>
<td>ADDED</td>
<td>TIMESTAMP</td>
<td>Date and time the user or user-created role became a member of ROLENAME.</td>
</tr>
</tbody>
</table>
**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.

This table is part of the Enterprise Control and Coordination option. In order to query RBW.Roles, this option must be enabled with a license key. It contains the following columns:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>CHAR(128)</td>
<td>Name of role.</td>
</tr>
<tr>
<td>CREATOR</td>
<td>CHAR(128)</td>
<td>Creator of role.</td>
</tr>
<tr>
<td>CREATED</td>
<td>TIMESTAMP</td>
<td>Date and time role was created.</td>
</tr>
<tr>
<td>COMMENT</td>
<td>CHAR(256)</td>
<td>User-specified comment; NULL if not set with the ALTER ROLE command or if the Enterprise Control and Coordination option key not found.</td>
</tr>
</tbody>
</table>
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 commands. It contains the following columns:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>CHAR(128)</td>
<td>Name of segment.</td>
</tr>
<tr>
<td>TNAME</td>
<td>CHAR(128)</td>
<td>Name of table that uses segment for row data or indexes; set to NULL for unattached segments.</td>
</tr>
<tr>
<td>CREATOR</td>
<td>CHAR(128)</td>
<td>Creator of segment.</td>
</tr>
<tr>
<td>DATETIME</td>
<td>TIMESTAMP</td>
<td>Date and time of segment creation.</td>
</tr>
<tr>
<td>NPSUS</td>
<td>INTEGER</td>
<td>Number of physical storage units (PSUs) used for segment.</td>
</tr>
<tr>
<td>NCOLS</td>
<td>INTEGER</td>
<td>Number of columns used to segment the data or index.</td>
</tr>
<tr>
<td>MINKEY</td>
<td>CHAR(256)</td>
<td>Minimum key value in the segment; displays first 256 characters.</td>
</tr>
<tr>
<td>MAXKEY</td>
<td>CHAR(256)</td>
<td>Maximum key value in the segment; displays first 256 characters.</td>
</tr>
<tr>
<td>ID</td>
<td>INTEGER</td>
<td>Segment ID.</td>
</tr>
<tr>
<td>TOTALFREE</td>
<td>INTEGER</td>
<td>Kilobytes of unused space in segment.</td>
</tr>
<tr>
<td>INAME</td>
<td>CHAR(128)</td>
<td>Name of index that uses segment; set to NULL for unattached and row data segments.</td>
</tr>
<tr>
<td>ONLINE</td>
<td>CHAR(1)</td>
<td>Whether a segment is online (Y or N). For System segment (NULL).</td>
</tr>
<tr>
<td>OPTICAL</td>
<td>CHAR(1)</td>
<td>Whether a segment contains one or more optical PSUs (Y or N). (NULL for system segment.)</td>
</tr>
<tr>
<td>Column Name</td>
<td>Column Type</td>
<td>Column Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>INTACT</td>
<td>CHAR(1)</td>
<td>Flag indicating whether segment is intact (Y), or there is detected, unrepaired damage (N). For System segment (NULL).</td>
</tr>
<tr>
<td>INSYNCH</td>
<td>CHAR(1)</td>
<td>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.</td>
</tr>
<tr>
<td>BACKUPLEVEL</td>
<td>INTEGER</td>
<td>Current segment backup level; all blocks within all files of this segment are backed up to at least this level (TMU Backup option only).</td>
</tr>
<tr>
<td>LAST_OFFLINE</td>
<td>TIMESTAMP</td>
<td>Date and time the segment last set offline; initially contains segment creation time.</td>
</tr>
<tr>
<td>LAST_ONLINE</td>
<td>TIMESTAMP</td>
<td>Date and time segment last set online; initially contains segment creation time.</td>
</tr>
<tr>
<td>LAST_LOAD</td>
<td>TIMESTAMP</td>
<td>Completion time of the last offline load into segment; initially set to NULL.</td>
</tr>
<tr>
<td>LAST_BACKUP</td>
<td>TIMESTAMP</td>
<td>Date and time segment was last backed up; initially set to NULL. (TMU Backup option only).</td>
</tr>
<tr>
<td>LAST_RESTORE</td>
<td>TIMESTAMP</td>
<td>Date and time of most recent restore of segment from backup; initially set to NULL (TMU Backup option only).</td>
</tr>
<tr>
<td>COMMENT</td>
<td>CHAR(256)</td>
<td>User-specified comment or descriptive data; NULL if not set with the ALTER SEGMENT command or if the Enterprise Control and Coordination option key not found.</td>
</tr>
<tr>
<td>LOCAL_ID</td>
<td>SMALLINT</td>
<td>The segment ID that is returned in the RBW_SEGID pseudo-column.</td>
</tr>
</tbody>
</table>
RBW_STORAGE Table

The RBW_STORAGE table describes the physical storage units (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 commands and contains the following columns:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEGNAME</td>
<td>CHAR(128)</td>
<td>Segment name containing PSU.</td>
</tr>
<tr>
<td>SEGID</td>
<td>SMALLINT</td>
<td>Segment ID containing PSU.</td>
</tr>
<tr>
<td>PSEQ</td>
<td>INTEGER</td>
<td>Sequence number of PSU in segment.</td>
</tr>
<tr>
<td>LOCATION</td>
<td>CHAR(1024)</td>
<td>The location of PSU.</td>
</tr>
<tr>
<td>MAXSIZE</td>
<td>INTEGER</td>
<td>Kilobytes of maximum allowed size of PSU.</td>
</tr>
<tr>
<td>INITSIZE</td>
<td>INTEGER</td>
<td>Kilobytes of initial allocated size of PSU.</td>
</tr>
<tr>
<td>EXTENDSIZE</td>
<td>INTEGER</td>
<td>Kilobytes of increment size to use when extending PSU.</td>
</tr>
<tr>
<td>USED</td>
<td>INTEGER</td>
<td>Kilobytes of current area in use in PSU.</td>
</tr>
<tr>
<td>INTACT</td>
<td>CHAR(1)</td>
<td>Flag indicating whether PSU is intact (Y), or there is detected, unrepaired damage (N). For System segment (NULL).</td>
</tr>
<tr>
<td>BACKUPLEVEL</td>
<td>SMALLINT</td>
<td>Backup level of PSU; all blocks within this PSU backed up to at least this level (TMU Backup option only).</td>
</tr>
</tbody>
</table>
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 commands and contains the following columns:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>CHAR(128)</td>
<td>Name of synonym.</td>
</tr>
<tr>
<td>TNAME</td>
<td>CHAR(128)</td>
<td>Name of table referenced by synonym.</td>
</tr>
<tr>
<td>CREATOR</td>
<td>CHAR(128)</td>
<td>Creator of synonym.</td>
</tr>
<tr>
<td>COMMENT</td>
<td>CHAR(256)</td>
<td>User-specified comment; NULL if not set with the ALTER SYNONYM command or if the Enterprise Control and Coordination option key not found.</td>
</tr>
</tbody>
</table>
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 commands and contains the following columns:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANTEE</td>
<td>CHAR(128)</td>
<td>User or role(^1) granted object privilege to a table or view.</td>
</tr>
<tr>
<td>GRANTOR</td>
<td>CHAR(128)</td>
<td>User granting privilege to GRANTEE.</td>
</tr>
<tr>
<td>TNAME</td>
<td>CHAR(128)</td>
<td>Name of table, view, or synonym.</td>
</tr>
<tr>
<td>SELAUTH</td>
<td>CHAR(1)</td>
<td>Whether grantee has SELECT privilege (Y, N, R, or I)(^2).</td>
</tr>
<tr>
<td>INSAUTH</td>
<td>CHAR(1)</td>
<td>Whether grantee has INSERT privilege (Y, N, R, or I)(^2).</td>
</tr>
<tr>
<td>DELAUTH</td>
<td>CHAR(1)</td>
<td>Whether grantee has DELETE privilege (Y, N, R, or I)(^2).</td>
</tr>
<tr>
<td>UPDAUTH</td>
<td>CHAR(1)</td>
<td>Whether grantee has UPDATE privilege (Y, N, R, or I)(^2).</td>
</tr>
<tr>
<td>DATETIME</td>
<td>TIMESTAMP</td>
<td>Date and time object privilege granted.</td>
</tr>
</tbody>
</table>

\(^1\)Object privileges can be granted to user-created roles.
(Enterprise Control and Coordination option only.)

\(^2\)Y—User has the object privilege.
N—User does not have the object privilege.
R—User has the object privilege directly through a role.
(Enterprise Control and Coordination option only.)
I—User has the object privilege indirectly through a role.
(Enterprise Control and Coordination option only.)
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 commands and contains the following columns:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>CHAR(128)</td>
<td>Name of table, view, or synonym.</td>
</tr>
<tr>
<td>TYPE</td>
<td>CHAR(8)</td>
<td>Type of object: TABLE, VIEW, SYNONYM, or SYSTEM.</td>
</tr>
<tr>
<td>CREATOR</td>
<td>CHAR(128)</td>
<td>Creator of table (blank for system tables).</td>
</tr>
<tr>
<td>ID</td>
<td>SMALLINT</td>
<td>Table identifier; negative numbers identify system tables.</td>
</tr>
<tr>
<td>DATETIME</td>
<td>TIMESTAMP</td>
<td>Date and time of table creation.</td>
</tr>
<tr>
<td>MAXSEGMENTS</td>
<td>INTEGER</td>
<td>Maximum number of segments allowed for the table (specified with CREATE TABLE or ALTER TABLE); indicates NULL if not specified.</td>
</tr>
<tr>
<td>MAXROWS_PER_SEG</td>
<td>INTEGER</td>
<td>Maximum number of rows allowed per segment (specified with CREATE TABLE or ALTER TABLE); indicates NULL if not specified.</td>
</tr>
<tr>
<td>MAXSIZE_ROWS</td>
<td>INTEGER</td>
<td>Maximum number of rows (calculated from MAXSIZE parameter of CREATE SEGMENT for all PSUs in each segment attached).</td>
</tr>
<tr>
<td>SEGMENT_BY</td>
<td>CHAR(11)</td>
<td>Segmentation scheme of table (range, hash, NULL).</td>
</tr>
<tr>
<td>INTACT</td>
<td>CHAR(1)</td>
<td>Flag indicating whether table is intact (Y), or there is detected, unrepaired damage (N).</td>
</tr>
</tbody>
</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 all users’ authorizations. 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 commands and contains the following columns:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARTIAL</td>
<td>CHAR(1)</td>
<td>Flag indicating whether table is partially available due to one or more offline segments (Y or N).</td>
</tr>
<tr>
<td>COMMENT</td>
<td>CHAR(256)</td>
<td>User-specified comment; NULL if not set or if the Enterprise Control and Coordination option key not found.</td>
</tr>
</tbody>
</table>

**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 all users’ authorizations. 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 commands and contains the following columns:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANTEE</td>
<td>CHAR(128)</td>
<td>User or role(^1) granted authorization.</td>
</tr>
<tr>
<td>GRANTOR</td>
<td>CHAR(128)</td>
<td>User granting authorization to GRANTEE.</td>
</tr>
<tr>
<td>DBAAUTH</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE is a member of DBA system role (Y, N, R, or I(^2)).</td>
</tr>
<tr>
<td>RESAUTH</td>
<td>CHAR(1)</td>
<td>Whether grantee is a member of RESOURCE system role (Y, N, R, or I(^2)).</td>
</tr>
<tr>
<td>DATETIME</td>
<td>TIMESTAMP</td>
<td>Date and time authorization granted.</td>
</tr>
<tr>
<td>LOCKED(^1)</td>
<td>CHAR(1)</td>
<td>Whether or not GRANTEE’s account is locked due to failed connection attempts (Y or N); NULL for roles.</td>
</tr>
<tr>
<td>Column Name</td>
<td>Column Type</td>
<td>Column Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>EXPIRED¹</td>
<td>CHAR(1)</td>
<td>Whether or not GRANTEE’s account is expired because of failure to change password before the defined expiration date (Y or N); NULL for roles.</td>
</tr>
<tr>
<td>PASSWORD_TS</td>
<td>TIMESTAMP</td>
<td>Date and time GRANTEE’s database password was added or last changed.</td>
</tr>
<tr>
<td>USER_MANAGEMENT</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE can alter, create and drop database users and change passwords (Y, N, R, or I)².</td>
</tr>
<tr>
<td>GRANT_TABLE</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE can grant object privileges to database users and to roles (Y, N, R, or I)².</td>
</tr>
<tr>
<td>ROLE_MANAGEMENT</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE can alter, create, drop, grant, and revoke roles (Y, N, R, or I)².</td>
</tr>
<tr>
<td>ALTER_ANY</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE can alter indexes, segments, tables, macros, views, and synonyms (Y, N, R, or I)².</td>
</tr>
<tr>
<td>PUBLIC_MACROS</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE can create and drop PUBLIC macros (Y, N, R, or I)².</td>
</tr>
<tr>
<td>ACCESS_ANY</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE can select data from all database objects, including private user information in the system tables (Y, N, R, or I)².</td>
</tr>
<tr>
<td>MODIFY_ANY</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE can insert, update, delete, and load any data (Y, N, R, or I)².</td>
</tr>
<tr>
<td>DROP_ANY</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE can drop objects created by any user (Y, N, R, or I)².</td>
</tr>
<tr>
<td>Column Name</td>
<td>Column Type</td>
<td>Column Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CREATE_ANY</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE can create any object, including those that use another’s resources (Y, N, R, or I)².</td>
</tr>
<tr>
<td>LOCK_DATABASE</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE can lock the database (Y, N, R, or I)².</td>
</tr>
<tr>
<td>BACKUP_DATABASE</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE can backup the database (Y, N, R, or I)².</td>
</tr>
<tr>
<td>RESTORE_DATABASE</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE can restore the database (Y, N, R, or I)².</td>
</tr>
<tr>
<td>UPGRADE_DATABASE</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE can upgrade the database (Y, N, R, or I)².</td>
</tr>
<tr>
<td>REORG_ANY</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE can re-organize any table or index (Y, N, R, or I)².</td>
</tr>
<tr>
<td>OFFLINE_LOAD</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE can use any segment as a working segment for offline loads or synchronize segments after offline loads (Y, N, R, or I)².</td>
</tr>
<tr>
<td>ALTER_SYSTEM</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE can issue the ALTER SYSTEM command to perform database administration tasks (Y, N, R, or I)².</td>
</tr>
<tr>
<td>ACCESS_SYSINFO</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE can query the dynamic statistic tables for statistics about database activity (Y, N, R, or I)².</td>
</tr>
<tr>
<td>ALTER_TABLE_INTO_ANY</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE can alter own tables into other users’ segments (Y, N, R, or I)².</td>
</tr>
<tr>
<td>CREATE_OWN</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE can create own objects: indexes, private macros, segments, synonyms, tables, and views (Y, N, R, or I)².</td>
</tr>
</tbody>
</table>
## System Tables and Dynamic Statistic Tables
### System Catalog

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DROP_OWN</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE can drop own objects (Y, N, R, or I)².</td>
</tr>
<tr>
<td>ALTER_OWN</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE can alter own indexes, macros, segments, synonyms, tables and views</td>
</tr>
<tr>
<td>GRANT_OWN</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE can grant object privileges on own objects to other users (Y, N, R, or I)².</td>
</tr>
<tr>
<td>IGNORE_QUIESCE</td>
<td>CHAR(1)</td>
<td>Flag indicating whether a user can access a quiesced database (Y) or is locked out (N).</td>
</tr>
<tr>
<td>ISROLE</td>
<td>CHAR(1)</td>
<td>Whether GRANTEE is a role (Y if a role; N if a user).</td>
</tr>
<tr>
<td>PRIORITY</td>
<td>SMALLINT</td>
<td>Value between 0–100 indicating user priority for warehouse resources. 0 is high priority; 100, low.</td>
</tr>
<tr>
<td>COMMENT</td>
<td>CHAR(256)</td>
<td>User-specified comment; NULL if not set or if the Enterprise Control and Coordination option key not found.</td>
</tr>
</tbody>
</table>

¹Task authorizations can be granted to user-created roles. (Enterprise Control and Coordination option only.)

