

RED BRICK® WAREHOUSE
Version 5.1

RED BRICK VISTA™
USER'S GUIDE

5.1

The information in this document is subject to change without notice and does not represent a commitment by Red Brick Systems. The software and/or databases described in this document are furnished under a license agreement and can be used or copied only in accordance with the terms of the agreement. Except as permitted by such license, no part of this document and/or database may be reproduced or transmitted in any form or by any means, electronic or mechanical (including photocopying and recording), or transferred to information storage and retrieval systems without the written permission of Red Brick Systems.

Restricted Rights: Use, duplication, or disclosure by the United States Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software—Restricted Rights at 48 CFR 52.227-19, as applicable.

© Copyright 1991–1998 Red Brick Systems, Inc.
All rights reserved.
Printed in the USA.

RISQL, Red Brick, the Red Brick logo ( RED BRICK®), and The Data Warehouse Company are registered trademarks of Red Brick Systems, Inc.
Red Brick Data Mine, Red Brick Data Mine Builder, Red Brick Vista, STARindex, STARjoin, TARGETindex, and TARGETjoin are trademarks of Red Brick Systems, Inc.

All other trademarks are the properties of their respective companies.

Revision number: 1
January, 1998
Part number: 502050

Red Brick Systems, Inc.
485 Alberto Way
Los Gatos, California 95032
USA
Telephone: +1 408 399 3200
+1 800 777 2585



Contents

About This Document

- Purpose vii
- Audience vii
- Organization viii
- Related Documentation ix
- Conventions xi
 - Syntax Notation xi
 - Syntax Diagrams xii
 - Keywords and Punctuation xiv
 - Identifiers and Names xiv
- Customer Support xv
 - Support Solutions Warehouse xv
 - General and Technical Questions xv
 - Troubleshooting Tips xvii
 - Documentation Questions and Comments xvii

1 *Introduction to Red Brick Vista*

- Aggregations in the Data Warehouse 1-2
- Aggregate Query Performance 1-2
- The Query Rewrite System 1-3
 - How Aggregate Queries Are Rewritten 1-3
- The Advisor 1-5
- Summary 1-6

2 *Key Concepts of Query Rewriting*

- Precomputed Views 2-2
 - Aggregate Tables 2-3
 - Aggregate Query Rewrites 2-3
- Rollups and Hierarchies 2-6
 - Functional Dependencies 2-6
 - Derived Dimensions 2-9

3 Using the Query Rewrite System

- Creating Aggregate Tables 3-2
 - Populating Aggregate Tables 3-2
 - Example of an Aggregate Table 3-3
- Creating Precomputed Views 3-5
 - CREATE VIEW...USING Command 3-5
 - Example View Definition 3-7
 - Cost-Based Analysis of Precomputed Views 3-8
- Using Hierarchies 3-11
 - Explicit Hierarchies 3-11
 - Implicit Hierarchies 3-15
- Optimizing Query Rewrites 3-17
 - Creating Derived Dimensions 3-17
 - Creating Indexes 3-22
- Setting Up the Query-Rewriting Environment 3-23
 - Marking Precomputed Views Valid 3-23
 - Turning On the Query Rewrite System 3-24
 - Generating Statistics 3-24
 - Querying the RBW_VIEWS System Table 3-25
 - Making Precomputed Views Invisible to Client Tools 3-27
- Checklist of Query-Rewriting Tasks 3-28

4 Query Rewrite Case Studies

- General Instructions 4-2
- Case 1—Rewriting a STARjoin Query 4-3
- Case 2—Making Use of Explicit Hierarchies 4-8
- Case 3—Optimizing Query Rewrites with Derived Dimensions 4-11
- Case 4—Using Implicit Hierarchies to Rewrite Queries 4-16
- Case 5—Rewriting Subqueries 4-20
- Case 6—Rewriting a Query That Calculates Averages 4-23

5 *Using the Advisor*

- Advisor Overview 5-2
 - Analysis of Query Patterns 5-2
 - Advisor System Tables 5-2
 - Advisor Log Files 5-3
- Configuring the Advisor Logging System 5-3
 - Creating the Advisor Log Files 5-3
 - Logging Queries 5-4
 - Setting the ACCESS_ADVISOR_INFO Task Authorization 5-7
 - Defining Valid Hierarchies 5-8
- Querying the Advisor 5-8
 - Inserting the Results of an Advisor Query Into a Table 5-8
 - Querying the RBW_PRECOMPVIEW_UTILIZATION Table 5-10
 - Querying the RBW_PRECOMPVIEW_CANDIDATES Table 5-12
- Interpreting the Results of Advisor Queries 5-17
 - BENEFIT Column 5-17
 - SIZE and REDUCTION_FACTOR Columns 5-17
 - REFERENCE_COUNT Column 5-18
 - Combining the Results 5-18
- Understanding the BENEFIT Column 5-19
 - How the BENEFIT Column Is Calculated 5-19
 - What the Numbers Mean 5-20
 - Uniform Probability 5-21
- Advisor System Table Column Descriptions 5-24
 - RBW_PRECOMPVIEW_CANDIDATES Table 5-24
 - RBW_PRECOMPVIEW_UTILIZATION Table 5-26
- Checklist of Advisor Tasks 5-28

A *Glossary*

Index

Contents





About This Document

Purpose

This document explains how to use the Red Brick Vista™ option, an integrated aggregate navigation and advice system that improves query performance by rewriting queries against detail data to use precomputed aggregate data.

Audience

The primary audience is the warehouse administrator who wants to improve query performance by using an efficient aggregation strategy. A secondary audience is the application developer or consultant who needs to know how to design schemas and write queries that will make the best use of the Red Brick Vista option.

A working knowledge of Red Brick® Warehouse products is assumed, as well as familiarity with the Structured Query Language (SQL).

Organization

This document introduces the concepts behind the design of the Red Brick Vista option, then explains how to use its two main components—the query rewrite system and the Advisor:

- Chapter 1, “Introduction to Red Brick Vista,” discusses the problems warehouse administrators face in managing aggregate tables and introduces the Red Brick Vista option as a solution.
- Chapter 2, “Key Concepts of Query Rewriting,” explains the concepts behind the query rewrite system that you need to understand before using the Red Brick Vista option.
- Chapter 3, “Using the Query Rewrite System,” explains how to rewrite aggregate queries by creating precomputed views linked to aggregate tables.
- Chapter 4, “Query Rewrite Case Studies,” concentrates on several examples of query rewriting that use the tables in the sample Aroma database.
- Chapter 5, “Using the Advisor,” explains how to query the Advisor system tables for advice about the usefulness of both existing precomputed views and new candidate views.
- Appendix A, “Glossary,” defines the key terms used throughout the document.

Complete syntax descriptions of the SQL commands introduced in this document are presented in the *SQL Reference Guide*.

Related Documentation

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

<i>Installation and Configuration Guide</i>	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.
<i>Warehouse Administrator's Guide</i>	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 (<i>rbw.config</i>). Customized for either UNIX-based or Windows NT systems.
<i>Table Management Utility Reference Guide</i>	Description of the Table Management Utility, including all activities related to loading and maintaining data. Also includes information about data replication and the <i>rb_cm</i> copy management utility.
<i>SQL Reference Guide</i>	Complete language reference for the Red Brick Systems SQL implementation and RISQL [®] extensions for warehouse databases.
<i>SQL Self-Study Guide</i>	Example-based review of SQL and introduction to the RISQL extensions, the macro facility, and Aroma, the sample database.
<i>RISQL Entry Tool and RISQL Reporter User's Guide</i>	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.
<i>Messages and Codes Reference Guide</i>	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.
<i>Release Notes</i>	Information pertinent to the current release that was unavailable when the documents were printed.

About This Document
Related Documentation

In addition to the standard documentation set, the following documents are included for specific sites:

<i>Red Brick Vista User's Guide</i>	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.
<i>SQL-BackTrack for Red Brick Warehouse User's Guide</i>	The 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.
<i>Client Connector Pack Installation Guide</i>	Procedures for installing and configuring 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.
<i>ODBC Connectivity Guide</i>	Information about ODBC conformance levels as well as instructions for compiling and linking an ODBC application using the Red Brick ODBClib SDK.
<i>Red Brick Data Mine User's Guide</i>	Description of the data mining process, and procedural information for using the Red Brick Data Mine™ 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.
<i>Red Brick Data Mine Builder™ User's Guide</i>	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).