²Y—User has the task authorization.
N—User does not have the task authorization.
R—User has the task authorization directly through a role. (Enterprise Control and Coordination option only.)
I—User has the task authorization indirectly through a role. (Enterprise Control and Coordination option only.)
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 commands and contains the following columns:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>CHAR(128)</td>
<td>Name of view.</td>
</tr>
<tr>
<td>CREATOR</td>
<td>CHAR(128)</td>
<td>Creator of view.</td>
</tr>
<tr>
<td>PRECOMPVIEW</td>
<td>CHAR(1)</td>
<td>Denotes whether the view is precomputed (Y or N).</td>
</tr>
<tr>
<td>PRECOMPVIEW_TABLE</td>
<td>CHAR(128)</td>
<td>Name of the aggregate table associated with the precomputed view. Indicates NULL if the view is not a precomputed view.</td>
</tr>
<tr>
<td>DETAIL_TABLE</td>
<td>CHAR(128)</td>
<td>Denotes the detail table on which the aggregate table is defined. Indicates NULL if the view is not a precomputed view.</td>
</tr>
<tr>
<td>VALID</td>
<td>CHAR(1)</td>
<td>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.</td>
</tr>
<tr>
<td>COMMENT</td>
<td>CHAR(256)</td>
<td>User-specified comment; NULL if not set with the ALTER VIEW command or if the Enterprise Control and Coordination option key not found.</td>
</tr>
</tbody>
</table>
**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 commands and contains the following columns:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Column Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>CHAR(128)</td>
<td>Name of view.</td>
</tr>
<tr>
<td>SEQ</td>
<td>INTEGER</td>
<td>Sequence number of the view text.</td>
</tr>
<tr>
<td>TEXT</td>
<td>CHAR(1024)</td>
<td>Text of view definition (including CREATE VIEW keywords).</td>
</tr>
</tbody>
</table>
Dynamic Statistic Tables

The Dynamic Statistic Tables (DSTs) are used with the Enterprise Control and Coordination Option to help monitor database activity. The DSTs consist of the following tables:

- DST_COMMANDS
- DST_DATABASES
- DST_LOCKS
- DST_SESSIONS
- DST_USERS

Refer to Chapter 7, “Managing Database Activity with the Enterprise Control and Coordination Option,” for more information on the DSTs.

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:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBNAME</td>
<td>CHAR(128)</td>
<td>Logical database name.</td>
</tr>
<tr>
<td>UNAME</td>
<td>CHAR(128)</td>
<td>User name for the user issuing the command.</td>
</tr>
<tr>
<td>NODE_NAME</td>
<td>CHAR(128)</td>
<td>Name of node on which command is running (for MPP servers).</td>
</tr>
<tr>
<td>PID</td>
<td>INTEGER</td>
<td>Process ID for session running the command.</td>
</tr>
<tr>
<td>STARTED</td>
<td>TIMESTAMP</td>
<td>Start time of the command.</td>
</tr>
<tr>
<td>STATE</td>
<td>CHAR(64)</td>
<td>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)</td>
</tr>
<tr>
<td>Column Name</td>
<td>Column Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>COMMAND</td>
<td>CHAR(1024)</td>
<td>Starting text of current command prior to macro expansion.</td>
</tr>
<tr>
<td>CACHE_READS</td>
<td>INTEGER</td>
<td>Number of times that a block was found in local buffer cache (avoiding a logical read request).</td>
</tr>
<tr>
<td>CACHE_WRITES</td>
<td>INTEGER</td>
<td>Number of times that a block was found in local buffer cache (avoiding a logical write request).</td>
</tr>
<tr>
<td>LOGICAL_READS</td>
<td>INTEGER</td>
<td>Number of logical reads performed by the command.</td>
</tr>
<tr>
<td>LOGICAL_WRITES</td>
<td>INTEGER</td>
<td>Number of logical writes performed by the command.</td>
</tr>
<tr>
<td>PHYSICAL_READS</td>
<td>INTEGER</td>
<td>Number of physical reads performed by the command (NULL for platforms that do not support this statistic).</td>
</tr>
<tr>
<td>PHYSICAL_WRITES</td>
<td>INTEGER</td>
<td>Number of physical writes performed by the command (NULL for platforms that do not support this statistic).</td>
</tr>
<tr>
<td>SYSTEM_CPUTIME</td>
<td>DEC(9,2)</td>
<td>System CPU time used by the command (in seconds).</td>
</tr>
<tr>
<td>USER_CPUTIME</td>
<td>DEC(9,2)</td>
<td>User CPU time used by the command (in seconds).</td>
</tr>
<tr>
<td>SPILL_COUNT</td>
<td>INTEGER</td>
<td>The number of spill files used in an operation</td>
</tr>
<tr>
<td>MEMORY_USED</td>
<td>INTEGER</td>
<td>Amount of memory being used by the command (in kilobytes).</td>
</tr>
<tr>
<td>TEMPSPACE_USED</td>
<td>INTEGER</td>
<td>Amount of spill space used by the command (in kilobytes).</td>
</tr>
<tr>
<td>PARALLELISM</td>
<td>INTEGER</td>
<td>Number of parallel tasks being performed by the command.</td>
</tr>
<tr>
<td>LAST_UPDATED</td>
<td>TIMESTAMP</td>
<td>Timestamp of when this row was last updated.</td>
</tr>
</tbody>
</table>
**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:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBNAME</td>
<td>CHAR(128)</td>
<td>Logical database name.</td>
</tr>
<tr>
<td>DBLOCATION</td>
<td>CHAR(1024)</td>
<td>Database directory path.</td>
</tr>
<tr>
<td>CURRENT_CONNECTS</td>
<td>INTEGER</td>
<td>Current number of connected sessions.</td>
</tr>
<tr>
<td>PEAK_CONNECTS</td>
<td>INTEGER</td>
<td>Maximum number of concurrent sessions.</td>
</tr>
<tr>
<td>TOTAL_CONNECTS</td>
<td>INTEGER</td>
<td>Cumulative count of connected sessions.</td>
</tr>
<tr>
<td>TOTAL_FATAL_EXITS</td>
<td>INTEGER</td>
<td>Number of times a session has terminated abnormally. This includes any invalid login attempts and failed attempts against quiesced databases.</td>
</tr>
<tr>
<td>TOTAL_COMMANDS</td>
<td>INTEGER</td>
<td>Number of commands executed against this database.</td>
</tr>
<tr>
<td>QUIESCED</td>
<td>CHAR(1)</td>
<td>Flag indicating whether the database is quiesced (Y) or active (N).</td>
</tr>
<tr>
<td>AADMINDB</td>
<td>CHAR(1)</td>
<td>Flag indicating whether the database is the administration database (Y) or a user-created database (N).</td>
</tr>
<tr>
<td>LAST_UPDATED</td>
<td>TIMESTAMP</td>
<td>Timestamp of when this row was last updated.</td>
</tr>
<tr>
<td>BACKUP_SEGMENT</td>
<td>CHAR(128)</td>
<td>Name of the backup segment (populated only if a backup segment exists for use with SQL-BackTrack for Red Brick Warehouse).</td>
</tr>
</tbody>
</table>
**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:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBNAME</td>
<td>CHAR(128)</td>
<td>Database logical name.</td>
</tr>
<tr>
<td>UNAME</td>
<td>CHAR(128)</td>
<td>User name.</td>
</tr>
<tr>
<td>NODE_NAME</td>
<td>CHAR(128)</td>
<td>Node name where the process is running (for MPP systems).</td>
</tr>
<tr>
<td>PID</td>
<td>INTEGER</td>
<td>PID of this process.</td>
</tr>
<tr>
<td>TNAME</td>
<td>CHAR(128)</td>
<td>Table being locked (NULL for segment lock).</td>
</tr>
<tr>
<td>SEGNAME</td>
<td>CHAR(128)</td>
<td>Name of the segment for a segment lock (NULL for table only lock).</td>
</tr>
<tr>
<td>DATETIME</td>
<td>TIMESTAMP</td>
<td>Time of the lock request.</td>
</tr>
<tr>
<td>TYPE</td>
<td>CHAR(1)</td>
<td>Flag indicating whether the lock type is read (R) or write (W).</td>
</tr>
<tr>
<td>BLOCKER_UNAME</td>
<td>CHAR(128)</td>
<td>Name of user holding lock or NULL if current process is holding the lock.</td>
</tr>
<tr>
<td>BLOCKER_PID</td>
<td>INTEGER</td>
<td>PID of process holding lock that is blocking current attempt or NULL if the current process holds the lock.</td>
</tr>
<tr>
<td>BLOCKER_NODE</td>
<td>CHAR(128)</td>
<td>Node name where the blocking process is running (for MPP systems) or NULL if the current process is holding the lock.</td>
</tr>
<tr>
<td>LAST_UPDATED</td>
<td>TIMESTAMP</td>
<td>Timestamp of when this row was last updated.</td>
</tr>
</tbody>
</table>
**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).

The following table lists and describes the DST_SESSIONS columns:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBNAME</td>
<td>CHAR(128)</td>
<td>Logical database name.</td>
</tr>
<tr>
<td>UNAME</td>
<td>CHAR(128)</td>
<td>Database user name for user running the session.</td>
</tr>
<tr>
<td>NODE_NAME</td>
<td>CHAR(128)</td>
<td>Name of node on which session is running (for MPP servers).</td>
</tr>
<tr>
<td>PID</td>
<td>INTEGER</td>
<td>The rbwsvr process ID for the session.</td>
</tr>
<tr>
<td>COMPONENT</td>
<td>CHAR(32)</td>
<td>SERVER, WORKGROUP SERVER, TMU, or PTMU</td>
</tr>
<tr>
<td>STARTED</td>
<td>TIMESTAMP</td>
<td>Start time of the session.</td>
</tr>
<tr>
<td>NET_ADDRESS</td>
<td>CHAR(32)</td>
<td>Network address of client process if any.</td>
</tr>
<tr>
<td>CLIENT_TOOL</td>
<td>CHAR(128)</td>
<td>Name of client front-end tool or NULL if no client tool used.</td>
</tr>
<tr>
<td>GATEWAY</td>
<td>CHAR(128)</td>
<td>Gateway identifier or NULL if no gateway used.</td>
</tr>
<tr>
<td>PRIORITY</td>
<td>SMALLINT</td>
<td>Current priority of this session.</td>
</tr>
<tr>
<td>QUERY_MEMORY_LIMIT</td>
<td>INTEGER</td>
<td>Size at which queries are written to disk (in 8-kilobyte blocks).</td>
</tr>
<tr>
<td>QUERY_TEMPSPACE_</td>
<td>CHAR(1024)</td>
<td>Directories for query-related temporary space.</td>
</tr>
<tr>
<td>DIRECTORIES</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### System Tables and Dynamic Statistic Tables

#### Dynamic Statistic Tables

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUERY_TEMPSPACE_MAXSPILLSIZE</td>
<td>INTEGER</td>
<td>Maximum amount of temporary space per query (in 8-kilobyte blocks).</td>
</tr>
<tr>
<td>INDEX_TEMPSPACE_DIRECTORIES</td>
<td>CHAR(1024)</td>
<td>Directories for index-building temporary space.</td>
</tr>
<tr>
<td>INDEX_TEMPSPACE_THRESHOLD</td>
<td>INTEGER</td>
<td>Size at which index-related files are written to disk (in 8-kilobyte blocks).</td>
</tr>
<tr>
<td>INDEX_TEMPSPACE_MAXSPILLSIZE</td>
<td>INTEGER</td>
<td>Maximum amount of temporary space per index-building operation (in 8-kilobyte blocks).</td>
</tr>
<tr>
<td>REPORT_INTERVAL</td>
<td>INTEGER</td>
<td>Current session reporting interval (in minutes).</td>
</tr>
<tr>
<td>TOTAL_COMMANDS</td>
<td>INTEGER</td>
<td>Number of statements executed during this session.</td>
</tr>
<tr>
<td>TOTAL_CANCELS</td>
<td>INTEGER</td>
<td>Number of statements canceled during this session.</td>
</tr>
<tr>
<td>TOTAL_CACHE_READS</td>
<td>INTEGER</td>
<td>Number of times that a block was found in local buffer cache (avoiding a logical read request).</td>
</tr>
<tr>
<td>TOTAL_CACHE_WRITES</td>
<td>INTEGER</td>
<td>Number of times that a block was found in local buffer cache (avoiding a logical write request).</td>
</tr>
<tr>
<td>TOTAL_LOGICAL_READS</td>
<td>INTEGER</td>
<td>Number of logical reads performed by this session.</td>
</tr>
<tr>
<td>TOTAL_LOGICAL_WRITES</td>
<td>INTEGER</td>
<td>Number of logical writes performed by this session.</td>
</tr>
<tr>
<td>TOTAL_PHYSICAL_READS</td>
<td>INTEGER</td>
<td>Number of physical reads performed by this session (NULL for platforms that do not support this statistic).</td>
</tr>
</tbody>
</table>
### Dynamic Statistic Tables

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL_PHYSICAL_WRITES</td>
<td>INTEGER</td>
<td>Number of physical writes performed by this session (NULL for platforms that do not support this statistic).</td>
</tr>
<tr>
<td>TOTAL_SYSTEM_CPUTIME</td>
<td>DEC(9,2)</td>
<td>System CPU time used by this session (in seconds).</td>
</tr>
<tr>
<td>TOTAL_USER_CPUTIME</td>
<td>DEC(9,2)</td>
<td>User CPU time used by this session (in seconds).</td>
</tr>
<tr>
<td>TOTAL_SPILL_COUNT</td>
<td>INTEGER</td>
<td>Number of times a spill area was used by this session.</td>
</tr>
<tr>
<td>PEAK_CACHE_READS</td>
<td>INTEGER</td>
<td>Maximum number of times that a block was found in local buffer cache for a single session command (avoiding logical read requests).</td>
</tr>
<tr>
<td>PEAK_CACHE_WRITES</td>
<td>INTEGER</td>
<td>Maximum number of times that a block was found in local buffer cache for a single session command (avoiding logical write requests).</td>
</tr>
<tr>
<td>PEAK_LOGICAL_READS</td>
<td>INTEGER</td>
<td>Maximum number of logical reads performed by a command within the current session.</td>
</tr>
<tr>
<td>PEAK_LOGICAL_WRITES</td>
<td>INTEGER</td>
<td>Maximum number of logical writes performed by a command within the current session.</td>
</tr>
<tr>
<td>PEAK_PHYSICAL_READS</td>
<td>INTEGER</td>
<td>Maximum number of physical reads performed by a command within the current session (NULL for platforms that do not support this statistic).</td>
</tr>
<tr>
<td>Column Name</td>
<td>Column Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PEAK_PHYSICAL_WRITES</td>
<td>INTEGER</td>
<td>Maximum number of physical writes performed by a command within the current session (NULL for platforms that do not support this statistic).</td>
</tr>
<tr>
<td>PEAK_SYSTEM_CPUTIME</td>
<td>DEC(9,2)</td>
<td>Maximum system CPU time used by a command within the session (in seconds).</td>
</tr>
<tr>
<td>PEAK_USER_CPUTIME</td>
<td>DEC(9,2)</td>
<td>Maximum user CPU time used by a command within the session (in seconds).</td>
</tr>
<tr>
<td>PEAK_SPILL_COUNT</td>
<td>INTEGER</td>
<td>Maximum number of times a spill area was used by a command within the session.</td>
</tr>
<tr>
<td>PEAK_PARALLELISM</td>
<td>INTEGER</td>
<td>Maximum number of parallel tasks performed by a command within the session.</td>
</tr>
<tr>
<td>PEAK_MEMORY_USED</td>
<td>INTEGER</td>
<td>Maximum memory used by a single session command to access this database (in kilobytes).</td>
</tr>
<tr>
<td>PEAK_TEMPSPACE_USED</td>
<td>INTEGER</td>
<td>Maximum amount of spill space used by a single session command to access this database (in kilobytes).</td>
</tr>
<tr>
<td>LAST_UPDATED</td>
<td>TIMESTAMP</td>
<td>Timestamp of when this row was last updated.</td>
</tr>
</tbody>
</table>
**DST_USERS Table**

The DST_USERS table contains information about each user who has accessed the database since the statistics were reset. This information includes both *cumulative* statistics over all of the user’s sessions on the database and *peak* statistics—that is, maximum single-session values.

The following table lists and describes the DST_USERS columns:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBNAME</td>
<td>CHAR(128)</td>
<td>Logical database name.</td>
</tr>
<tr>
<td>UNAME</td>
<td>CHAR(128)</td>
<td>User name.</td>
</tr>
<tr>
<td>FIRST_LOGIN</td>
<td>TIMESTAMP</td>
<td>Time of user’s first access of this database.</td>
</tr>
<tr>
<td>LAST_LOGIN</td>
<td>TIMESTAMP</td>
<td>Time of user’s most recent access of this database.</td>
</tr>
<tr>
<td>CURRENT_CONNECTS</td>
<td>INTEGER</td>
<td>Number of currently connected sessions.</td>
</tr>
<tr>
<td>PEAK_CONNECTS</td>
<td>INTEGER</td>
<td>Maximum number of concurrent sessions at any time by this user.</td>
</tr>
<tr>
<td>TOTAL_CONNECTS</td>
<td>INTEGER</td>
<td>Total number of connects by this user.</td>
</tr>
<tr>
<td>TOTAL_FATAL_EXITS</td>
<td>INTEGER</td>
<td>Number of times this user has had a session terminate abnormally.</td>
</tr>
<tr>
<td>TOTAL_COMMANDS</td>
<td>INTEGER</td>
<td>Total number of commands executed by this user.</td>
</tr>
<tr>
<td>TOTAL_CANCELS</td>
<td>INTEGER</td>
<td>Number of canceled commands by this user.</td>
</tr>
<tr>
<td>TOTAL_CACHE_READS</td>
<td>INTEGER</td>
<td>Number of times that a block was found in local buffer cache (avoiding a logical read request).</td>
</tr>
<tr>
<td>TOTAL_CACHE_WRITES</td>
<td>INTEGER</td>
<td>Number of times that a block was found in local buffer cache (avoiding a logical write request).</td>
</tr>
<tr>
<td>Column Name</td>
<td>Column Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TOTAL_LOGICAL_READS</td>
<td>INTEGER</td>
<td>Number of logical reads performed by this user on this database.</td>
</tr>
<tr>
<td>TOTAL_LOGICAL_WRITES</td>
<td>INTEGER</td>
<td>Number of logical writes performed by this user on this database.</td>
</tr>
<tr>
<td>TOTAL_PHYSICAL_READS</td>
<td>INTEGER</td>
<td>Number of physical reads performed by this user on this database (NULL for platforms that do not support this statistic).</td>
</tr>
<tr>
<td>TOTAL_PHYSICAL_WRITES</td>
<td>INTEGER</td>
<td>Number of physical writes performed by this user on this database (NULL for platforms that do not support this statistic).</td>
</tr>
<tr>
<td>TOTAL_SYSTEM_CPUTIME</td>
<td>DEC(9,2)</td>
<td>Cumulative system CPU time used by this user to access this database (in seconds).</td>
</tr>
<tr>
<td>TOTAL_USER_CPUTIME</td>
<td>DEC(9,2)</td>
<td>Cumulative user CPU time used by this user to access this database (in seconds).</td>
</tr>
<tr>
<td>TOTAL_SPILL_COUNT</td>
<td>INTEGER</td>
<td>Cumulative count of the number of times that a spill area was used.</td>
</tr>
<tr>
<td>PEAK_CACHE_READS</td>
<td>INTEGER</td>
<td>Maximum number of times that a block was found in the local buffer cache for a single session (avoiding logical read requests).</td>
</tr>
<tr>
<td>PEAK_CACHE_WRITES</td>
<td>INTEGER</td>
<td>Maximum number of times that a block was found in the local buffer cache for a single session (avoiding logical write requests).</td>
</tr>
<tr>
<td>Column Name</td>
<td>Column Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PEAK_LOGICAL_READS</td>
<td>INTEGER</td>
<td>Maximum number of logical reads performed by this user on this database in one session.</td>
</tr>
<tr>
<td>PEAK_LOGICAL_WRITES</td>
<td>INTEGER</td>
<td>Maximum number of logical writes performed by this user on this database in one session.</td>
</tr>
<tr>
<td>PEAK_PHYSICAL_READS</td>
<td>INTEGER</td>
<td>Maximum number of physical reads performed by this user on this database in one session (NULL for platforms that do not support this statistic).</td>
</tr>
<tr>
<td>PEAK_PHYSICAL_WRITES</td>
<td>INTEGER</td>
<td>Maximum number of physical writes performed by this user on this database in one session (NULL for platforms that do not support this statistic).</td>
</tr>
<tr>
<td>PEAK_SYSTEM_CPUTIME</td>
<td>DEC(9,2)</td>
<td>Maximum system CPU time used by this user to access this database in one session (in seconds).</td>
</tr>
<tr>
<td>PEAK_USER_CPUTIME</td>
<td>DEC(9,2)</td>
<td>Maximum user CPU time used by this user to access this database in one session (in seconds).</td>
</tr>
<tr>
<td>PEAK_SPILL_COUNT</td>
<td>INTEGER</td>
<td>Maximum number of times that spill area was used by one session.</td>
</tr>
<tr>
<td>PEAK_PARALLELISM</td>
<td>INTEGER</td>
<td>Maximum number of parallel tasks performed by this user on this database in one session.</td>
</tr>
</tbody>
</table>
## Dynamic Statistic Tables

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEAK_MEMORY_USED</td>
<td>INTEGER</td>
<td>Maximum memory (in kilobytes) used by this user to access this database in one session.</td>
</tr>
<tr>
<td>PEAK_TEMPSPACE_USED</td>
<td>INTEGER</td>
<td>Maximum amount of spill area (in kilobytes) used by this user in one session.</td>
</tr>
<tr>
<td>LAST_UPDATED</td>
<td>TIMESTAMP</td>
<td>Timestamp of when this row was last updated.</td>
</tr>
</tbody>
</table>
Datatypes and Their Sizes

The size in bytes of each datatype is defined in the following table.

<table>
<thead>
<tr>
<th>Datatypes</th>
<th>Size in Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>Length (number of characters); maximum is 1,024</td>
</tr>
<tr>
<td>DATE</td>
<td>3</td>
</tr>
<tr>
<td>TIME</td>
<td>3 without fractional seconds; 5 with fractional seconds</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>6 without fractional seconds; 8 with fractional seconds</td>
</tr>
<tr>
<td>INTEGER</td>
<td>4 (range: (-2^{31} \text{ to } 2^{31}-1; 2^{31} = 2,146,483,648))</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>2 (range: (-2^{15} \text{ to } 2^{15}-1; 2^{15} = 32,768))</td>
</tr>
<tr>
<td>TINYINT</td>
<td>1 (range: (-2^{7} \text{ to } 2^{7}-1; 2^{7} = 128))</td>
</tr>
<tr>
<td>FLOAT, DOUBLE</td>
<td>8 (range: approximately 1.E-308 to 1.E308)</td>
</tr>
<tr>
<td>REAL</td>
<td>4 (range: approximately 1.E-38 to 1.E37)</td>
</tr>
<tr>
<td>DECIMAL or NUMERIC with precision:</td>
<td></td>
</tr>
<tr>
<td>1-2</td>
<td>1</td>
</tr>
<tr>
<td>3-4</td>
<td>2</td>
</tr>
<tr>
<td>5-9</td>
<td>4</td>
</tr>
<tr>
<td>10-11</td>
<td>5</td>
</tr>
<tr>
<td>12-14</td>
<td>6</td>
</tr>
<tr>
<td>15-16</td>
<td>7</td>
</tr>
<tr>
<td>17-18</td>
<td>8</td>
</tr>
<tr>
<td>19-21</td>
<td>9</td>
</tr>
<tr>
<td>22-23</td>
<td>10</td>
</tr>
<tr>
<td>24-26</td>
<td>11</td>
</tr>
<tr>
<td>27-28</td>
<td>12</td>
</tr>
<tr>
<td>29-31</td>
<td>13</td>
</tr>
<tr>
<td>32-33</td>
<td>14</td>
</tr>
<tr>
<td>34-35</td>
<td>15</td>
</tr>
<tr>
<td>36-38</td>
<td>16</td>
</tr>
</tbody>
</table>
Example: Using Segments with Time-Cyclic Data

This example illustrates how multiple segments can be used in a time-cyclic database. The example includes a simple star schema with a single STAR index, and the index is segmented like the data. The example demonstrates two techniques for management of time-cyclic data:

• Rolling off old segments and reusing them with new data.
• Creating new segments and adding them to the database.

This appendix includes the following sections:

• Background
• Rolling Off and Reusing Data and Index Segments
• Adding a New Segment
• Using an Offline Load Operation
• Deleting the Oldest Data
• Reusing the Segments
Example: Using Segments with Time-Cyclic Data

Background

The database used for this example contains three dimension (referenced) tables—Period, Product, and Market—and a fact (referencing) table, Sales, which contains sales data for two full years—eight quarters—plus the current quarter.

- Data for the Sales table is divided into quarterly segments, each quarter’s data 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, which resides in the same number of segments as the Sales table. The automatically created B-TREE index on the primary key columns has been dropped because the STAR index serves as the primary key index for the Sales table.
- Data and indexes for the dimension tables all reside in default segments.
- In the Period table, the primary key (Perkey) is of the DATE datatype.

The following figure illustrates the tables used in this example:

Assume the database was created during Q1 ‘97 and loaded with data for eight full quarters (all of 1995 and 1996) plus data for the current quarter, which is updated weekly. Q2 ‘97 is about to start. You have two tasks approaching:

- Before Q2 starts, you need to add a new segment to hold Q2 ‘97 data.
- After Q2 starts, you need to remove the Q1 ‘95 data.

Note: When you are ready to delete the oldest quarter’s data, you need to decide whether to save the segment for reuse by detaching it from the table and then reattaching it wherever you want to use it or whether to just drop the segment.
Example: Using Segments with Time-Cyclic Data

**Background**

**The Data Segments**

For the purpose of this example, assume the data segments for the Sales table are defined as follows:

These `CREATE SEGMENT` statements create these segments containing these files:

<table>
<thead>
<tr>
<th>Create Statement</th>
<th>Segment</th>
<th>Storage</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>create segment s_1q95</code></td>
<td>s_1q95</td>
<td>/disk1</td>
<td>s1, s2</td>
</tr>
<tr>
<td><code>create segment s_2q95</code></td>
<td>s_2q95</td>
<td>/disk2</td>
<td>s1, s2</td>
</tr>
<tr>
<td><code>create segment s_3q95</code></td>
<td>s_3q95</td>
<td>/disk3</td>
<td>s1, s2</td>
</tr>
<tr>
<td><code>create segment s_4q95</code></td>
<td>s_4q95</td>
<td>/disk4</td>
<td>s1, s2</td>
</tr>
<tr>
<td><code>create segment s_1q96</code></td>
<td>s_1q96</td>
<td>/disk5</td>
<td>s1, s2</td>
</tr>
<tr>
<td><code>create segment s_2q96</code></td>
<td>s_2q96</td>
<td>/disk6</td>
<td>s1, s2</td>
</tr>
<tr>
<td><code>create segment s_3q96</code></td>
<td>s_3q96</td>
<td>/disk7</td>
<td>s1, s2</td>
</tr>
<tr>
<td><code>create segment s_4q96</code></td>
<td>s_4q96</td>
<td>/disk8</td>
<td>s1, s2, s3</td>
</tr>
<tr>
<td><code>create segment s_1q97</code></td>
<td>s_1q97</td>
<td>/disk9</td>
<td>s1, s2, s3</td>
</tr>
<tr>
<td><code>create segment s_max</code></td>
<td>s_max</td>
<td>/disk10</td>
<td>s_max</td>
</tr>
</tbody>
</table>

Warehouse Administrator's Guide for UNIX Platforms  D-3
Example: Using Segments with Time-Cyclic Data

Background

The last segment, $s_{max}$, is an empty segment that is a placeholder at the high end of the range. This segment remains empty and therefore can be searched more quickly than a segment containing actual data. (When a segment is attached, the surrounding segments are searched to verify that they contain no data that overlaps the new segment range.) While such a segment is not necessary, its existence allows a new segment to be attached more quickly.