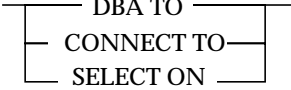

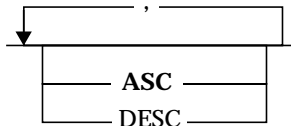
Syntax Notation

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

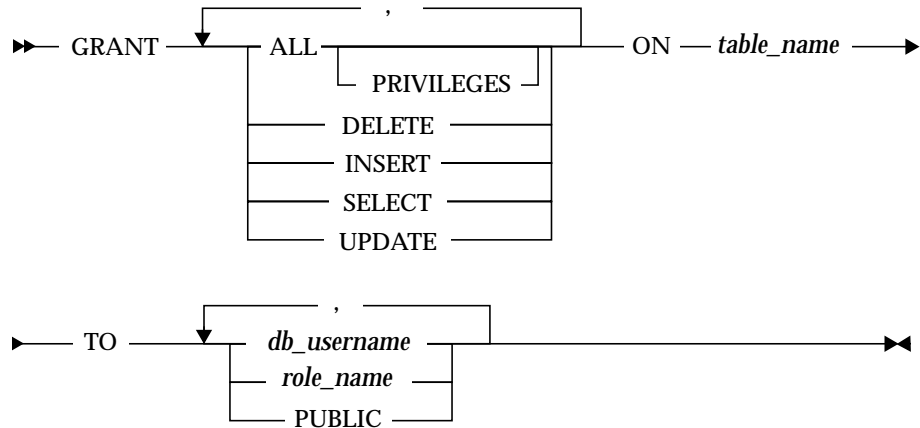
Command Element	Example	Convention
Values and parameters	<i>table_name</i>	Items that you replace with an appropriate name, value, or expression are in <i>italic</i> type style.
Optional items	[]	Optional items are enclosed by square brackets. Do not type the brackets.
Choices	ONE TWO	Choices are separated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

Syntax Diagrams

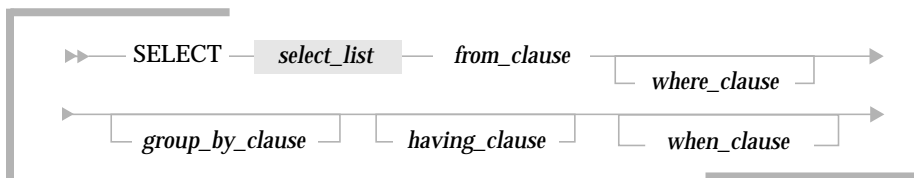
This guide uses diagrams built with the following components to describe the syntax for statements and all commands other than system-level commands:

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

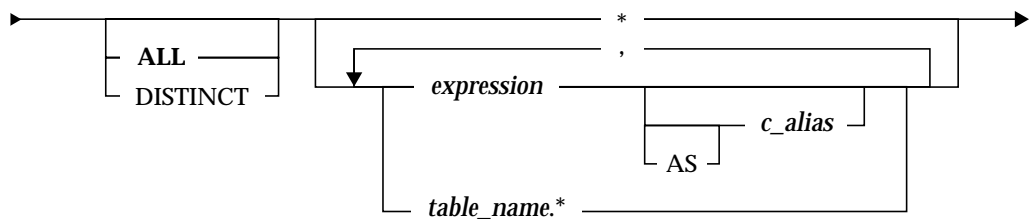
The syntax elements shown above are combined to form a diagram as follows:



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



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



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



1

Introduction to Red Brick Vista

This chapter introduces the Red Brick Vista™ option as a tool for optimizing aggregate query performance. The following sections are included:

- Aggregations in the Data Warehouse
- Aggregate Query Performance
- The Query Rewrite System
- The Advisor
- Summary

Aggregations in the Data Warehouse

Decision-support queries often require aggregations—typically, sums of revenues or costs over periods of time, markets, or products. Such queries can be expensive in terms of both selecting the appropriate rows to sum and performing the aggregate calculation. Consequently, the quickest way to run an aggregation query is not to run it at all, but to precompute the aggregate totals and store them in separate tables.

Although this sounds like a simple solution to aggregate query performance, it causes administrators to create, load, and maintain a very large number of aggregate tables, and it requires application developers and users to manually rewrite their queries as new aggregate tables enter or leave the schema. Add to these problems the fact that aggregate tables usually require a lot of disk space and take a long time to load, and developing applications for high-performance aggregate queries becomes a very difficult task.

To help administrators guarantee good query performance, minimize the maintenance workload, and keep their users' applications simple and stable, an integrated solution for working with aggregate tables is required, not just a separate mechanism for precomputing the aggregate records.

Aggregate Query Performance

The Red Brick Vista option offers a systematic approach to precomputing aggregate data for decision-support queries. The key to this approach is that users always query the same set of database tables; however, before each query is executed, a cost-based analysis determines whether the query can be intercepted and rewritten to improve its performance. In addition, statistics about query execution are logged so that administrators can find out how efficiently the existing aggregation strategy is working, as well as how to improve that strategy.

The Red Brick Vista option consists of two logical components:

- The query rewrite system, or *aggregate navigator*, which intercepts and rewrites queries to use aggregate tables. Queries are rewritten transparently to client applications and end users.
- The Advisor—a query-logging and analysis facility that can be queried for advice on the size and relative benefits of both existing aggregate tables and new tables that would be useful to create.

The Query Rewrite System

The Red Brick Vista option is fully integrated with the RDBMS engine. Queries are intercepted and rewritten by the warehouse server, not by a separate piece of software. This is not the case with many commercial aggregate navigation systems, which function as middleware programs in between the RDBMS and the query tool. There are two advantages to the integrated approach:

1

- Aggregation information is stored in the system tables along with all the other metadata for the database, making knowledge of all database activity centralized. The result of this integration is consistency; for example, if the aggregate table data is stale, the system knows about it instantly and does not use the table to rewrite queries (unless requested to do so).
- Optimization strategies are known to the RDBMS but not necessarily known or heeded in the middleware case.

You create database objects for aggregate query rewriting by using a set of RISQL[®] extensions to standard SQL commands and some new commands specific to the Red Brick[®] Warehouse server. Everything that the administrator needs to do to make query rewriting possible can be done via the RISQL Entry Tool (or any other ODBC-compliant client tool) and the Table Management Utility (TMU).

How Aggregate Queries Are Rewritten

The query rewrite system relies on the existence of *precomputed views*. A precomputed view defines a query whose results are stored in an associated aggregate table. The administrator creates and loads the aggregate table (or uses an existing aggregate table), then defines the view with a query expression that reflects the exact contents of the table.

The administrator knows that the precomputed view exists, but the database users need not. When a query is submitted, the query rewrite system evaluates the precomputed views the administrator has created and, if possible, rewrites the query to select from aggregate tables that are much smaller than the tables in the original query.

Where possible, joins are simplified or removed, and depending on the degree of consolidation that occurs between the detail and aggregate data, query response times are highly accelerated. Moreover, queries can be rewritten against a precomputed view even when the query and the view definition do not match exactly.

For detailed information about the query rewrite system, refer to Chapters 2, 3, and 4:

- The concept of precomputed views is further explained in Chapter 2, “Key Concepts of Query Rewriting.”
- The process of creating database objects to make query rewriting possible is described in Chapter 3, “Using the Query Rewrite System.”
- Several detailed examples of rewritten queries are presented in Chapter 4, “Query Rewrite Case Studies.”

The Advisor

The Advisor is an integral part of the Red Brick Vista option. The Advisor provides a facility to log activity of aggregate queries against a database. From the logged queries, you can analyze the following:

- The use of existing aggregates in the database.
- Potential new aggregates to create that can improve query performance.

As the database is used, the Advisor logs queries that are rewritten and queries that would benefit from being rewritten if the appropriate aggregate table existed. After a period of time, the DBA uses the Advisor to analyze the aggregate query logs by querying one of the Advisor system tables:

- `RBW_PRECOMPVIEW_UTILIZATION` to analyze the use of existing precomputed views in the database.
- `RBW_PRECOMPVIEW_CANDIDATES` to view the optimal set of precomputed views that the Advisor suggests based on the query activity in the log and any existing precomputed views.

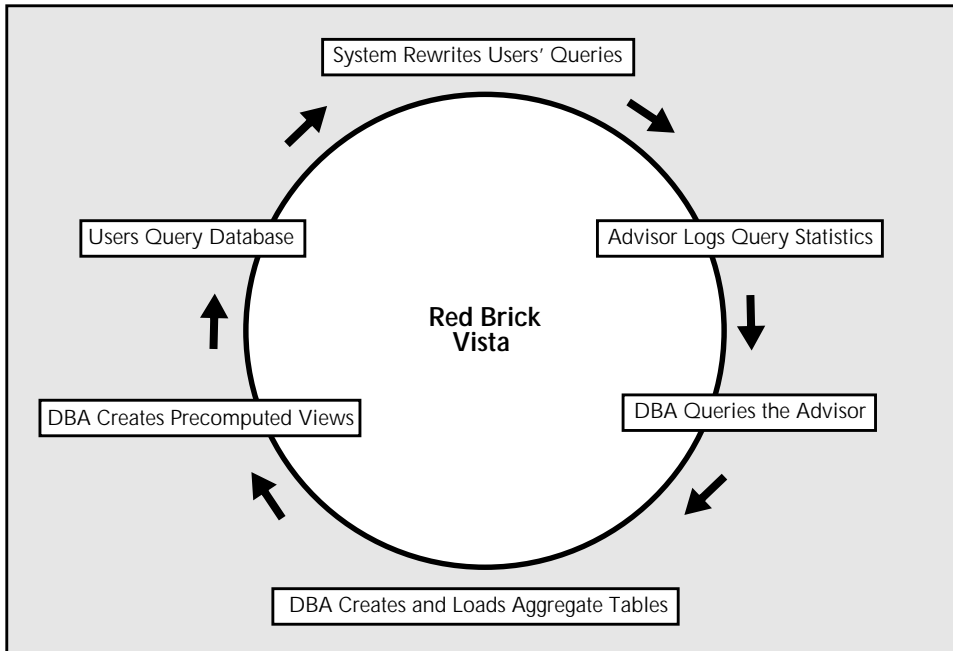
Queries against the Advisor system tables perform detailed analysis of the information stored in the logs. The analysis is based on sophisticated algorithms that determine the best set of aggregate tables for the actual query history. As part of the analysis, a benefit is assigned to each existing view and to each candidate view. The Benefit column in the Advisor system tables is a cost metric based on the number of rows saved by processing the query through the precomputed view instead of the detail table and the number of times the view has been used (for existing views) or the number of times the view would have been used (for candidate views).

For detailed information about configuring and using the Advisor, refer to Chapter 5, “Using the Advisor.”

Summary

Used in combination with other Red Brick Warehouse techniques for accelerating query performance—such as STARjoin™ and TARGETjoin™ processing—the Red Brick Vista option offers the warehouse administrator a flexible set of features for optimizing aggregation queries.

The process of getting advice, creating precomputing views, and using the query rewrite system is iterative, as shown in the following diagram. This combination of features allows the administrator to continuously improve the aggregation strategy while users query the same set of detail tables. Query performance improves but the user's view of the database schema does not change.



The following chapters expand on the concepts and tasks introduced in this chapter, using examples based on the Aroma database. This database is installed during the installation of the warehouse software and is described in detail in the *SQL Self-Study Guide*.

2

Key Concepts of Query Rewriting

Before using the Red Brick Vista option to rewrite aggregate queries, you need to understand several concepts behind the design of the query rewrite system. A clear understanding of these concepts is critical to using the system effectively and ensuring the integrity of rewritten query results.

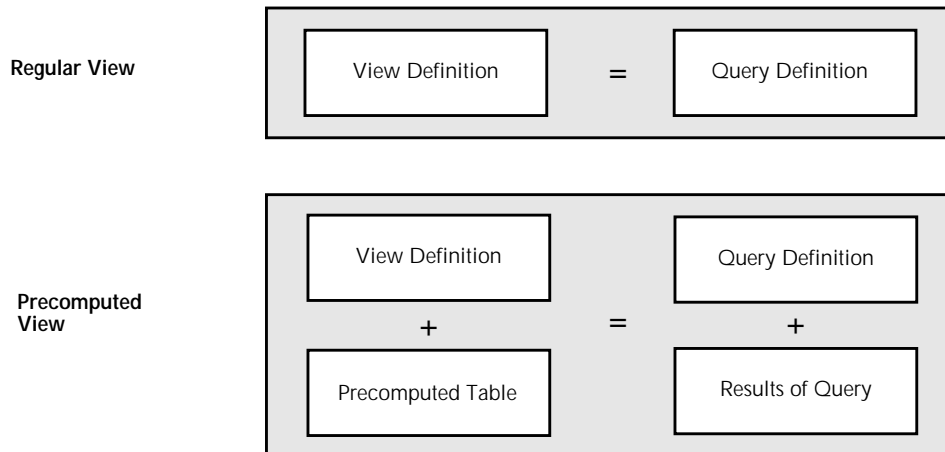
This chapter contains the following sections:

- Precomputed Views
- Rollups and Hierarchies

Precomputed Views

A *precomputed view* is a view that is linked to a database table known as a *precomputed table*. The view defines a query, and the table contains the precomputed results of the query.

In the sense that it defines a query, a precomputed view is similar to a regular view; however, a query in a regular view is not precomputed into a table. The results of a query in a regular view must be computed every time the view is referenced, whereas a query in a precomputed view is already computed and its results stored in the associated precomputed table.



A query defined in a precomputed view is not precomputed automatically. The database administrator must populate the table linked to the view with either a TMU LOAD DATA operation or an SQL INSERT INTO...SELECT statement. In this way, existing aggregate tables (as well as new aggregate tables) can be integrated into an application that uses the Red Brick Vista option.

A view can be said to be precomputed only when both of the following statements are true:

1. It is linked to a table.
2. The linked table is populated with data.

Furthermore, the Red Brick Vista option does not *validate* the data used to populate the precomputed table. *The administrator must ensure that the table contains the exact result set that would be returned by the query defined in the view.* If the administrator fails to validate the data in this way, rewritten queries will return incorrect results.

Aggregate Tables

An *aggregate table* is a special case of a precomputed table: it is a database table that stores the results of an aggregate query defined in an associated precomputed view.

Typically, an aggregate query uses a standard SQL function such as SUM or COUNT to aggregate factual data such as sales totals over given periods of time. Other aggregation queries use a GROUP BY clause to select distinct rows of dimensional data from a large dimension table, such as distinct combinations of quarters and years or districts and regions. In these ways, aggregation queries *roll up* rows of data that have a fine granularity into groups of rows that have a coarser granularity.

Throughout this document, the term *aggregate table* is used consistently to refer to a table associated with a precomputed view. This terminology is used because in the Red Brick Vista option a precomputed view definition always contains an aggregate query.

2

Aggregate Query Rewrites

Precomputing query results is a powerful means of speeding up query performance. The Red Brick Vista option realizes the performance gain by intercepting and rewriting users' queries to use precomputed views.

However, space and maintenance costs make precomputing the results of every possible aggregate query prohibitive. There is a practical limit to the number of precomputed views and tables that can be efficiently created and maintained. The Red Brick Vista option solves this problem in two ways:

- By rewriting queries even when they do not exactly match the view definition:
 - A query that requests some subset of the data in the view can be rewritten. See page 4-3.
 - Queries that apply additional calculations to the view data (by using a RISQL display function, for example) can be rewritten. See page 4-3.
 - Queries that group by columns of a coarser granularity than the grouping columns in the view can be rewritten, such as queries grouped by the State column when the view is grouped by City. This feature is explained on page 2-6, and examples are presented on page 4-8.
- By logging rewritten query activity and offering advice on the optimal set of precomputed views to create. For details about this functionality, refer to Chapter 5, "Using the Advisor."

In these ways, and in combination with other Red Brick Warehouse techniques for accelerating query performance (such as STARjoin and TARGETjoin processing), the Red Brick Vista option improves query performance while minimizing both space requirements and the administrator's maintenance workload. For information about indexing requirements for query rewriting, refer to Chapter 3, "Using the Query Rewrite System."

Example

A group of retail sales analysts routinely request a report that compares sales totals for specific products during specific quarters. Using the Aroma database, the analyst's query joins the Sales fact table to the Product and Period dimension tables:

User's Query

```
select prod_name, qtr, sum(dollars) as total_sales
from sales, product, period
where sales.prodkey = product.prodkey
      and sales.classkey = product.classkey
      and sales.perkey = period.perkey
group by prod_name, qtr;
```

To take advantage of the Red Brick Vista option, the administrator creates and populates the Product_Sales aggregate table:

Aggregate Table Definition

```
create table product_sales
  (prod_name char(30), qtr char(5), dollars dec(13,2));
```

INSERT Statement for the Aggregate Table

```
insert into product_sales
select prod_name, qtr, sum(dollars) as total_sales
from sales, product, period
where sales.prodkey = product.prodkey
      and sales.classkey = product.classkey
      and sales.perkey = period.perkey
group by prod_name, qtr;
```

Then the administrator creates a precomputed view associated with the Product_Sales aggregate table. The view definition contains a query expression that reflects the exact contents of the aggregate table:

Precomputed View Definition

```
create view product_sales_view(prod_name, qtr, dollars) as
select prod_name, qtr, sum(dollars) as total_sales
from sales, product, period
where sales.prodkey = product.prodkey
      and sales.classkey = product.classkey
      and sales.perkey = period.perkey
group by prod_name, qtr
using product_sales(prod_name, qtr, dollars);
```

Using the Product_Sales_View, the query rewrite system can intercept the multi-table join query and replace it with a scan of the Product_Sales aggregate table. The query rewrite system assigns table names, rewrites join predicates, and represents queries in a form that will provide optimal query performance.