Note that the segments must be created before the tables that use them.
**Example: Using Segments with Time-Cyclic Data**

**Background**

**The Index Segments**

For the purpose of this example, assume the index segments for the Sales_star index are defined as follows:

<table>
<thead>
<tr>
<th>These CREATE SEGMENT statements</th>
<th>create these segments containing these files</th>
</tr>
</thead>
<tbody>
<tr>
<td>create segment star_1q95</td>
<td><img src="/disk1/star1" alt="star_1q95" /></td>
</tr>
<tr>
<td>storage '/disk1/star1' maxsize 200,</td>
<td><img src="/disk2/star1" alt="star_1q95" /></td>
</tr>
<tr>
<td>storage '/disk1/star2' maxsize 200;</td>
<td><img src="/disk3/star1" alt="star_1q95" /></td>
</tr>
<tr>
<td>create segment star_2q95</td>
<td><img src="/disk4/star1" alt="star_2q95" /></td>
</tr>
<tr>
<td>storage '/disk2/star1' maxsize 200,</td>
<td><img src="/disk5/star1" alt="star_2q95" /></td>
</tr>
<tr>
<td>storage '/disk2/star2' maxsize 200;</td>
<td><img src="/disk6/star1" alt="star_2q95" /></td>
</tr>
<tr>
<td>create segment star_3q95</td>
<td><img src="/disk7/star1" alt="star_3q95" /></td>
</tr>
<tr>
<td>storage '/disk3/star1' maxsize 200,</td>
<td><img src="/disk8/star1" alt="star_3q95" /></td>
</tr>
<tr>
<td>storage '/disk3/star2' maxsize 200;</td>
<td><img src="/disk9/star1" alt="star_3q95" /></td>
</tr>
<tr>
<td>create segment star_4q95</td>
<td><img src="/disk10/star1" alt="star_4q95" /></td>
</tr>
<tr>
<td>storage '/disk4/star1' maxsize 200,</td>
<td><img src="/disk1/star1" alt="star_4q95" /></td>
</tr>
<tr>
<td>storage '/disk4/star2' maxsize 200;</td>
<td><img src="/disk2/star1" alt="star_4q95" /></td>
</tr>
<tr>
<td>create segment star_1q96</td>
<td><img src="/disk1/star1" alt="star_1q96" /></td>
</tr>
<tr>
<td>storage '/disk5/star1' maxsize 200,</td>
<td><img src="/disk2/star1" alt="star_1q96" /></td>
</tr>
<tr>
<td>storage '/disk5/star2' maxsize 200;</td>
<td><img src="/disk3/star1" alt="star_1q96" /></td>
</tr>
<tr>
<td>create segment star_2q96</td>
<td><img src="/disk4/star1" alt="star_2q96" /></td>
</tr>
<tr>
<td>storage '/disk6/star1' maxsize 200,</td>
<td><img src="/disk5/star1" alt="star_2q96" /></td>
</tr>
<tr>
<td>storage '/disk6/star2' maxsize 200;</td>
<td><img src="/disk6/star1" alt="star_2q96" /></td>
</tr>
<tr>
<td>create segment star_3q96</td>
<td><img src="/disk7/star1" alt="star_3q96" /></td>
</tr>
<tr>
<td>storage '/disk7/star1' maxsize 200,</td>
<td><img src="/disk8/star1" alt="star_3q96" /></td>
</tr>
<tr>
<td>storage '/disk7/star2' maxsize 200;</td>
<td><img src="/disk9/star1" alt="star_3q96" /></td>
</tr>
<tr>
<td>create segment star_4q96</td>
<td><img src="/disk10/star1" alt="star_4q96" /></td>
</tr>
<tr>
<td>storage '/disk8/star1' maxsize 200,</td>
<td><img src="/disk1/star1" alt="star_4q96" /></td>
</tr>
<tr>
<td>storage '/disk8/star2' maxsize 200;</td>
<td><img src="/disk2/star1" alt="star_4q96" /></td>
</tr>
<tr>
<td>create segment star_1q97</td>
<td><img src="/disk3/star1" alt="star_1q97" /></td>
</tr>
<tr>
<td>storage '/disk9/star1' maxsize 200,</td>
<td><img src="/disk4/star1" alt="star_1q97" /></td>
</tr>
<tr>
<td>storage '/disk9/star2' maxsize 200;</td>
<td><img src="/disk5/star1" alt="star_1q97" /></td>
</tr>
<tr>
<td>create segment star_max</td>
<td><img src="/disk10/star_max" alt="star_max" /></td>
</tr>
<tr>
<td>storage '/disk10/star_max' maxsize 16;</td>
<td><img src="/disk10/star_max" alt="star_max" /></td>
</tr>
</tbody>
</table>
Example: Using Segments with Time-Cyclic Data

Background

The last segment, \textit{star\_max}, is an empty segment that is a place holder at the high end of the range. This segment remains empty and therefore can be searched more quickly than a segment containing actual data. (When a segment is attached, the surrounding segments are searched to verify that they contain no data that overlaps the new segment range.) While such a segment is not necessary, its existence allows a new segment to be attached more quickly.

Note that the segments must be created before the indexes that use them.
Example: Using Segments with Time-Cyclic Data

Background

The Tables

Assume the Period table resides in a default segment and is defined as follows:

```
create table period (  
    perkey date not null,  
    month char (15),  
    year integer,  
    quarter integer,  
    tri integer,  
    primary key (perkey))  
maxrows per segment 2048;
```

Assume the Sales table was created and segmented by quarters with the following CREATE TABLE statement. The segment specification assigns the segments, segmenting column, and ranges of data that can reside in each segment. The Perkey column is the segmenting column; therefore, the ranges indicate values in the Perkey column. The range for each segment defines one quarter, so each quarter of sales data resides in a separate segment.

```
create table sales (  
    perkey date not null,  
    prodkey integer not null,  
    mktkey integer not null,  
    dollars decimal (7, 2),  
    weight smallint,  
    primary key (perkey, prodkey, mktkey),  
    foreign key (perkey) references period (perkey),  
    foreign key (prodkey) references product (prodkey),  
    foreign key (mktkey) references market (mktkey))  
data in (s_1q95, s_2q95, s_3q95, s_4q95, s_1q96, s_2q96,  
    s_3q96, s_4q96, s_1q97, s_max)  
segment by values of (perkey) ranges (  
    min:DATE'1995-04-01',  
    DATE'1995-04-01':DATE'1995-07-01',  
    DATE'1995-07-01':DATE'1995-10-01',  
    DATE'1995-10-01':DATE'1996-01-01',  
    DATE'1996-01-01':DATE'1996-04-01',  
    DATE'1996-04-01':DATE'1996-07-01',  
    DATE'1996-07-01':DATE'1996-10-01',  
    DATE'1996-10-01':DATE'1997-01-01',  
    DATE'1997-01-01':DATE'1997-04-01',  
    DATE'1997-04-01': max);
```
**The STAR Index**

Assume a STAR index covering all of the foreign keys is created on the Sales table, and that the STAR index is segmented *exactly* like the data; that is, the index entries corresponding to each row of data in the s_1q95 data segment are in the star_1q95 index segment, the index entries corresponding to each row of data in the s_2q95 data segment are in the star_2q95 index segment, and so on. This is accomplished by specifying the range for each segment to include only the dates corresponding to a given quarter.

```sql
create star index sales_star
  on sales (perkey, prodkey, mktkey)
  in (star_1q95, star_2q95, star_3q95, star_4q95, star_1q96, star_2q96, star_3q96, star_4q96, star_max)
  segment by references of (perkey)
```

**Note:** The ranges in the CREATE INDEX statement refer to row numbers in the referenced table *Period*, not row numbers in the Sales table. For the complete syntax of the CREATE INDEX statement, refer to SQL Reference Guide.

To find the boundaries for the STAR index segment ranges, query the RBW ROWNUM pseudo-column and the primary key column (Date) of the Period table (if the Period table was segmented, you would also need to find the segment name which is stored in the RBW SEGNAME pseudo-column). The following query returns the range boundaries for this STAR index:

```sql
select rbw_rownum, date
from period
where date = '04-01-95'
  or date = '07-01-95'
  or date = '10-01-95'
  or date = '01-01-96'
  or date = '04-01-96'
  or date = '07-01-96'
  or date = '10-01-96'
  or date = '01-01-97'
  or date = '04-01-97';
```

Notice the constraints on the Date column in the WHERE clause of this query correspond to the range boundaries from the CREATE TABLE statement for the Sales table.
The following figure illustrates how sales data is mapped to the data segments and to the STAR index segments.

<table>
<thead>
<tr>
<th>Sales data for these ranges</th>
<th>is mapped to these data segments</th>
<th>and these STAR index segments</th>
</tr>
</thead>
<tbody>
<tr>
<td>min:DATE'1995-04-01'</td>
<td>s_1q95</td>
<td>star_1q95</td>
</tr>
<tr>
<td>DATE'1995-04-01':DATE'1995-07-01'</td>
<td>s_2q95</td>
<td>star_2q95</td>
</tr>
<tr>
<td>DATE'1995-07-01':DATE'1995-10-01'</td>
<td>s_3q95</td>
<td>star_3q95</td>
</tr>
<tr>
<td>DATE'1995-10-01':DATE'1996-01-01'</td>
<td>s_4q95</td>
<td>star_4q95</td>
</tr>
<tr>
<td>DATE'1996-01-01':DATE'1996-04-01'</td>
<td>s_1q96</td>
<td>star_1q96</td>
</tr>
<tr>
<td>DATE'1996-04-01':DATE'1996-07-01'</td>
<td>s_2q96</td>
<td>star_2q96</td>
</tr>
<tr>
<td>DATE'1996-07-01':DATE'1996-10-01'</td>
<td>s_3q96</td>
<td>star_3q96</td>
</tr>
<tr>
<td>DATE'1996-10-01':DATE'1997-01-01'</td>
<td>s_4q96</td>
<td>star_4q96</td>
</tr>
<tr>
<td>DATE'1997-01-01':DATE'1997-04-01'</td>
<td>s_1q97</td>
<td>star_1q97</td>
</tr>
<tr>
<td>DATE'1997-04-01':max</td>
<td>s_max</td>
<td>star_max</td>
</tr>
</tbody>
</table>

Each segment includes the lower end of its range but excludes the upper end; for example, the segment \( s_{1q97} \) includes the data for 1997-01-01 but does not include data for 1997-04-01. Similarly, the STAR index segment \( star_{1q97} \) includes the index entries for 1997-01-01 data, but does not include index entries for 1997-04-01 data.

Note that the last segment, \( s_{max} \), is designated to contain data with a date of April 1, 1997 or later. At this time, however, the segment is empty because no such data has yet been loaded.
**Example: Using Segments with Time-Cyclic Data**

*Background*

**The Data**

Assume data is loaded into the Sales and Period tables as follows:

```sql
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
  };
```

**Note:** The Product and Market tables must be loaded before the Sales table; however, the contents of those tables are not relevant to this example.
Example: Using Segments with Time-Cyclic Data

Background

The following figure illustrates the data format in the Period and Sales tables and how the data is distributed among the segments; the index for the Period table resides in a separate default segment, the STAR index for the Sales table is segmented like the data as shown on page D-9.

**period.txt**

<table>
<thead>
<tr>
<th>Date</th>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>01/92/01:JAN:1992:1:1</td>
<td>default_segment</td>
</tr>
<tr>
<td>01/92/02:JAN:1992:1:1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>12/96/31:DEC:1996:4:3</td>
<td></td>
</tr>
</tbody>
</table>

**sales.txt**

<table>
<thead>
<tr>
<th>Date</th>
<th>Segment</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>01/95/01:00:00:02:890:80</td>
<td>s_1q95</td>
<td>min:DATE'1995-04-01'</td>
</tr>
<tr>
<td>03/95/31:31:19:400:40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>04/95/01:21:04:703:71</td>
<td>s_2q95</td>
<td>DATE'1995-04-01':DATE'1995-07-01'</td>
</tr>
<tr>
<td>06/95/30:20:10:559:53</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>07/96/01:01:04:456:45</td>
<td>s_3q96</td>
<td>DATE'1996-07-01':DATE'1996-10-01'</td>
</tr>
<tr>
<td>09/96/30:10:06:897:81</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10/96/01:31:04:225:20</td>
<td>s_4q96</td>
<td>DATE'1996-10-01':DATE'1997-01-01'</td>
</tr>
<tr>
<td>12/96/31:21:10:111:10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01/97/01:00:02:725:20</td>
<td>s_1q97</td>
<td>DATE'1997-01-01':DATE'1997-04-01'</td>
</tr>
<tr>
<td>03/97/31:31:19:321:10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Assume that the database holds a rolling 9 quarters of data at a time. That means that when it is time to add the data for the second quarter of 1997, you can remove the data from the first quarter of 1995. This section shows a procedure to reuse the \texttt{s\_1q95} data and \texttt{star\_1q95} index segments for the data and index entries for the second quarter of 1997.

\textbf{Note:} This procedure only works if you have the data and the index(es) segmented identically, as outlined in the previous section.

1. Take the segment containing the oldest data offline:
   \begin{verbatim}
   alter segment s\_1q95 of table sales offline;
   \end{verbatim}
2. Detach the segment from the table (this removes all data from the segment, so make sure this is what you want to do!):
   \begin{verbatim}
   alter segment s\_1q95 of table sales detach
   override fullindexcheck on segments (star\_1q95);
   \end{verbatim}

\textbf{Caution:} The OVERRIDE FULLINDEXCHECK clause is designed to take advantage of the fact that all index entries corresponding to the \texttt{s\_1q95} data segment are stored in the \texttt{star\_1q95} index segment. This dramatically speeds the performance of the DETACH operation. If your data and indexes are not identically segmented, do not include the OVERRIDE FULLINDEXCHECK clause in your DETACH operation. Omitting the clause ensures that any index entries in other parts of the index are removed.