2

The SQL generated by the query rewrite system in this case would be equivalent to the following query:

```
select * from product_sales;
```

Note: This example represents the simplest approach to using the query rewrite system and demonstrates how the system works rather than the best way to use it; in most cases, Red Brick Systems does not recommend the definition of aggregate tables that are isolated from the database schema. Several more practical examples, which involve aggregate tables that can be efficiently joined to dimension tables, are presented in Chapter 4, “Query Rewrite Case Studies.”

For more information about the process of creating precomputed views and aggregate tables, refer to Chapter 3, “Using the Query Rewrite System.”

Rollups and Hierarchies

Aggregate queries group or “roll up” values of a finer granularity into smaller sets of values of a coarser granularity. The performance gain offered by the query rewrite system derives from the precomputation of these rollups. However, one of the innovative and powerful features of the Red Brick Vista option is the ability to rewrite queries that require *additional* rollups, involving columns of a coarser granularity still than the grouping columns of the view.

In other words, the Red Brick Vista option can be used to rewrite a large number of queries that do not match the query defined in the view. For example, the view might define a query that returns rows grouped by a Month column, yet this view can be used to rewrite queries grouped by the Qtr and Year columns, despite the fact that neither of these columns is named in the query defined by the view.

If this rollup capability did not exist, the administrator would have to create three views—grouped by Month, Qtr, and Year, respectively—or one very wide view grouped by all three columns. The following sections explain how rollups work.

Functional Dependencies

Rollups to columns not defined in precomputed views are made possible by the existence of functional dependencies inherent in warehouse data. A *functional dependency* is a many-to-one relationship shared by two columns of values. In other words, a functional dependency from column X to column Y is a constraint *that requires two rows to have the same value for the Y column if they have the same value for the X column.*

The two columns might be in the same table or different tables. For example, if a functional dependency exists between the Store_Number and City columns in a Store table, it must be true that whenever the value in the Store_Number column is, say, *Store#56*, the value in the City column is the same —*Los Angeles*, for example. This relationship is many-to-one because there could be many stores in a city, but a given store can only be in one city. Similarly, the City column in the Store table might have a many-to-one relationship with a Region column in the Market table; for example, if the city is *Los Angeles*, the region is always *West*.

The existence of a functional dependency allows precomputed views grouped by columns of a finer granularity to be used to rewrite queries grouped by columns of a coarser granularity. For example, the existence of a Store_Number-to-City functional dependency means that it is safe to group the

Store_Number values into distinct City values. If a precomputed view is grouped by Store_Number, it is not necessary to create another view grouped by City; the same view can be used to rewrite queries that constrain on one or both of these columns.

As long as the query rewrite system *knows* about the functional dependencies that exist in the database, it can use them intelligently to rewrite queries that require a rollup beyond the scope of the precomputed view definition. In this sense, there are two types of functional dependencies—those known to the query rewrite system by default and those that must be declared by the warehouse administrator.

Dependencies Declared by the Administrator

The query rewrite system is not aware of the functional dependencies that exist between non-key columns in the database. For example, if a view is grouped by the Month column in the Period table and dependencies exist from Month to Qtr and from Qtr to Year, both dependencies need to be declared. After they have been declared, the query rewrite system can use the same precomputed view to rewrite queries grouped by any combination of the three columns. Declaring these dependencies also helps the Red Brick Vista Advisor recommend an optimal set of candidate views.

The mechanism for declaring a functional dependency is the CREATE HIERARCHY command. A CREATE HIERARCHY statement names pairs of columns that satisfy functional dependencies and identifies the tables to which the columns belong. (In the context of the Red Brick Vista option, the terms *functional dependency* and *hierarchy* are synonymous.)

Functional dependencies declared with CREATE HIERARCHY statements are not validated by the warehouse server. A many-to-one relationship is assumed to exist between the two columns, and the query rewrite system will use the dependency regardless of the data values stored in the columns. *Therefore, it is the administrator's responsibility to ensure the validity of each explicitly defined hierarchy before declaring it.* Otherwise, the query rewrite system might return incorrect results.

For example, compare the pairs of values in the following table. If the Period table contains the second set of values, a hierarchy from Qtr to Year would be valid because there is a unique first-quarter value for each year, a unique second-quarter value for each year, and so on. If the Period table contains the first set of values, however, the hierarchy would not be valid because the Qtr column has the same first-quarter value (Q1) for 1997, 1998, and beyond.

Invalid Relationship		Valid Relationship	
Qtr Column	Year Column	Qtr Column	Year Column
Q1	1997	Q1_97	1997
Q2	1997	Q2_97	1997
Q3	1997	Q3_97	1997
Q4	1997	Q4_97	1997
Q1	1998	Q1_98	1998
Q2	1998	Q2_98	1998
Q3	1998	Q3_98	1998
Q4	1998	Q4_98	1998
Q1	1999	Q1_99	1999
...

For information about querying the data to verify that a hierarchy is valid, refer to page 3-14. For the complete syntax of the CREATE HIERARCHY command, refer to the *SQL Reference Guide*.

Dependencies Known Implicitly to the Query Rewrite System

Hierarchies that follow the path of a primary key/foreign key relationship are implicitly known to the query rewrite system. The result of this knowledge is that a view grouped by the Sales.Perkey column, where Perkey is a foreign key column that references the Period table, can be used *automatically* to rewrite queries grouped by *any combination of columns* in the Period table.

This behavior is also true for queries grouped by columns in outboard tables (tables referenced by dimension tables). For example, a view grouped by the Sales.Storekey column, where Storekey is a foreign key column that references the Store table and Store.Mktkey is a foreign key column that references the Market table, can be used *automatically* to rewrite queries that group by *any combination of columns* in the Store and Market tables. For examples of rewritten queries that demonstrate the use of implicit hierarchies, refer to page 4-16.

Derived Dimensions

Derived dimensions are aggregate dimension tables that contain a subset of the columns in the corresponding detail dimension. These tables can be joined to aggregate fact tables to form an aggregate table “family.” Precomputed views associated with derived dimension tables optimize the performance of rewritten queries that require additional rollups because they simplify the generated SQL and facilitate efficient indexing.

For detailed information about creating and using derived dimensions, refer to page 3-17 and the case study that begins on page 4-11.

Key Concepts of Query Rewriting
Rollups and Hierarchies



Using the Query Rewrite System

The query rewrite system offers substantial performance gains when users' queries request aggregations of factual information stored in large databases. This chapter explains how to make query rewriting possible by creating precomputed views and declaring hierarchies.

These procedures assume that you have a clear understanding of the concepts introduced in Chapter 2. In Chapter 4, several tutorial-based examples focus on specific queries and how they are rewritten.

This chapter is divided into the following sections:

- Creating Aggregate Tables
- Creating Precomputed Views
- Using Hierarchies
- Optimizing Query Rewrites
- Setting Up the Query-Rewriting Environment
- Checklist of Query-Rewriting Tasks

For reference information and complete syntax diagrams for each SQL command described in this chapter, refer to the *SQL Reference Guide*.

Creating Aggregate Tables

In general terms, *aggregate tables* contain information that has a coarser granularity (fewer rows) than the *detail* records used to load data warehouse tables. For example, in a retail database, the transaction-level data drawn from the OLTP system might be in the form of individual sales receipts. The `Sales_Receipts` fact table would contain this detail data, but the records in that table might also be aggregated over certain time periods to produce a set of aggregate tables (`Sales_Daily`, `Sales_Monthly`, and so on).

In the context of the Red Brick Vista option, an *aggregate table* is a special case of a precomputed table. It is a physical database table defined expressly for the purpose of storing the results of an aggregate query defined in a precomputed view.

An aggregate table is created with a standard `CREATE TABLE` statement. It does not have to share a primary key/foreign key relationship with other tables in the database. However, a common design strategy involves creating a “family of aggregate tables,” including a referencing aggregate table that references one or more aggregate dimension tables known as *derived dimensions*. This strategy is a means of optimizing query-rewriting performance, as described on page 3-17.

Populating Aggregate Tables

Aggregate tables can be loaded with either Table Management Utility (TMU) `LOAD DATA` operations or `INSERT INTO...SELECT` statements.

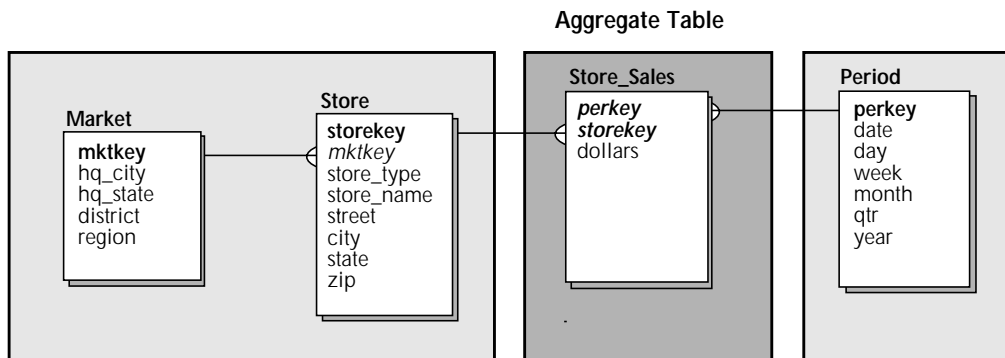
For information about the TMU and the Auto-Aggregate feature, refer to the *Table Management Utility Reference Guide*. For the syntax of the `INSERT` command, refer to the *SQL Reference Guide*.

Example of an Aggregate Table

The following example is based on the sample Aroma database, which is described in detail in the *SQL Self-Study Guide*.

The retail schema of the Aroma database consists of the detail Sales table and six dimension tables—Period, Store, Product, Promotion, Market, and Class. The Market and Class tables are outboard tables. The Dollars column in the Sales table represents totals *per day, per store, per product, per promotion*. For example, a single row in the fact table might record that on January 2, 1996, the San Jose Roasting Company sold \$95.00 of whole-bean Aroma Roma coffee to customers using Aroma catalog coupons.

If users routinely submit queries that request sales totals per some time period, per some store or geographical area (such as *per day, per region* or *per month, per state*), the administrator might define a Store_Sales table that contains sales totals for all products and all promotions *per day, per store*. This aggregate table would retain the same relationship to the Store and Period dimension tables as the detail Sales table but would not reference the other dimensions in the Aroma retail schema.



For example, a single row in this aggregate table might record that on January 2, 1996, total sales at the San Jose Roasting Company were \$4,288.50.

CREATE TABLE Statement

The CREATE TABLE statement for Store_Sales would look like this:

```
create table store_sales
(perkey int not null, storekey int not null, dollars dec(13,2),
primary key (perkey, storekey),
foreign key (perkey) references period (perkey),
foreign key (storekey) references store (storekey))
maxrows per segment 50000;
```

It is not necessary to define an entirely new set of aggregate tables for use with the Red Brick Vista option; existing aggregate tables can be associated with precomputed views. However, the contents of an existing aggregate table must match the definition of its precomputed view (see page 3-5).

INSERT INTO...SELECT Statement

The INSERT statement used to load the Store_Sales table would look like this:

```
insert into store_sales
(select perkey, storekey, sum(dollars)
from sales
group by perkey, storekey);
```

This is a very simple example of an aggregate table whose data is drawn from a single fact table and grouped by a subset of the foreign keys that make up the detail table's primary key. The Store_Sales table contains less than 15,000 rows, whereas the detail Sales table contains almost 70,000 rows.

For examples of other types of aggregate tables whose data is drawn from joins of multiple detail tables and grouped by non-key columns, refer to Chapter 4, "Query Rewrite Case Studies."

Creating Precomputed Views

The query rewrite system uses precomputed views to determine which queries can be rewritten to improve performance. A *precomputed view* is a special type of view that is linked to an aggregate table. The aggregate table definition (CREATE TABLE statement) describes the columns and datatypes that the table contains, whereas the precomputed view definition (CREATE VIEW...USING statement) describes the exact *contents* of those columns; it describes how, in terms of an SQL query expression, the aggregate data is precomputed from the detail data.

In this sense, the view is “precomputed” *when the aggregate table is loaded with the appropriate data*. Creating a precomputed view does not populate or “materialize” its associated aggregate table. The administrator must ensure that the view definition and the TMU or SQL language used to load the aggregate table evaluate to the same contents. (Precomputed views can be defined before or after their aggregate tables are loaded.)

CREATE VIEW...USING Command

A precomputed view is defined with a CREATE VIEW...USING statement. The statement contains two distinct blocks of information:

- A query expression that “selects” the data for the aggregate table from one or more tables—typically a detail-level fact table and some of its dimension tables. The select list must include at least one aggregation column or at least one grouping column. The grouping columns in the select list and GROUP BY clause can be key or non-key columns (or a combination of the two).

Issued as a SELECT statement, the query expression used to define the view *must* return a result set that exactly matches the contents of the named aggregate table.

- A USING clause that names the aggregate table and its columns. (Without this clause, the CREATE VIEW statement creates a regular view, not a precomputed view.) Each precomputed view you create must use a different aggregate table.

For example:

```
create view store_sales_view as
  select perkey, storekey, sum(dollars) as dollars
  from sales
  group by perkey, storekey
  using store_sales (perkey, storekey, dollars);
```

Aggregation Columns

Aggregation columns in precomputed view definitions must be of the form

```
set_function(expression)
```

where *set_function* is one of the following aggregation functions:

- SUM
- MIN
- MAX
- COUNT

and *expression* is a simple or compound expression that contains column names from the detail fact table in the FROM clause of the view definition and/or constants.

The COUNT DISTINCT and COUNT(*) functions are also supported.

Note the following restrictions:

- Expressions used as arguments to the COUNT function must be simple expressions.
- Expressions that contain scalar functions, RSQL display functions, and subqueries are not supported.
- The expression for the SUM function must be numeric.
- The AVG set function cannot be used; however, AVG queries can be rewritten if the appropriate aggregate table contains SUM and COUNT values for the same column. For an example, refer to “Case 6—Rewriting a Query That Calculates Averages” on page 4-23.
- The DISTINCT function can only be used as an argument to the COUNT function. SELECT DISTINCT queries cannot be used.

For detailed information about set functions and expressions, refer to the *SQL Reference Guide*.

Examples

The following expressions are examples of valid aggregation columns, assuming that the Sales table is the detail fact table named in the FROM clause of the view definition:

```
sum(sales.dollars)  
min(sales.dollars/sales.quantity)  
max((sales.quantity) * 10)
```


Precomputed Query Expressions

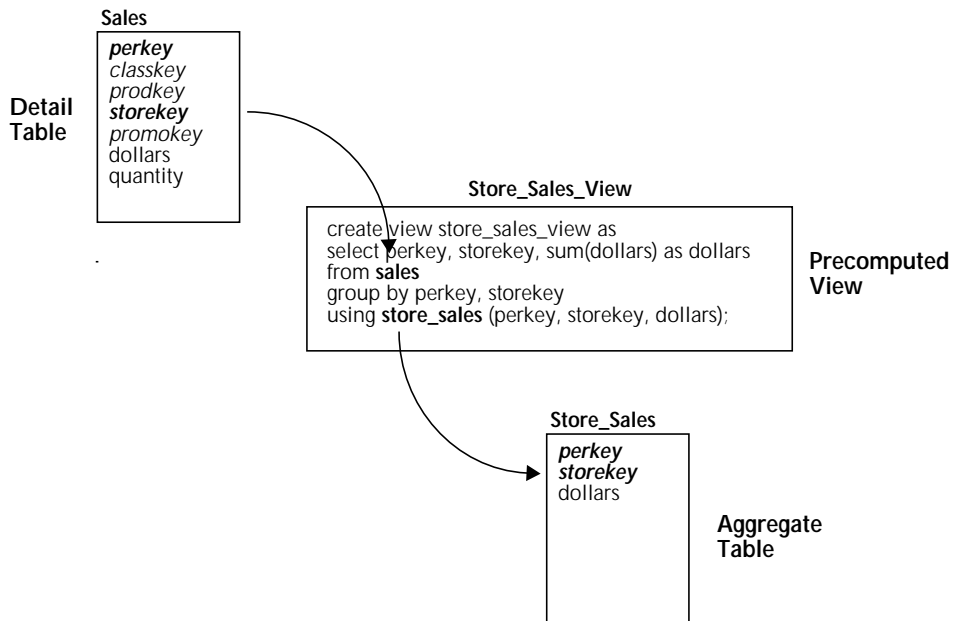
The query expression defined for a precomputed view is also restricted in the following ways:

- It cannot contain a subquery, a HAVING clause, or a WHEN clause.
- Join predicates must be expressed in the WHERE clause, with equality conditions only.