3. Take the index segment corresponding to the oldest data offline:
   \begin{verbatim}
   alter segment star\_1q95 of index sales\_star offline;
   \end{verbatim}
4. Detach the index segment from the index (this removes all index entries from the segment):
   \begin{verbatim}
   alter segment star\_1q95 of index sales\_star detach;
   \end{verbatim}
5. Rename the old data segment, which will now hold data for the second quarter of 1997:
   \begin{verbatim}
   alter segment s\_1q95 rename s\_2q97;
   \end{verbatim}
6. Rename the old index segment:
   \begin{verbatim}
   alter segment star\_1q95 rename star\_2q97;
   \end{verbatim}
7. Make any other needed changes to the segments, for example, MAXSIZE, PATH, or adding new storage.
8. Attach the newly renamed data segment to the table:

   alter segment s_2q97 attach to table sales
   range (DATE'1997-04-01':DATE'1997-07-01');

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

   DATE'1997-07-01':max

   Attaching a segment automatically sets the segment to ONLINE mode.

   **Note:** The Period table must contain rows corresponding to the days in the new quarter, otherwise new data inserted into the Sales table would be discarded due to referential integrity failure.

9. Attach the newly renamed index segment to the table:

   alter segment star_2q97 attach to index sales_star
   range (821:899);

10. You can now use the TMU or SQL INSERT statements to populate the Sales table with data for the second quarter of 1997.
Adding a New Segment

As the end of Q2 '97 approaches, you want to add another segment between \( s_{2q97} \) and \( s_{max} \) to hold data for the next quarter. Because you do not have any unused segments available, create a segment and attach it to the Sales table. As data is entered for Q3 '97, it will be placed in this new segment.

**Note:** 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, as discussed on page D-12 and page D-19.

1. Create a new segment named \( s_{3q97} \):
   ```sql
   create segment s_3q97
   storage '/disk10/s1' maxsize 200,
   storage '/disk10/s2' maxsize 200,
   storage '/disk10/s3' maxsize 200;
   ```

2. Add the additional Perkey entries to the Period table to cover the additional date ranges for quarter 3 of 1997. Omitting this step will cause referential integrity failures when the data for the new quarter is loaded. For example, enter:
   ```sql
   insert into period values
      (DATE'1997-07-01', 'JULY', 1997, 3, 2);
   ```
   ...
   ```sql
   insert into period values
      (DATE'1997-08-01', 'AUG', 1997, 3, 2);
   ```
   ...
   ```sql
   insert into period values
      (DATE'1997-09-01', 'SEPT', 1997, 3, 3);
   ```
   ...

3. Attach the \( s_{3q97} \) segment to the Sales table, specifying the date range for quarter 3 of 1997:
   ```sql
   alter segment s_2q97 attach to table sales
   range (DATE'1997-07-01':DATE'1997-10-01');
   ```

   The range of the \( s_{max} \) segment automatically moves to the range:

   ```sql
   DATE'1997-10-01':max
   ```

   Attaching a segment automatically sets the segment to ONLINE mode.

4. Create a new index segment named \( star_{3q97} \):
   ```sql
   create segment star_3q97
   storage '/disk10/star1' maxsize 200,
   storage '/disk10/star2' maxsize 200,
   storage '/disk10/star3' maxsize 200;
   ```

5. Attach the new index segment to the index:
Example: Using Segments with Time-Cyclic Data

Adding a New Segment

```
alter segment star_3q97 attach to index sales_star
  range (899:992);
```

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

```
<table>
<thead>
<tr>
<th>Sales</th>
<th>min:DATE'1995-07-01'</th>
</tr>
</thead>
<tbody>
<tr>
<td>s_2q95</td>
<td>DATE'1995-07-01:DATE'1995-10-01'</td>
</tr>
<tr>
<td>s_3q95</td>
<td>DATE'1995-10-01:DATE'1996-01-01'</td>
</tr>
<tr>
<td>s_4q95</td>
<td>DATE'1996-01-01:DATE'1996-04-01'</td>
</tr>
<tr>
<td>s_1q96</td>
<td>DATE'1996-04-01:DATE'1996-07-01'</td>
</tr>
<tr>
<td>s_2q96</td>
<td>DATE'1996-07-01:DATE'1996-10-01'</td>
</tr>
<tr>
<td>s_3q96</td>
<td>DATE'1996-10-01:DATE'1997-01-01'</td>
</tr>
<tr>
<td>s_4q96</td>
<td>DATE'1997-01-01:DATE'1997-04-01'</td>
</tr>
<tr>
<td>s_1q97</td>
<td>DATE'1997-04-01:DATE'1997-07-01'</td>
</tr>
<tr>
<td>s_2q97</td>
<td>DATE'1997-07-01:DATE'1997-10-01'</td>
</tr>
<tr>
<td>s_3q97</td>
<td>DATE'1997-07-01:max Notice that this segment shifts its lower boundary and retains the maximum boundary.</td>
</tr>
<tr>
<td>s_max</td>
<td></td>
</tr>
</tbody>
</table>
```

Now you can load data into the new segment, using either the standard or the offline load procedure.
**Example: Using Segments with Time-Cyclic Data**

**Using an Offline Load Operation**

Assume you want to perform an offline load operation to load a batch of data into the new segment `s_3q97`.

1. Set the segment to OFFLINE mode:
   ```sql
   alter segment s_3q97 of table sales offline;
   ```

2. Create an extra segment to provide working space for building indexes.
   ```sql
   create segment work01 storage 'work01' maxsize 100;
   ```

3. Create a TMU control file (`s3q97_input`) that contains:
   - A LOAD DATA statement to read the data input file (`sales_97_data`) and map each field into a column in the offline segment.
   - A SYNCH statement to synchronize the indexes of the Sales table with the new data.
   ```sql
   load data
   inputfile 'sales_97_data'
   append
   separated by ':'
   discardfile 'discards_sales_97'
   discards 3
   into offline segment s_3q97 of table sales
   working_space work01 (perkey date 'MM/Y*/DD',
   prodkey integer external,
   mktkey integer external,
   dollars integer external);
   synch offline segment s_3q97 with table sales
   discardfile 'discards_synch';
   ```

   **Note:** Because the SYNCH operation locks the table, you might prefer to load the data with one control statement and perform the SYNCH operation with another control statement at another time.

4. Log in as the `redbrick` user.
5. Verify that the `redbrick_dir/bin` directory is in your path.
6. Change to the directory that contains the `s3q97_input` file.
7. Invoke the TMU and load the new data by entering the following command.
   ```bash
   rb_tmu -d database_name s_3q97_input system password
   ```
   The segment `s_3q97` now contains the new data.
Example: Using Segments with Time-Cyclic Data

Deleting the Oldest Data

8. Set the segment s_3q97 to ONLINE mode:
   ```sql
   alter segment s_3q97 of table sales online;
   ```

9. Drop the working segment:
   ```sql
   drop segment work01;
   ```

Deleting the Oldest Data

Now you need to remove data for the oldest quarter, the second quarter of 1995, which resides in the s_2q95 segment.

1. If you want to save the data, use the TMU to unload the segment to a file or tape.
2. Invoke RISQL (or the tool you use for administrative activities), connecting the database that you want to modify.
3. Set the s_2q95 segment to OFFLINE mode:
   ```sql
   alter segment s_2q95 of table sales offline;
   ```
4. Detach the s_2q95 segment:
   ```sql
   alter segment s_2q95 of table sales detach;
   ```
   or:
   ```sql
   alter segment s_2q95 of table sales detach
   override fullindexcheck on segments (star_2q95);
   ```
   See page D-12 for a caution about OVERRIDE FULLINDEXCHECK.
5. Modify the Period table by deleting the period keys for the old data (second quarter of 1995):
   ```sql
   delete from period where perkey < DATE'1995-07-01';
   ```
6. Detach the star_2q95 index segment:
   ```sql
   alter segment star_2q95 of index sales_star detach;
   ```
7. You can either save these segments for reuse or drop them with DROP statements:
   ```sql
   drop segment s_2q95;
   drop segment star2q95;
   ```
Example: Using Segments with Time-Cyclic Data
Deleting the Oldest Data

The ranges are now as follows:

<table>
<thead>
<tr>
<th>Sales</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s_3q95</td>
<td>min:DATE’1995-10-01'</td>
</tr>
<tr>
<td>s_3q95</td>
<td>DATE’1995-10-01':DATE’1996-01-01'</td>
</tr>
<tr>
<td>s_1q96</td>
<td>DATE’1996-01-01':DATE’1996-04-01'</td>
</tr>
<tr>
<td>s_2q96</td>
<td>DATE’1996-04-01':DATE’1996-07-01'</td>
</tr>
<tr>
<td>s_3q96</td>
<td>DATE’1996-07-01':DATE’1996-10-01'</td>
</tr>
<tr>
<td>s_4q96</td>
<td>DATE’1996-10-01':DATE’1997-01-01'</td>
</tr>
<tr>
<td>s_1q97</td>
<td>DATE’1997-01-01':DATE’1997-04-01'</td>
</tr>
<tr>
<td>s_2q97</td>
<td>DATE’1997-04-01':DATE’1997-07-01'</td>
</tr>
<tr>
<td>s_3q97</td>
<td>DATE’1997-07-01':DATE’1997-10-01'</td>
</tr>
<tr>
<td>s_max</td>
<td>DATE’1997-10-01':max</td>
</tr>
</tbody>
</table>

s_2q95 is detached from the table and dropped from database. The next segment covers the minimum range.
Reusing the Segments

If you chose not to drop the old segment `s_2q95`, you can reuse it when you need a new segment for Q4 '97 data.

1. If you want to rename this segment, use the ALTER SEGMENT command:
   ```
   alter segment s_2q95 rename s_4q97;
   ```

2. Attach it to the Sales table with the new range:
   ```
   alter segment s_4q97 attach to table sales
   range (DATE'1997-10-01':DATE'1998-01-01');
   ```

   **Note:** The Period table must contain rows corresponding to the days in the new quarter, otherwise new data inserted into the Sales table would be discarded due to referential integrity failure.

3. Make any other needed changes, for example, MAXSIZE, PATH, or adding new storage.

4. Rename the old index segment:
   ```
   alter segment star_2q95 rename star_4q97;
   ```

5. Make any other needed changes to the segments, for example, MAXSIZE, PATH, or adding new storage.

6. Attach the newly renamed index segment to the table:
   ```
   alter segment star_4q97 attach to index sales_star
   range (992:1084);
   ```

7. The data and index segments are now ready to load data either using the TMU or with INSERT statements.
Example: Using Segments with Time-Cyclic Data
Reusing the Segments
Symbols
.odbc.ini file, processing of 2-18
.rbretc file, processing of 2-17
.rbwerr file, defined 2-17
.rbwrc file, usage 2-16

Numerics
7-bit ASCII character set 2-22
8-bit character sets 2-22

A
access, database 6-1 to 6-33
ACCESS_ADVISOR_INFO task authorization 6-9
ACCESS_ANY task authorization 6-9
ACCESS_SYSINFO task authorization 6-9, 7-4
accounting changing modes 7-36
configuring 7-34
files 7-33
job mode 7-31
operations generating records 7-31
overview 7-31
process 7-32
rbwlogd daemon 7-32
record format 7-32
sample code for 7-32
setting mode 7-35
starting 7-36
startup state 7-34
stopping 7-36
switching files 7-36
workload mode 7-31

ACCOUNTING configuration parameter
description 7-34
rbw.config file entry B-19
accounting file
location 7-33, 7-34
removing 7-33
size 7-33, 7-35
switching 7-36
accounting records 7-32
accounts, see user accounts
ACCT_DIRECTORY configuration parameter
described 7-34
rbw.config file entry B-19
use with accounting files 7-33
ACCT_LEVEL configuration parameter
described 7-35
rbw.config file entry B-19
ACCT_MAXSIZE configuration parameter
described 7-35
rbw.config file entry B-19
use with accounting files 7-33
ADD STORAGE clause, ALTER SEGMENT command, usage 8-11
ADMIN ADVISOR_LOGGING parameter
described 7-26
modifying 8-34
rbw.config file entry B-19
ADMIN database
described 7-4
rbw.config file entry B-22
ADMIN parameters
rbw.config file entries B-19
administrative account, see redbrick user account
administration daemon 2-30
administration database 7-4
Advisor
described 2-11
logging 7-24
Advisor log files
removing 7-24
specifying size 7-29
ADVISOR_LOG_DIRECTORY configuration parameter
rbw.config file entry B-20
use with log files 7-24
ADVISOR_LOG_MAXSIZE configuration parameter
described 7-29
rbw.config file entry B-20
use with log files 7-24
ADVISOR_LOGGING, ADMIN parameter 7-26, B-19
ADVISOR_LOGGING, OPTION parameter 7-28, B-13
aggregated data, example 3-23
ALLOW_POSSIBLE_DEADLOCKS parameter
modifying 8-35
rbw.config file entry B-11
syntax and usage 8-3
ALTER SYSTEM command
activating a database 7-9
Advisor logging 7-27
canceling a command 7-9
changing logging level 7-23
changing modes 7-36
changing user priority 7-10
clauses
ADVISOR_LOGGING 7-27
CANCEL USER COMMAND 7-9
CHANGE ACCOUNTING LEVEL 7-36
CHANGE LOGGING LEVEL 7-23
CHANGE USER PRIORITY 7-10
CLOSE USER SESSION 7-10
QUIESCE DATABASE 7-8
RESET STATISTICS 7-9
RESUME DATABASE 7-9
START ACCOUNTING 7-9
START LOGGING 7-22
ALTER SYSTEM command clauses (continued)
STOP ACCOUNTING 7-36
STOP LOGGING 7-22
SWITCH ACCOUNTING FILE 7-36
SWITCH LOGGING FILE 7-22
SWITCH_ADVISOR_LOG_FILE 7-27
TERMINATE LOGGING
DAEMON 7-23
closing a session 7-10
quiescing a database 7-8
required authorizations 7-8
resetting statistics 7-9
starting accounting 7-36
starting logging 7-22
stopping accounting 7-36
stopping logging 7-22
switching accounting file 7-36
switching log files 7-22
terminating rbwlogd 7-23
ALTER TABLE command
interrupted operations 8-28
usage 8-24 to 8-28
ALTER USER command 7-10
ALTER_ANY task authorization 6-9
ALTER_SYSTEM task authorization 6-9, 7-4
ARITHABORT parameter
modifying 8-35
rbw.config file entry B-11
arithmetic errors, options B-11
Aroma database
building, tutorial A-1 to A-23
rbw.config file entry B-22
ASCII character set 2-22
associative tables 3-9
attributes, influence on schema
design 3-20
audit events 7-18
auditing, See event logging
authorizations, See task authorizations
AUTO_AGGREGATE license, rbw.config file entry B-12
AUTO_INVALIDATE_PRECOMPUTED_VIEW parameter
modifying 8-35
rbw.config file entry B-17
Automatic Row Generation, TMU and referential integrity 2-35
AUTOROWGEN parameter modifying 8-35
rbw.config file entry B-11