For the complete syntax of the CREATE VIEW...USING command, refer to the *SQL Reference Guide*.

Example View Definition

The following diagram illustrates the definition of a precomputed view. The query in the view definition aggregates data derived from the detail Sales table and stores the data in the Store_Sales aggregate table.



This precomputed view can be used to rewrite a large number of queries that select from the Sales, Period, and Store tables. For specific examples, refer to “Case 4—Using Implicit Hierarchies to Rewrite Queries” on page 4-16.

Cost-Based Analysis of Precomputed Views

Query rewriting is cost-based; that is, every valid precomputed view in the database is evaluated before the optimal SQL is generated for each “block” of the query. In this context, a block is a query expression. For example, a subquery is a separate query block, and each query expression on either side of a UNION operator is a separate query block. The cost-based analysis chooses the best possible rewrite for each block by considering the size of the aggregate table associated with each view and the joins required to rewrite the query with that table.

Because the query rewrite system considers every valid view before generating the rewritten SQL, the query rewrite system incurs some overhead of its own when the database contains a large number of views. For example, if there are 50 valid views in the database, a UNION query (two query blocks) will trigger 100 “view checks,” each one requiring a fraction of a second to complete. These fractions of a second add to the query-processing time incurred to compute the result set. Precomputed views are not screened, except in terms of being marked *valid* or *invalid*—as discussed under “Marking Precomputed Views Valid” on page 3-23.

Indexes Not Considered by the Cost Model

The query rewrite system does not take into account the indexes available to rewrite queries; therefore, indexing is an essential prerequisite to using the Red Brick Vista option. For example, query rewrites that involve aggregate table families in a star schema must make use of STAR indexes and TARGET indexes equivalent to those defined on the detail tables they replace. For examples of rewritten queries optimized by the use of indexes, refer to “Case 3—Optimizing Query Rewrites with Derived Dimensions” on page 4-11.

For information about standard approaches to indexing database tables, refer to the *Warehouse Administrator’s Guide*.

Rewritten INSERT INTO...SELECT Statements

SQL statements used to insert rows into tables can be rewritten as well as queries that select rows from tables. If you are using INSERT INTO...SELECT statements to populate your aggregate tables, you can dramatically increase the performance of the INSERT operations by creating and loading aggregate tables and precomputed views in order of granularity (from finest to coarsest).

For example, if you intend to create a set of aggregate Sales tables with data grouped by different periods of time, you should create and populate the Weekly_Sales table before you create the Monthly_Sales, Quarterly_Sales, and Annual_Sales tables. After you have created and populated the Weekly_Sales table, create its precomputed view, mark it valid, and turn the query rewrite system on (see page 3-23). When you create the Monthly_Sales table, the INSERT statement you use to populate the table will be rewritten to use the Weekly_Sales aggregate table. Repeat this “cascading insert” process for each table and view combination.

Queries That Cannot Be Rewritten

Regardless of the validity and definition of precomputed views, some types of queries cannot be rewritten. The following list is not exhaustive, but it does describe the main classes of queries that cannot be rewritten.

- Queries that contain BREAK BY and RESET BY clauses.
- Queries that contain outer joins.
- Subqueries in the WHERE clause of DELETE and UPDATE statements.
- Queries whose GROUP BY clauses contain compound expressions. For example, the following query cannot be rewritten because the GROUP BY clause references an expression that contains an arithmetic operator:

```
select perkey +1 as day, sum(dollars) as total_sales
from sales
group by day;
```

For information about compound expressions, refer to the *SQL Reference Guide*.

- Queries that contain multiple references to the same table in the FROM clause. For example, the following self-join query cannot be rewritten because its FROM clause contains two references to the Product table:

```
select a.prod_name as products, a.pkg_type, sum(dollars)
from product a, product b, sales
where a.prod_name = b.prod_name
and a.pkg_type <> b.pkg_type
and sales.prodkey = a.prodkey
and sales.classkey = a.classkey
group by a.prod_name, a.pkg_type
order by products, a.pkg_type;
```

For more information about self-joins, refer to the *SQL Self-Study Guide*.

- Queries that contain both a UNION, INTERSECT, or EXCEPT set operator *and* a predicate on the result of the set operation. For example, the following query *can* be rewritten:

```
select perkey from period
union
select perkey from sales;
```

However, the following query contains a predicate on the result of the UNION operation and *cannot* be rewritten:

```
select * from
  (select storekey from sales
   union
   select storekey from store) as t1
where t1.storekey <10;
```

Using Hierarchies

When an aggregate query requires a coarser grouping of the data than the grouping expressed in the precomputed view, the query rewrite system can compute the additional rollup to answer the query using that view. Because of this extended rollup feature, the administrator can define a relatively small number of views that can speed up most, if not all, of the aggregate queries that users routinely submit.

The key to this query-rewriting flexibility is the exploitation of functional dependencies inherent in warehouse data (see page 2-6). These dependencies follow the path of primary key/foreign key relationships that are an integral part of the warehouse schema design; as long as the query rewrite system knows that these dependencies exist, they can be used to extend the range of queries that can be rewritten.

The following sections explain how functional dependencies are made known to and used by the query rewrite system.

Explicit Hierarchies

An *explicit hierarchy* is a functional dependency defined with a CREATE HIERARCHY statement. The declaration of explicit hierarchies is a routine task that prepares the database to make optimal use of the Red Brick Vista option, in much the same way that defining primary key/foreign key relationships is a routine prerequisite for creating database tables.

3

The definition of hierarchies not only extends the usability of existing precomputed views; it also gives the Advisor more information to work with when it generates candidate views. Therefore, it is recommended that hierarchies be created for all the functional dependencies in the database, regardless of existing or anticipated aggregation strategies.

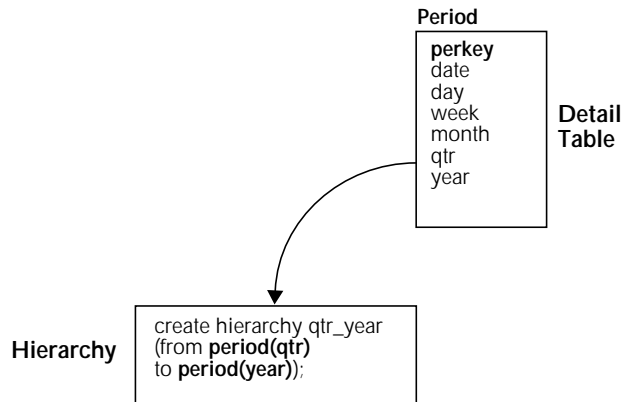
Caution: Hierarchies must be defined with great care. The declaration of hierarchies on columns whose values do not satisfy a many-to-one relationship might cause rewritten queries to return incorrect results, *without warning*. The warehouse server does not validate hierarchies when they are declared; nor does it inform the user when a valid hierarchy becomes invalid because of modifications to the database. It is the administrator's responsibility to ensure the validity of a hierarchy before declaring it and to drop valid hierarchies should they become invalid.

For examples of valid and invalid functional dependencies, refer to “Dependencies Declared by the Administrator” on page 2-7.

CREATE HIERARCHY Command

This command gives the hierarchy a unique name, then identifies the columns (and tables) that satisfy the functional dependency. A single CREATE HIERARCHY statement can define multiple dependencies, and each dependency can reference a pair of columns from the same table or two different tables.

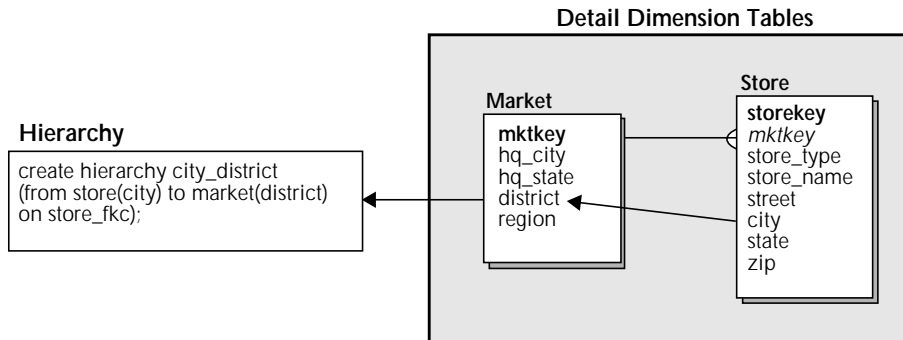
For example, the following hierarchy declares that a many-to-one relationship exists between the Qtr and Year columns in the Period table. This relationship is valid because every row in the Period table that contains a given value in the Qtr column has the same value in the Year column. For example, when the Qtr value is Q1_96, the Year value is always 1996.