B
backup operations, during index creation 5-12
BACKUP RESTORE license, rbw.config file entry B-12
backup segment 8-11
BACKUP_DATABASE task authorization 6-9
binary character comparisons 2-23
bit vector sort operator 9-29
block, 8K 8-5
browsing dimension tables 3-19
B-TREE 1-1 match operator 9-28
B-TREE indexes, See indexes, B-TREE
B-TREE scan operator 9-29
buffer cache size 9-11

cache I/O statistics 7-6
CASCADE keyword, usage 5-10
cascaded delete operations 2-35 to 2-41, 5-10
cases, tracked by technical support xxvi
CATEGORY value, macro 5-17
cautions
FORCE INTACT for segments 8-16
low fill factors 9-23
OVERRIDE FULLINDEXCHECK D-12
OVERRIDE REFCHECK 5-11
warehouse limits 1-13
CHANGE MAXSIZE clause, ALTER SEGMENT command 8-11
CHANGE PATH clause, ALTER SEGMENT command 8-11
character data
binary and linguistic sorts 2-23
comparing and sorting 2-23
sort methods 2-23
character encoding defined 2-22
See also character set
character set
ASCII 2-22
codings 2-26, 2-29
defined 2-22
effect of changing 2-26
multibyte 2-23
choose plan operator 9-30
cleanup, temporary space 9-10
CLEANUP_SCRIPT parameter modifying file 8-34
rbw.config file entry B-10
syntax and usage 9-10
client tools
locales for 2-26
non-Red Brick 2-26
RISQL Entry Tool and RISQL Reporter 2-26
client/server environment
compatibility issues 2-28
different locales in 2-26
code page defined 2-22
See also character set
collation sequence, defined 2-23
columns
adding 8-24
changing datatype 8-26
default values, changing 8-25
dropping 8-24
estimating width 4-20
renaming 8-25
COMMENT value, macro 5-17
commit operation 8-2
complete family, defined 2-36
configuration file, See rbw.config file
Index

configuration parameter
ACCOUNTING 7-34, B-19
ACCT_DIRECTORY 7-33, 7-34, B-19
ACCT_LEVEL 7-35, B-19
ACCT_MAXSIZE 7-33, 7-35, B-19
ADMIN ADVISOR_LOGGING 7-26
ADVISOR_LOG_DIRECTORY 7-24
ADVISOR_LOG_MAXSIZE 7-24, 7-29, B-20
AUTO_INVALIDATE_PRECOMPUTED_ VIEW B-17
GRANT_TEMP_RESOURCE_TO_ALL
B-17
LOG_Audit_LEVEL 7-21, B-20
LOG_DIRECTORY 7-19, 7-20, B-20
LOG_ERROR_LEVEL 7-21, B-20
LOG_MAXSIZE 7-19, 7-21, B-20
LOG_OPERATIONAL_LEVEL 7-21, B-20
LOG_SCHEMA_LEVEL 7-21, B-20
LOG_USAGE_LEVEL 7-21, B-21
LOGGING 7-20, B-20
NLS_LOCALE LOCALE 2-24
NLS_LOCALE MESSAGE_DIR 2-29
OPTION ADVISOR_LOGGING 7-28, B-13
PRECOMPUTED_VIEW_QUERY_rewrite B-16
RENICE_COMMAND 7-10, B-21
REPORT_INTERVAL 7-12, B-21
TEMPORARY_DATA_SEGMENT B-17
TEMPORARY_INDEX_SEGMENT B-17
USE_INVALIDATE_PRECOMPUTED_ VIEWS B-17
configuration, overview of system 2-16
CONNECT system role, described 2-32, 6-5
connections limit
changing 8-34
rbw.config file entry B-10
CONSTRAINT keyword 5-10
controlling database activity 7-8

conventions
syntax diagrams xxii
syntax notation xxi
copy management option 7-3
correlated subquery 9-57
COUNT function B-11
COUNT_RESULT parameter, rbw.config
file entry B-11
CPUs, allocation for parallel
queries 10-29
CREATE INDEX command,
usage 5-12 to 5-14
CREATE MACRO command,
usage 5-17 to 5-20
CREATE ROLE command, usage 6-12
CREATE SEGMENT command, usage 5-8
CREATE TABLE command
CONSTRanMent keyword 5-10
examples, Aroma database A-7
usage 5-9
CREATE VIEW command,
usage 5-15 to 5-16
CREATE_ANY task authorization 6-9
creating a database, Aroma
tutorial A-1 to A-24
CROSS JOIN parameter
modifying 8-35
rbw.config file entry B-11
cross-reference tables 3-9
CURRENT_USER, SQL variable 5-16
custom roles, See user-created roles
Customer Support Center xxv

D
daemon processes
general discussion 1-7
See also rbwapid, rbw.servermon,
rbwlogd, or rbwadmd daemon
data, referential integrity
during delete 2-35 to 2-41, 5-10
during insert and load 2-35
database directory, defined 2-12
database locale, See warehouse locale
database objects, dropping 8-40 to 8-43
database sizing tool (dbsize) 4-20
databases
   access to 6-1 to 6-33
   activating 7-9
   adding new 5-4
   adding users 6-3
   changing passwords 6-4, A-6
   controlling activity 7-8
   copying 8-15, 8-30
   creating
      example, Aroma A-1 to A-24
      overview 1-9
      procedure 5-1 to 5-20
   default segment location 5-8
   deleting 8-43
   dropping database objects 8-40 to 8-43
   growth patterns 4-44
   limits and maximum sizes 1-13
   loading data  A-20
   logical names 2-12
      defining 5-4
         rbw.config file entry B-22
   monitoring activity 7-5
   moving 8-15, 8-30
   organizing data into tables 4-2
   password security 6-24 to 6-33
   physical implementation 2-4 to 2-11
   quiescing 7-8
   sizing example, database 4-24 to 4-31
   system account and password 5-5
   table and index sizes,
      estimating 4-18 to 4-24
   when to lock 8-2
   datatypes
      changing column 8-26
      sizes C-39
   DBA system role
      described 2-32, 6-5
      granting 6-6, 6-14
      revoking 6-6, 6-18
      See also warehouse administrator
dbsize tool 4-20
deadlocks 8-3
decision-support databases 3-3
default locale 2-24

DEFAULT_DATA_SEGMENT parameter
   modifying 8-35
   rbw.config file entry  B-18
   syntax and usage 9-15
   with multiple databases 5-8

DEFAULT_INDEX_SEGMENT parameter
   modifying 8-35
   rbw.config file entry  B-18
   syntax and usage 9-15
   with multiple databases 5-8
delete cascade operator 9-31
delete operations
   locking tables 8-2
   modes, actions 5-10
delete operator 9-30
   delete refcheck operator 9-31
directories for temporary
   space 4-32 to 4-40, 9-2 to 9-10
   DIRECTORY, INDEX_TEMPSPACE, See
   INDEX_TEMPSPACE parameters
   disk groups
      defining 10-4
      effect on parallel queries 10-24
      specifying processes 10-6
disk space
      allocation 4-44, 5-8
      offline load requirements 4-37
      organizing with segmented
         storage 4-41 to 4-43
      re-use 8-6
      table and index sizes,
         estimating 4-18 to 4-24
      table growth, planning for 4-44
      temporary space requirements,
         estimating 4-34 to 4-40
disk spill files, removing 9-10
disk usage, parallel queries 10-26
divide-by-zero errors, options B-11
documentation
   list of Red Brick Systems xix
   support xxvii
domains, for TARGET indexes
   defined 4-8
   specifying size 4-8
   use with TARGETjoin 4-16 to 4-17
DROP commands 8-40 to 8-43
DROP ROLE command, usage 8-41
DROP_ANY task authorization 6-9
Index

DST_COMMANDS table  C-27
DST_DATABASES table  C-29
DST_LOCKS table  C-30
DST_SESSIONS table  C-31
DST_USERS table  C-35
dynamic statistic tables
  and RBW_TABLES table  7-5
  definition of read statistics  7-6
  definition of write statistics  7-7
DST_COMMANDS  C-27
DST_DATABASES  C-29
DST_LOCKS  C-30
DST_SESSIONS  C-31
DST_USERS  C-35
overview  7-5
refresh interval  7-12
statistic collection interval  7-11

E
e-mail addresses, for Red Brick
Systems  xxv
Enterprise Control and Coordination option
  features included  7-2
  license, rbw.config file entry  B-12
Enterprise Copy Management option
  described  7-3
  license, rbw.config file entry  B-12
environment variables
  RB_CONFIG, described  2-18
  RB_DSN, described  2-18
  RB_HOST, described  2-18
  RB_NLS_LOCALE  2-27
  RB_PATH, described  2-18
  RISQL Entry Tool  6-2
  user accounts  6-2
error events  7-18

event logging
  categories  7-17
  changing severity  7-23
  configuring  7-20
  log daemon  7-13
  log files  7-19
  logging subsystem  7-13
  message templates  7-14
  messages  7-17
  overview  7-13
  severity levels  7-17
  starting  7-22
  startup state  7-20
  USAGE ROUTINE  7-19
  exchange operator  9-31
  execute operator  9-32
  EXPAND command  5-19
  expected domain size  4-8
  expiration, passwords  6-26
  expired user accounts  6-26
  EXPLAIN command  9-37
  extended star schema  3-17

F
fact-to-fact joins
  rules for STARjoin  4-4
  using synonyms with  9-54
file groups, See disk groups
FILE_GROUP parameter
  described  10-2
  modifying  8-35
  rbw.config file entry  B-15
  syntax and usage  10-4
  See also disk groups
file-system-full messages  8-4
fill factors
  changing  8-10, 9-22
  described  4-21 to 4-23
  effect on index key size  9-23
  setting  9-20
FILLFACTOR parameters
  modifying  8-35
  rbw.config file entries  B-15
Findserver utility
  INTERVAL  B-11
  usage  8-36
  FOR DELETE option  8-2
FORCE INTACT to verify segment  8-16
FORCE_AGGREGATION_TASKS parameter
  described 10-2
  rbw.config file entry  B-14
FORCE_FETCH_TASKS parameter
  described  10-2
  rbw.config file entry  B-14
FORCE_HASHJOIN_TASKS parameter
  described  10-21
  rbw.config file entry  B-14
FORCE_JOIN_TASKS parameter
  described 10-2
  rbw.config file entry  B-14
FORCE_SCAN_TASKS parameter
  described 10-2
  rbw.config file entry  B-14
FOREIGN KEY REFERENCES clause, relating tables  4-44
  foreign keys
    adding and dropping  8-27
    defined 3-6
    multi-column 3-10
    functional join operator 9-32
GUI tool, Red Brick Warehouse Administrator 2-19

H
hash 1-1 match operator  9-32
hash AVL aggregate operator 9-33
hash join 4-10

I
I/O contention, parallel queries  10-26
IGNORE_OPTICAL_INDEXES parameter
  rbw.config file entry  B-19
  syntax and usage 8-22
IGNORE_PARTIAL_INDEXES parameter
  modifying 8-35
  rbw.config file entry  B-18
  syntax and usage 9-18
IGNORE_QUIESCE task
  authorization 6-9
immediate family, defined 2-36
INDEX_TEMPSPACE parameters 9-2 to 9-10
  current values 9-10
  described 4-33 to 4-40
determining values for 4-33 to 4-40
  DIRECTORY(IES)
    location 5-12
    rbw.config file entries  B-16
    use of 9-4
MAXSPILLSIZE
  determining size 4-35
  for offline load 4-37
  rbw.config file entries  B-16
  modifying 8-35
  syntax 9-7
THRESHOLD
  determining value 4-35
  rbw.config file entries  B-16
  syntax  9-8
  value 9-8
  use in offline load operations 4-37
indexes
  additional, when to create  4-3 to 4-10
B-TREE
  defined 2-4
  size estimates  4-24
creation guidelines  5-12 to 5-14
dropping  8-41
estimating sizes  4-21 to 4-24
growth, monitoring  8-4
partial availability  9-18
segmenting column  8-13
selection  9-18
selection with optical segments  8-22
size  4-21
STAR
  creating  5-14
  defined  2-4
  examples  4-5 to 4-7
  fill factor, changing  8-10
  invalid, cause  4-44
  key order  4-4
  multiple  2-4
  performance with  9-24
  size estimates  4-23
  size without MAXROWS PER SEGMENT  4-44
  use of MAXROWS PER SEGMENT parameter  5-9
  use of MAXSEGMENTS parameter  5-9
  with simple star schema  3-6
TARGET
  choosing domain size  4-8
  defined  2-4
  domains, defined  4-8
  hybrid  4-9, 4-16
  overview  5-14
  performance  5-14
  selectivity  4-8
  size estimates  4-24
  temporary space requirements  4-38
  use with TARGETjoin  4-11 to 4-17
  verifying creation  5-13
  when to build  5-13
indirect role membership
  defined  6-8
  example  6-17, 6-21
  usage  6-14
INFO_MESSAGE_LIMIT parameter,
  rbw.config file entries  B-18
initialization files, described  2-16
insert operator  9-33
installation, defining locale during  2-24
interdimensional OR queries  9-57
INTERVAL parameter
  modifying  8-34
  rbw.config file entry  B-11
  usage  8-37
invalid STAR indexes  4-44

J
  job accounting  7-31
  joins
    algorithms  9-25
    between fact tables, indexes for  4-4
    foreign key references  2-4
    relating tables  4-44