Now that the query rewrite system knows that this dependency exists in the data, it will be able to use a precomputed view grouped by the Qtr column to rewrite queries grouped by the Year column. For specific examples of this kind of rewrite, refer to “Case 2—Making Use of Explicit Hierarchies” on page 4-8.

If the columns named in a hierarchy are from two different tables, the tables must have a foreign key/primary key relationship. This kind of hierarchy expresses a functional dependency between the *from* column and the foreign key column that references the table that contains the *to* column. Via this functional dependency, rollups to *any column* in the referenced table are implied.

For example, the following CREATE HIERARCHY statement declares a functional dependency between the Store.City and Market.District columns, but the query rewrite system interprets this relationship as a functional dependency between Store.City and Store.Mktkey.



Given knowledge of this hierarchy, the query rewrite system can use a precomputed view grouped by the City column of the Store table to rewrite queries grouped by *any column* in the Market table.

If the tables in a hierarchy contain multiple foreign key/primary key relationships, the statement must specify the constraint name that defines the foreign-key reference. The constraint name *store_fk* is optional in the previous example because only one foreign key/primary key relationship exists between the two tables (Mktkey to Mktkey).

3

A hierarchy can be dropped with the DROP HIERARCHY command, and the RBW_HIERARCHIES system table can be queried to obtain a list of explicitly defined hierarchies in the database.

For the syntax of the CREATE HIERARCHY and DROP HIERARCHY commands and information about foreign-key constraint names, refer to the *SQL Reference Guide*. For information about system tables, refer to the *Warehouse Administrator's Guide*.

Verifying the Validity of Hierarchies

To verify that a functional dependency exists in the data, run the following query, where *from_column* and *to_column* are the columns on which the hierarchy will be defined, and *table_name* is the name of the detail dimension table to which those columns belong:

```
select from_column, count(distinct to_column)
from table_name
group by from_column;
```

If the result of the COUNT(DISTINCT) function is 1 for all the rows in the result set, the hierarchy is valid. For example, the results of the following query show that the Qtr-to-Year hierarchy is valid:

```
select qtr, count(distinct year)
from period
group by qtr;
```

QTR	
Q1_94	1
Q1_95	1
Q1_96	1
Q2_94	1
Q2_95	1
Q3_94	1
Q3_95	1
Q4_94	1
Q4_95	1

If you are running the COUNT(DISTINCT) query on a very large dimension table, include a HAVING clause to verify that all the rows return 1 without having to scan the entire result set. For example:

```
select qtr, count(distinct year) as count_col
from period
group by qtr
having count_col <> 1;

QTR          COUNT_COL
```

In this case, the fact that the query returns *no rows* verifies that the hierarchy is valid.

If the hierarchy you want to create refers to columns in different tables, the COUNT(DISTINCT) function must operate on the foreign key column from the referencing table. For example:

```
select store_type, count(distinct mktkey)
from store
group by store_type;
```

STORE_TYPE	
Large	4
Medium	5
Small	8

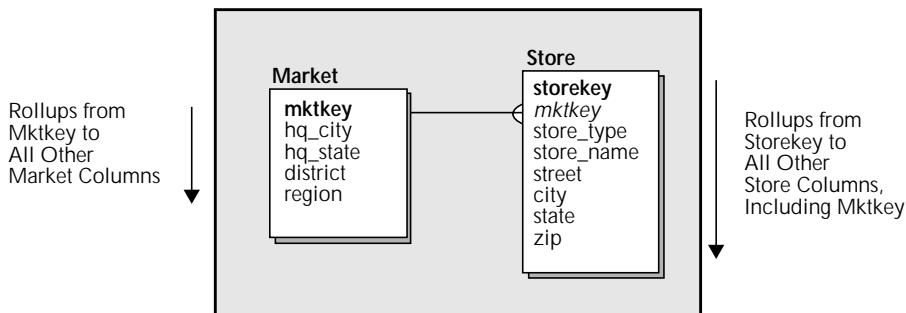
Note that the Mktkey column, which is a foreign key in the Store table and the primary key of the Market table, is the argument of the COUNT(DISTINCT) function in this case. The results of this query demonstrate that a hierarchy from the Store_Type column to *any column* in the Market table would not be valid.

Implicit Hierarchies

Implicit hierarchies are many-to-one relationships that already exist between columns and tables by virtue of their primary key/foreign key relationships. The existence of these hierarchies is known to the query rewrite system when it uses a precomputed view grouped by one or more key columns. In turn, when a query constrains on a non-key column from either the same table as the grouping key column or another referenced table, *the view can be used to rewrite queries grouped by any column in either table*. These kinds of rollups are possible regardless of the definition of any explicit hierarchies.

The Store_Sales_View illustrated on page 3-7 is an example of a view definition grouped by key columns. The view is grouped by Perkey and Storekey, which are the primary key columns of the Period and Store tables, respectively, and foreign key columns that make up the primary key of the Store_Sales aggregate table. A query that requests sums of dollars from the detail Sales table grouped by *any column* in the Period table and/or *any column* in the Store table can be rewritten to use this view.

Implicit rollups will also work between the Store table and its outboard table, Market. For example, a query that requests sales figures grouped by the Region or District columns in the Market table can be rewritten to use Store_Sales. This is possible because Storekey is a grouping column in the precomputed view and there is a known primary-key/foreign-key relationship between the Store and Market tables.



For specific examples of queries that can be rewritten using implicit hierarchies, refer to page 4-16.

Optimizing Query Rewrites

Although the query rewrite system can rewrite a large number of queries based on the existence of precomputed views and hierarchies, optimal performance is not guaranteed for some rewritten queries unless the administrator also creates two more structures:

- Derived dimensions
- Indexes on aggregate tables

Creating Derived Dimensions

Although the notion of aggregating data implies the manipulation of additive facts such as sales figures and costs, dimension table data can also be “aggregated” in logical ways. For example, product brands can be grouped into distinct product subcategories, categories, and departments, and days into weeks, months, quarters, and years.

Aggregate dimension tables are known as “derived dimensions” because they derive from an existing dimension table and contain some subset of its columns. The granularity of these columns is equal to or coarser than the granularity of the referencing aggregate fact table. Any logical subset of columns can be used; for example, you could define a derived Market dimension with three columns—State, District, and Region—or with any two of those columns, or even with any one of those columns.

Although the query rewrite system can roll up from non-key grouping columns as long as the functional dependencies in the database have been declared with CREATE HIERARCHY statements, derived dimensions simplify the generated SQL and optimize query performance in these cases.

A good rule of thumb is to define a derived dimension whenever you have declared a hierarchy between two columns from the same table and the first column (the *from* column in the hierarchy definition) is a non-key column.

Note: If an explicit hierarchy is declared between columns in different tables, the precomputed view for the derived dimension must be grouped by the *from* column in the hierarchy and the foreign key column that references the table in which the *to* column resides. For example, if the Qtr column is in the Period1 table, the Year column is in the Period2 table, and there is a primary key/foreign key relationship on the Period1.Yearkey and Period2.Yearkey columns, the precomputed view must be grouped by Period1.Qtr and Period1.Yearkey (*not* Period2.Year).

Simplified SQL Generation

When aggregate data is grouped by non-key columns and explicit hierarchies are used to rewrite queries, the generated SQL is quite complex. For example, if there is no derived dimension for the Period table, queries that require sales totals by Year when the aggregate Sales table is grouped by Qtr will require a join to the detail Period table. At the detail level, the Qtr values are not unique, so some GROUP BY processing is necessary to find the Year for each Qtr value.

However, in the derived dimension, the Qtr values are unique primary keys, so this additional processing is not required. In other words, by *precomputing* the grouped Qtr and Year rows into the derived dimension, the system avoids having to compute those rows as part of each query rewrite.

In a derived dimension, the column with the finest granularity is defined as its primary key, creating a primary-key/foreign-key relationship between the aggregate fact table and the derived dimension. Therefore, it is easy to create a STAR index that can join the family of aggregate tables; this index is equivalent to the STAR index that joins the detail tables in the same schema.