K
  key order in STAR indexes  4-4
  keywords, in syntax diagrams  xxiv

L
  language
    defined  2-21
    effect of changing  2-26, 2-29
  lexical character comparisons  2-24
  LICENSE_KEY entries in rbw.config  B-12
  licensed options
    enabling  1-8, 8-34
    rbw.config file entries  B-12
  limits, warehouse database  1-13
  linguistic character comparisons  2-24
  LOAD DATA statement examples, Aroma database  A-14 to A-21
  loading data
    examples, Aroma database  A-14 to A-22
    time-cyclic data, example  D-1 to D-19
  LOCALE parameter
    modifying  8-34
    rbw.config file entry  B-11
locales 2-21 to 2-30
character set component 2-22
client tool 2-26
client/server environment 2-26
components of 2-21
database 2-24
default 2-24, 2-27
defined 2-21
defining during installation 2-24
elements of 2-21
language component 2-21
overriding warehouse 2-26
sort component 2-23
system table references 2-25
territory component 2-22
warehouse 2-24
LOCK command, described 2-34
LOCK_DATABASE task
authorization 6-9
locked user accounts 6-32
locking
types of locks 2-33
wait behavior, changing 8-3
when to lock tables and databases 8-2
lock-out period, specifying 6-33
LOG_MAXSIZE configuration parameter 7-19
log daemon, See rbwlogd daemon
log files
Advisor 7-24
naming conventions 7-19, 7-24
removing 7-20
specifying location 7-20
specifying size 7-21
switching 7-22
types 8-38
log viewer
syntax 7-15
usage 7-16
LOG_AUDIT_LEVEL configuration parameter
described 7-21
rbw.config file entry B-20
LOG_DIRECTORY configuration parameter
described 7-20
rbw.config file entry B-20
LOG_ERROR_LEVEL configuration parameter
described 7-21
rbw.config file entry B-20
LOG_MAXSIZE configuration parameter
described 7-21
rbw.config file entry B-20
LOG_OPERATIONAL_LEVEL configuration parameter
described 7-21
rbw.config file entry B-20
LOG_SCHEMA_LEVEL configuration parameter
described 7-21
rbw.config file entry B-20
LOG_USAGE_LEVEL configuration parameter
described 7-21
rbw.config file entry B-20
LOGGING configuration parameter
described 7-20
rbw.config file entry B-20
logging queries 8-38
logging, Advisor See Advisor logging
logging, See event logging
logical database names
defining 5-4
described 2-12
rbw.config file entries B-22
logical I/O statistics 7-6
M
macros
creation and use 5-17 to 5-20
dropping 8-41
examples 5-19
EXPAND command 5-19
resolving references 5-18
scope 5-18
many-to-many relationship
defined by fact table 3-9
defined by multi-column foreign key 3-10
Index

many-to-one relationship, foreign keys 3-7
MAPFILE parameter
  changing 8-34
  rbw.config file entry B-10
MAX_SERVERS parameter
  changing 8-34
  rbw.config file entry B-10
MAXROWS_PER_SEGMENT parameter
  changing value 8-26
  effect on STAR indexes 4-23
  forcing REORG operation 5-9
  usage 5-9
  value in RBW_TABLES C-20
MAXSEGMENTS parameter
  changing value 8-26
  effect on STAR indexes 4-23
  forcing REORG operation 5-9
  usage 5-9
  value in RBW_TABLES C-20
MAXSIZE value, in RBW_TABLES table 8-5
MAXSIZE_ROWS value, in RBW_TABLES table C-20
MAXSPILLSIZE value
  for INDEX_TEMPSPACE 4-35
  for QUERY_TEMPSPACE 4-40
  syntax 9-8
MAXSPILLSIZE, INDEX_TEMPSPACE, See INDEX_TEMPSPACE parameters
  membership, roles 6-8
memory use
  loading and index requirements 4-34 to 4-40
  parallel queries 10-27
  merge phase of optimized index
    build 4-33
merge sort operator 9-33
message files 2-29
message templates 7-14
MESSAGE_DIR parameter
  modifying 8-34
  rbw.config file entry B-11
messages
  displayed in appropriate language 2-29
  internal, not translated 2-30
metavariabes, in syntax diagrams xxiv
Microsoft Access, database access
  with 6-3
MIGRATE TO clause, ALTER SEGMENT command 8-13
MODIFY_ANY task authorization 6-9
monitoring database activity 7-5
multibyte character sets 2-23
multiple databases, default segment location 5-8
multiple STAR indexes, See indexes
multi-star schema 3-11
N
  naive 1-1 match operator 9-33
NLS_LOCALE entries in rbw.config B-11
NLS_LOCALE LOCALE parameter 2-24
NLS_LOCALE MESSAGE_DIR parameter 2-29
NO ACTION keyword
  described 2-36
  usage 5-10
NO WAIT on locks
  described 2-34
  server 8-3
notation conventions xxi
NULL values, in ORDER BY clauses 2-17
O
object privileges
  defined 6-7
  granting 6-7, 6-13
  overview 2-32
ODBC client application requirements 6-3
ODBC Driver, Red Brick, described 1-5
.odbc.ini file, processing of 2-18
offline load operations
  example D-16
  space-planning 4-37
OFFLINE_LOAD task authorization 6-10
online transaction-processing databases (OLTP) 3-2
operational events 7-18
operators
  bit vector sort  9-29
  B-TREE 1-1 match  9-28
  B-TREE scan  9-29
  choose plan  9-30
  defined  9-28
  delete  9-30
  delete cascade  9-31
  delete refcheck  9-31
  exchange  9-31
  execute  9-32
  functional join  9-32
  general purpose  9-32
  hash 1-1 match  9-32
  hash AVL aggregate  9-33
  insert  9-33
  merge sort  9-33
  naive 1-1 match  9-33
  RISQL calculate  9-34
  simple merge  9-34
  sort 1-1 match  9-34
  STARjoin  9-34
  subquery  9-35
  table scan  9-35
  TARGET scan  9-36
  TARGETjoin  9-35
  update  9-36
  virtual table scan  9-36
optical segments
  defined  8-19
  index selection  8-22
optical storage support  8-19 to 8-22
OPTICAL_AVAILABILITY parameter
  rbw.config file entry  B-19
  syntax and usage  8-20
OPTION ADVISOR_LOGGING parameter
  described  7-28
  modifying  8-35
  rbw.config file entry  B-13
OPTION parameters
  ADVISOR_LOGGING  B-13
  ALLOW_POSSIBLE_DEADLOCK  B-11
  ARITHABORT  B-11
  AUTOROWGEN  B-11
  COUNT_RESULT  B-11
  CROSS_JOIN  B-11
  IGNORE_OPTICAL_INDEXES  8-23
  in rbw.config file  B-11
  OPTICAL_AVAILABILITY  8-20
options, licensed
  enabling  1-8, 8-39
  rbw.config file entries  B-12
OR versus UNION operation  9-57
outboard tables  3-10
out-of-memory errors, possible causes  4-35, 5-12
out-of-space errors, possible causes  4-35
outrigger tables  3-10
OVERRIDE FULLINDEXCHECK clause
  caution  D-12
  example  D-12
OVERRIDE REFCHECK clause
  caution  2-36
  usage  5-11
P
  parallel processing, overview  2-3
  parallel queries, allocating tasks for  10-19 to 10-20
  parallel query processing
    limiting processes  10-7
    minimum row requirements  10-10
    number of PSUs  4-43
    reducing disk contention  10-4
    system limitations  10-24
    tuning  10-1 to 10-34
  Parallel TMU, license, rbw.config file entry  B-12
  PARALLEL_HASHJOIN parameter
    enabling parallel hash joins  10-21
    rbw.config file entry  B-14
    parallelism on demand  10-2
    partial availability
      indexes  9-18
      tables  9-17
Index

PARTIAL_AVAILABILITY parameter
modifying 8-35
rbw.config file entry B-18
syntax and usage 9-17

PARTITIONED_PARALLEL_AGGREGATION parameter
described 10-2
rbw.config file entry B-14

PASSWORD parameters
CHANGE_MINIMUM_DAYS 6-29
COMPLEX_NUM_ALPHA 6-30
COMPLEX_NUM_NUMERICs 6-31
COMPLEX_NUM_PUNCTUATION 6-31
described 6-25
EXPIRATION_DAYS 6-26
EXPIRATION_WARNING_DAYS 6-27
LOCK_FAILED_ATTEMPTS 6-32
LOCK_PERIOD_HOURS 6-33
MINIMUM_LENGTH 6-30
modifying 8-35
rbw.config file entries B-21
RESTRICT_PREVIOUS 6-28

passwords
changes, enforcing 6-26
changing 6-4
changing system default A-6
complexity and length 6-30
expiration 6-26
expiration warning 6-27
frequency of changes 6-29
limiting reuse 6-28
locked accounts 6-32
new users 6-3
restrictions, standard 6-4
RISQL Entry Tool 5-6, 6-4
security 6-24 to 6-33
security parameters 6-25
standard restrictions 5-6
system account, changing 5-5

pathnames, relative and full 8-30
performance
general tuning 9-1 to 9-59
parallel query tuning 10-1 to 10-34
parallel table scan 10-32
resource and workload analysis 10-26 to 10-29
schema design 3-6
STARjoin queries 10-30
SuperScan technology 10-33
TARGET indexes 5-14
permission, See object privileges and databases, access to
physical I/O statistics 7-6
physical storage units, See PSUs C-17
precomputed views 2-11
PRECOMPUTED_VIEW_QUERY_REWRITE parameter
modifying 8-35
rbw.config file entries B-16
predicate, defined 9-29
primary key, defined 3-6
private macros 5-18
privileges, See object privileges
processes for parallel queries,
allocating 10-19 to 10-20
processes, monitoring 8-36
pseudocolumns 8-6
PSU sequence ID C-17
PSUs
allocating disk space 5-8
changing location 8-15
changing size 8-14
for indexes 5-8
MAXSIZE value 8-5
sequence ID C-17
PUBLIC access 6-7
public macro 5-18
PUBLIC_MACROS task
authorization 6-10

IX-12  Warehouse Administrator’s Guide for UNIX Platforms
Index

Q

queries
  parallel processes for 10-3 to 10-20
queries, logging 7-19, 8-38
query performance
  additional indexes 9-24
  SQL improvements 9-57
  See also performance and parallel query processing
query processing 9-25 to 9-39
QUERY_MEMORY_LIMIT parameter
  setting 9-11
  value 9-9
QUERY_TEMPSPACE parameters 9-2 to 9-12
  current values 9-10
  described 4-39 to 4-40
DIRECTORY(IES)
  rbw.config file entries B-16
  use of 4-9
MAXSPILLSIZE
  rbw.config file entries B-16
  value 4-40
  modifying 8-35
QUERY_MEMORY_LIMIT, rbw.config file entries B-16
  syntax 9-7
QUERYPROCS parameter
  definition 10-2
  described 10-3 to 10-19
  modifying 8-34
  rbw.config file entry B-13
  syntax and usage 10-7
quiescing a database 7-8

R

rb_cm utility 7-3
RB_CONFIG environment variable
  described 2-18
  user accounts 6-2
rb_creator script
  example A-5
  syntax and usage 5-3
RB_DEFAULT_IDX file 5-4
RB_DEFAULT_INDEXES file 5-4
RB_DEFAULT_LOADINFO file 5-4
RB_DEFAULT_LOCKS file 5-4
RB_DEFAULT_SEGMENTS file 5-4
RB_DEFAULT_TABLES file 5-4
rb_deleter script, syntax and usage 8-43
RB_DSN environment variable
  described 2-18
RB_HOST environment variable
  described 2-18
  user accounts 6-2
RB_NLS_LOCALE environment variable 2-27
RB_PATH environment variable
  described 2-18
  user accounts 6-2
rb_sample.cleanup script, usage 9-10
.rbretc file, processing of 2-17
rbw.config file
  described 2-16, B-1
  entries in B-10 to B-22
  example B-2 to B-9
  FORCE_FETCH_TASKS 10-3
  FORCE_JOIN_TASKS 10-3
  FORCE_SCAN_TASKS 10-3
  IGNORE_OPTICAL_INDEXES 8-23
  INDEX_TEMPSPACE_DIRECTORY 9-7
  INDEX_TEMPSPACE_MAXSPILLSIZE 9-7
  INDEX_TEMPSPACE_THRESHOLD 9-7
  modifying 8-34
  NLS_LOCALE LOCALE parameter 2-24
  NLS_LOCALE MESSAGE_DIR parameter 2-29
  OPTICAL_AVAILABILITY 8-20
  parameter summary B-23
  QUERY_MEMORY_LIMIT 9-7
  QUERY_TEMPSPACE_DIRECTORY 9-7
  QUERY_TEMPSPACE_MAXSPILLSIZE 9-7
  rbw.findserver utility 8-36
  rbw.servermon daemon
  and Findserver 8-37
  interval setting B-11
  rbw.show script, usage 8-36
  rbw.start script
  rbwadmd daemon 7-11
  usage 8-36
rbw.stop script, usage 8-36
RBW_COLUMNS table C-4
RBW_CONSTRAINT_COLUMNS table C-5
RBW_CONSTRAINTS table 5-10, C-6
RBW_HIERARCHIES table C-6
RBW_INDEXCOLUMN table C-7
RBW_INDEXES table C-7
RBW_LOADINFO table C-8
RBW_LOADINFO_LIMIT parameter, rbw.config file entries B-18
RBW_MACROS table C-9
RBW_OPTIONS table 2-25, C-10
RBW_PRECOMPVIEW_CANDIDATES table C-3
RBW_PRECOMPVIEW_UTILIZATION table C-3
RBW_PRECOMPVIEWCOLUMNS table C-11
RBW_RELATIONSHIPS table 5-10, C-12
RBW_ROLE_MEMBERS table 6-20, C-13
RBW_ROLES table 6-20, C-14
RBW_ROWNUM pseudocolumn 8-6, 8-12
RBW_SEGID pseudocolumn 8-7
RBW_SEGMENTS table C-15
RBW_SEGNAME pseudocolumn 8-6, 8-7
RBW_STORAGE table C-17
RBW_SYNONYMS table C-18
RBW_TABAUTH table 6-20, C-19
RBW_TABLES table C-20
RBW_USERAUTH table 6-20, C-21
RBW_VIEWS table C-25
RBW_VIEWTEXT table C-26
rbwadmd daemon 2-30
description 1-7, 7-11
starting 7-11
statistic collection interval 7-11
stopping 7-11
RBWAPI entries in rbw.config B-10
rbwapid daemon
description 1-7
rbw.config file entries B-10
.rbwerr file, defined 2-17
rbwlogd daemon 2-30
description 7-13
role in accounting 7-32
starting 7-13
stopping 7-23
rbwlogview executable 7-14
.rbwrc file, usage 2-16
rbwsvr process 1-7
read locks 2-33
recovering a damaged segment 8-17
Red Brick Vista option
Advisor logging commands 7-24
described 2-11
rbw.config license entry B-12
Red Brick Warehouse
component overview 1-3 to 1-5
described 1-1
installation 2-24
license, rbw.config file entry B-12
process overview 1-6 to 1-7
utilities 2-25
Red Brick Warehouse Administrator tool
described 2-19
warehouse component 1-5
Red Brick Warehouse for Workgroups
license, rbw.config file entry B-12
limits 1-13
redbrick
directory permissions 5-3
user account 2-31
user umask setting 5-3
redbrick user account 2-31
referenced tables
defined 3-6
outboard, outrigger 3-10
referencing tables 3-6
referential integrity
defined 2-35
delete behavior, defining 5-10 to 5-11
delete operations 2-35 to 2-41
load and insert operations 2-35
ON DELETE clause 5-10
relation scans 10-2, 10-18
RENAME clause, ALTER SEGMENT command 8-11
RENICE_COMMAND parameter
rbw.config file entry B-21
user priorities 7-10
REORG operations
  after ALTER SEGMENT 4-45
  effect of MAXROWS PER SEGMENT parameter 5-9
  effect of MAXSEGMENTS parameter 5-9
REORG_ANY task authorization 6-10
REPORT_INTERVAL parameter
  rbw.config file entry B-21
  usage 7-12
resetting statistics 7-9
resetting TEMPSPACE parameters 9-8
resource privileges, with ODBC 6-3
RESOURCE system role
  described 2-32, 6-5
  granting 6-6, 6-14
  revoking 6-6, 6-18
restore operations, during index creation 5-12
RESTORE_DATABASE task
  authorization 6-10
restricted delete operations 5-10
RESULT BUFFER FULL ACTION parameter
  described 9-14
  modifying 8-35
  rbw.config file entry B-15
RESULT BUFFER parameter
  described 9-13
  modifying 8-35
  rbw.config file entry B-14
re-use by row 8-6
REVOKE authorization and role command examples 6-18
usage 6-18
RISQL calculate operator 9-34
RISQL Entry Tool
  described 1-5
  invoking A-6
  user accounts 6-2
RISQL Entry Tool and RISQL Reporter locales for 2-26
RISQL extensions, described 1-2
RISQL Reporter
  described 1-5
  license, rbw.config file entry B-12
  user accounts 6-2
ROLE_MANAGEMENT task
  authorization 6-10
role-based security, described 6-8
roles
  direct and indirect membership 6-8
  dropping 8-41
  granting 6-6, 6-14 to 6-17
  membership 6-14 to 6-17
  revoking 6-6, 6-18
  system 6-5
  system table information 6-20
  user-created 6-8 to 6-23
rollback operation 8-2
row IDs 8-12
row numbers 8-12
ROWCOUNT parameter, rbw.config file entries B-17
rows, deleting
  locking tables 8-2
  referential integrity 2-35 to 2-41
ROWS_PER_FETCH_TASK parameter
  described 10-2, 10-10
  modifying 8-35
  rbw.config file entry B-13
  syntax and usage 10-13
ROWS_PER_JOIN_TASK parameter
  described 10-2, 10-10
  modifying 8-35
  rbw.config file entry B-13
  syntax and usage 10-13
ROWS_PER_SCAN_TASK parameter
  described 10-2, 10-10
  modifying 8-35
  rbw.config file entry B-13
  syntax and usage 10-11
S
schema events 7-18
schemas, See star schemas
scratch space, estimating
  requirements 4-33 to 4-40
secondary dimension tables 3-10
segmented storage
  default locations 9-15
  described 2-5
  planning for use 4-41 to 4-45
See also segments
Index

segments
  adding a new segment  D-14
  altering  8-11 to 8-16
  attaching, detaching  8-12
  automatic locking  2-33
  creation guidelines  5-8
  damaged, recovering  8-17 to 8-18
  default
    changing to named  4-42
    defined  2-6
    defined  2-5
  determining name  2-15
  dropping  8-42
  moving entire  8-13
  named  2-5
  named versus default  4-41
  offline load, example  D-16
  online, offline  2-9, 8-13
  optical, defined  8-19
  partial availability  2-10
  range, changing  8-13
  reusing  D-12, D-19
  rolling off  D-12
  running out of space  8-8
  space available  8-4 to 8-5
  time-cyclic data, example  D-1 to D-19
  verifying  8-15
SEGMENTS parameter
  modifying  8-35
  rbw.config file entry  B-18
  usage  9-16
selectivity, defined  4-8
server initialization file, See .rbwr file
SERVER parameter
  modifying  8-34
  rbw.config file entry  B-10
server processes, monitoring  8-36
server, warehouse
  described  1-4
  maximum number  B-10
SERVER_NAME parameter
  modifying  8-34
  rbw.config file entry  B-10
session, closing  7-10
SET commands, SQL
  ADVISOR LOGGING  7-28
  DEFAULT SEGMENT STORAGE
    PATH  9-15
  FORCE_FETCHTASKS  10-19
  FORCE_HASHJOIN_TASKS  10-21
  FORCE_JOIN_TASKS  10-19
  FORCE_SCAN_TASKS  10-18
  IGNORE_OPTICAL_INDEXES  8-23
  INDEX_TEMPSPACE  9-7
  OPTICAL_AVAILABILITY  8-21
  overview  2-18
  PARTIAL_AVAILABILITY  9-17
  QUERY_MEMORY_LIMIT  9-7
  QUERY_TEMPSPACE  9-7
  REPORT_INTERVAL  7-12
  RESULT_BUFFER  9-13
  RESULT BUFFER FULL ACTION  9-14
  SEGMENT  9-16
  STATS  9-1, 9-37
  UNIFORM PROBABILITY FOR
  ADVISOR  7-30
SHMEM parameter
  modifying  8-34
  rbw.config file entry  B-10
simple merge operator  9-34
size estimates
  B-TREE indexes  4-24
  example, database sizing  4-24 to 4-31
  STAR indexes  4-23
  system tables  4-29
  tables and indexes  4-18 to 4-24
  TARGET indexes  4-24
sizing tool for databases (dbsize)  4-20
SKU number  3-12
sort 1-1 match operator  9-34
sort component
  defined  2-23
  effect of changing  2-26
  sort phase of optimized index build  4-33
SQL command, canceling  7-9
SQL-BackTrack license, rbw.config file
  entry  B-12
STAR indexes, See indexes, STAR
star schemas 3-5 to 3-17
  described 3-5 to 3-13
  design considerations 3-14 to 3-15
  examples 3-18 to 3-26
  performance 3-6
  supported types 3-16 to 3-17
  use of attributes 3-20
STARjoin
  defined 2-4
  parallel query performance 10-30
STARjoin operator 9-34
statistic collection interval 7-11
statistics reporting 9-1
storage devices, optical 8-19
subquery in the FROM clause 9-57
subquery operator 9-35
SuperScan technology 2-3, 10-4, 10-33
support
  documentation xxvii
  technical xxv
synonyms, dropping 8-42
syntax diagrams
  conventions for xxii
  keywords in xxiv
  metavariables in xxiv
syntax notation xxi
system accounts, for new users 6-2
system catalog, contents C-2
system roles
  described 6-5
  granting 6-6, 6-14
  overview 2-32
  revoking 6-6, 6-18
system table references to locale 2-25
system tables
  described C-1 to C-26
  size estimates 4-29
system, database user account 5-5
T
  table scan operator 9-35
tables
  access, limiting with views 5-16
  adding rows, effect on STAR index 4-44
  altering 8-24 to 8-28
  browsing 3-19
  creating Aroma tables A-7
  creation guidelines
    foreign key order 5-9
    order 5-9
    referential integrity on delete 5-9
    setting MAXROWS PER SEGMENT 5-9
    setting MAXSEGMENTS 5-9
  dropping 8-42
  families, immediate and complete 2-36
  growth patterns 4-18, 4-44
  growth, monitoring 8-4
  in segments 5-8
  multiple fact tables 3-8
  partial availability 9-17
  segmenting column 8-13
  size estimates 4-20
  system tables, size estimates 4-29
  when to lock 8-2
  tables, optical storage 8-20
TARGET indexes, See indexes, TARGET
TARGET scan operator 9-36
TARGETjoin
  defined 2-4
  planning for 4-11 to 4-17
  query processing 9-40 to 9-53
  rules 9-41
  when to use 9-42
TARGETjoin operator 9-35
  task authorizations
    ACCESS_SYSINFO 7-4
    ALTER_SYSTEM 7-4
    defined 6-9
    granting 6-12
    tasks for parallel queries,
      allocating 10-16 to 10-20
    technical support xxv
    TEMP_RESOURCE task
      authorization 6-10
temporary macros
  deleting 8-41
  scope 5-18
temporary space 4-32 to 4-40, 9-2 to 9-12
temporary tables 4-20
TEMPORARY_DATA_SEGMENT parameter
  modifying 8-35
  rbw.config file entries B-17
TEMPORARY_INDEX_SEGMENT parameter
  modifying 8-35
  rbw.config file entries B-17
territory
  defined 2-22
  effect of changing 2-26, 2-29
THRESHOLD, INDEX_TEMPSPACE,
  See INDEX_TEMPSPACE parameters
time-cyclic data, example D-1 to D-19
TMU, Aroma examples A-20 to A-22
TMU_BUFFERS parameter
  modifying 8-35
  rbw.config file entry B-13
TMU_CONVERSION_TASKS parameter
  modifying 8-35
  rbw.config file entry B-15
TMU_INDEX_TASKS parameter
  modifying 8-35
  rbw.config file entry B-15
TMU_OPTIMIZE parameter
  modifying 8-35
  rbw.config file entries B-15
TMU_SERIAL_MODE parameter
  modifying 8-35
  rbw.config file entry B-15
TOTALFREE column, RBW_SEGMENTS table 8-6
TOTALQUERYPROCS parameter
  definition 10-2
  described 10-3 to 10-19
  modifying 8-34
  rbw.config file entry B-14
  syntax and usage 10-7
transaction-processing databases 3-2
troubleshooting, general problems xxvii
TUNE parameters
  FORCE_AGGREGATION_TASKS 10-22
  FORCE_FETCH_TASKS 10-16
  FORCE_JOIN_TASKS 10-16
  FORCE_SCAN_TASKS 10-16
  INDEX_TEMPSPACE_DIRECTORY 9-7
  INDEX_TEMPSPACE_MAXSPILLSIZE 9-7
  INDEX_TEMPSPACE_THRESHOLD 9-7
  OPTICAL_AVAILABILITY 8-21
  PARTITIONED_PARALLEL_AGGREGATION 10-22
  QUERY_MEMORY_LIMIT 9-7
  QUERY_TEMPSPACE_DIRECTORY 9-7
  QUERY_TEMPSPACE_MAXSPILLSIZE 9-7
  tuning parameters, parallel query 10-2
U
UNION versus OR operation 9-57
UPC level 3-18
update operator 9-36
UPGRADE_DATABASE task
  authorization 6-10
  usage events 7-19
USAGE ROUTINE event 7-19, 8-38
USE_INVALID_PRECOMPUTED_VIEWS parameter
  modifying 8-35
  rbw.config file entries B-17
USED column, RBW_STORAGE table 8-6
user accounts
  adding new users 6-2 to 6-4
  expired 6-26
  locked 6-32
  RISQL Entry Tool 6-2
  user priority, changing 7-10
USER, SQL variable 5-16
USER_MANAGEMENT task
  authorization 6-10
user-created roles
  creating 6-12
  direct and indirect membership 6-8
dropping 8-41
  granting 6-14 to 6-17
  managing 6-8 to 6-23
  revoking 6-18
users
  adding to a database 6-3
  changing passwords 6-4
utility programs, Red Brick Warehouse 2-25

V
version of warehouse software 8-39
views
  creation guidelines 5-15 to 5-16
  described 3-13
dropping 8-43
  precomputed 2-11
virtual table scan operator 9-36
Visual Basic, database access with 6-3

W
WAIT/NO WAIT on locks
  described 2-34
  server 8-3
warehouse administrator, defined xvii
Warehouse for Workgroups, See Red Brick Warehouse for Workgroups
warehouse locale
  defined 2-24
  defining during installation 2-24
  overriding 2-26
warehouse processes
  monitoring 8-36
  overview 1-6
weak selectivity
  defined 4-8
  improving performance for queries with 5-14
workload accounting 7-31
World Wide Web address, for Red Brick Systems xxv
write locks 2-33
<table>
<thead>
<tr>
<th>Region</th>
<th>Address</th>
<th>Phone</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>USA SALES OFFICES</strong></td>
<td>1040 Crowne Point Parkway, Suite 250, Atlanta, GA 30338 +1 770 804 2440</td>
<td>+1 770 804 2440</td>
<td>USA</td>
</tr>
<tr>
<td></td>
<td>2215 York Road, Suite 409, Oak Brook, IL 60521 +1 630 472 8500</td>
<td>+1 630 472 8500</td>
<td>USA</td>
</tr>
<tr>
<td></td>
<td>1120 Avenue of the Americas, 4th Floor, New York, NY 10036 +1 212 626 6815</td>
<td>+1 212 626 6815</td>
<td>USA</td>
</tr>
<tr>
<td></td>
<td>5314 Arapaho Road, Dallas, TX 75248 +1 972 702 1750</td>
<td>+1 972 702 1750</td>
<td>USA</td>
</tr>
<tr>
<td></td>
<td>2010 Corporate Ridge, 7th Floor, McLean, VA 22102 +1 703 883 9310</td>
<td>+1 703 883 9310</td>
<td>USA</td>
</tr>
<tr>
<td><strong>UK SALES OFFICE</strong></td>
<td>Red Brick Systems UK Ltd., 45 Berkeley Square, Mayfair, London W1A 1EB</td>
<td>+44 171 290 8373</td>
<td>United Kingdom</td>
</tr>
<tr>
<td><strong>AUSTRALASIA HEADQUARTERS</strong></td>
<td>Red Brick Systems Australasia Pty. Ltd., Level 20, 99 Walker Street, North Sydney, NSW 2060 Australia</td>
<td>+61 02 9911 7744</td>
<td>Australia</td>
</tr>
<tr>
<td><strong>JAPAN HEADQUARTERS</strong></td>
<td>Red Brick Japan Co. Ltd., Level 16 Shiroyama Hills, 4-3-1 Toranomon, Minato-ku, Tokyo 105 Japan</td>
<td>+81 3 5403 4638</td>
<td>Japan</td>
</tr>
</tbody>
</table>