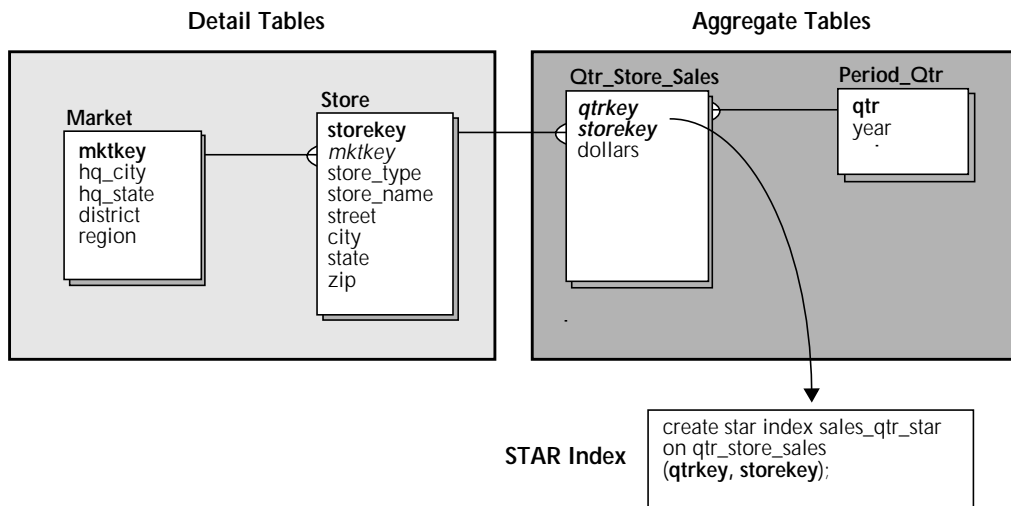
To summarize, if the administrator creates a precomputed view linked to an aggregate Sales table grouped by the Qtr column, the following additional objects will optimize rewrites of queries grouped by Year:

- A precomputed view linked to a derived Period dimension that contains only the Qtr and Year columns. This table would be referenced by the aggregate Sales table, and the Qtr column would be defined as the primary key.
- A hierarchy between the Qtr and Year columns in the Period table. Note that the hierarchy definition must refer to those columns in the detail dimension table, not the derived dimension. Also, the data in those columns must satisfy the functional dependency that the hierarchy declares, as explained on page 3-11.
- A STAR index on the aggregate Sales table to join it to the derived Period dimension.

Without the derived dimension and the STAR index, queries that group by Qtr and Year would still be rewritten, but queries grouped by Year might be rewritten with sub-optimal performance.

The following diagram illustrates this scenario. Because its values are unique, the Qtr column can be defined as the primary key of the derived dimension, Period_Qtr. In turn, the Qtr_Store_Sales aggregate table is defined to contain a Qtrkey column that is a foreign-key reference to the Qtr column.

The Store and Market dimension tables are not “derived” in this example and have the same relationship to both the detail Sales table and the aggregate Qtr_Store_Sales table. The relationships in this “family of aggregate tables” preserve the STARjoin and TARGETjoin capabilities of the detail table schema.



The generated SQL presented in “Case 3—Optimizing Query Rewrites with Derived Dimensions” on page 4-11 further illustrates the benefit of creating derived dimensions.

CREATE TABLE Statements

The CREATE TABLE statements for the two aggregate tables in this example look like this:

```
create table period_qtr
(qtr char(6) not null,
year int,
primary key (qtr));

create table qtr_store_sales
(qtrkey char(6) not null,
storekey int not null,
dollars dec(13,2),
primary key (qtrkey, storekey),
foreign key (qtrkey) references period_qtr (qtr),
foreign key (storekey) references store (storekey))
maxrows per segment 20000;
```

Note: Because derived dimensions are *referenced* tables, you have to create them *before* you create the *referencing* aggregate table.

INSERT Statements

These tables are loaded with the following INSERT statements:

```
insert into period_qtr
select qtr, year
from period
group by qtr, year;

insert into qtr_store_sales
select qtr, storekey, sum(dollars)
from sales, period
where sales.perkey = period.perkey
group by qtr, storekey;
```

Note that the INSERT statement for the Period_Qtr table contains no aggregation column, just two grouping columns. Instead of calculating sums or other aggregations, the query expression selects distinct combinations of Qtr and Year values.

Precomputed View Definition

The query expression in the precomputed view for the derived dimension table must match the query expression in the INSERT statement that loaded the table:

```
create view pd_qtr_view as
  select qtr, year
  from period
  group by qtr, year
using period_qtr (qtr, year);
```

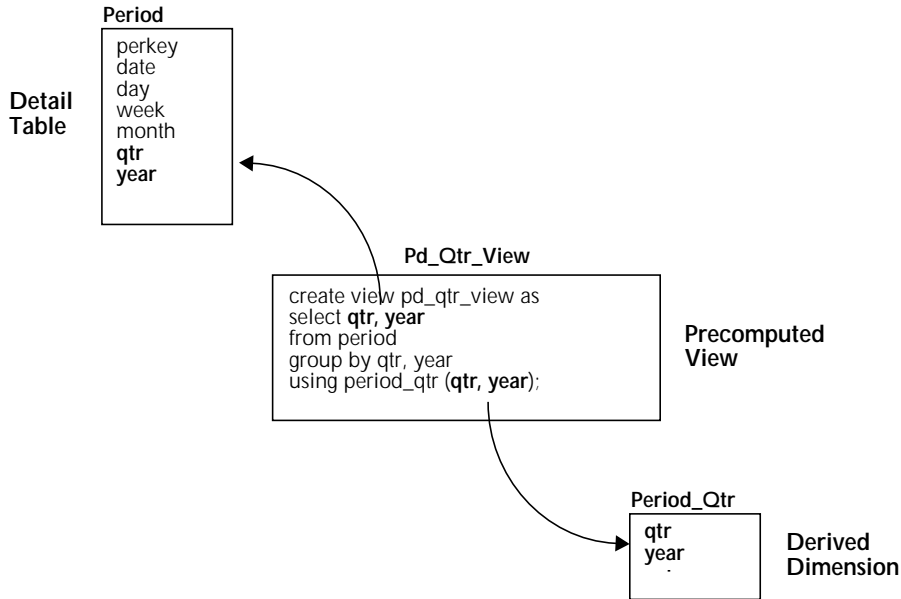
Note: This kind of precomputed view must be defined with a GROUP BY clause. The following equivalent query expression, which uses the DISTINCT function in the select list, will return a syntax error despite the fact that it computes the same result set.

```
create view pd_qtr_view as
  select distinct qtr, year
  from period
using period_qtr (qtr, year);

** ERROR ** (1901) The query expression for the precomputed
view is invalid.
```

In general, the same syntax rules apply to all precomputed views, whether their linked aggregate tables are derived dimensions, referencing (fact) tables, or stand-alone tables that have no relationship to other tables in the database schema. For complete syntax information, refer to the *SQL Reference Guide*.

The following diagram illustrates the view definition for the Period_Qtr table:



Creating Indexes

In general terms, query rewriting guarantees accelerated query performance when *indexes equivalent to those defined on the detail tables are defined on the aggregate tables*. In other words, the Red Brick Vista option should be used in conjunction with standard Red Brick Warehouse performance and tuning techniques—especially STARjoin and TARGETjoin query processing. Indexes are particularly important when families of aggregate tables are used, as explained previously in the discussion of derived dimensions.

Before using the query rewrite system, compare the aggregate tables you have created to their corresponding detail tables and create the following types of indexes on equivalent columns:

- B-TREE indexes
- STAR indexes
- TARGET indexes, including those intended for TARGETjoin processing.

For general indexing instructions, refer to the *Warehouse Administrator's Guide*.

Setting Up the Query-Rewriting Environment

Before you can use the query rewrite system to rewrite aggregate queries, complete the following configuration tasks.

For more information about the SET commands and configuration parameters discussed here, refer to the *SQL Reference Guide* and the *Warehouse Administrator's Guide*.

Marking Precomputed Views Valid

When a precomputed view is created, it defaults to an *invalid* state, and its associated aggregate table will not be used to rewrite queries until you have marked the view valid.

You can mark a precomputed view valid in two ways:

- By marking the view itself valid. For example:

```
set precomputed view store_sales_view valid;
```

where *store_sales_view* is the name of the precomputed view.

- By marking all the views associated with a given detail table valid. For example:

```
set precomputed views for sales valid;
```

where *sales* is the detail table, not the aggregate table.

These commands can also be used to mark views invalid.

Using Invalid Views

The following command makes all the invalid views in the database available to the query rewrite system:

```
set use invalid precomputed views on;
```

This command should be used with caution; its purpose is to provide an administrator's shortcut that allows all of the existing precomputed views in the database to be considered for query rewrites, regardless of the validity of the data in the tables they reference.

The following warning message is displayed when an invalid view is used to rewrite a query:

```
** WARNING ** (1928) Query was executed using one or more invalid precomputed views.
```

The following command invalidates the data for all views associated with detail tables that are updated after the view is created:

```
set auto invalidate precomputed views on;
```

This command, and its corresponding configuration parameter, is the warehouse administrator's mechanism for ensuring that invalid views are unavailable to the query rewrite system as soon as the data stored in associated detail tables changes.

For example, assume that the SET AUTO INVALIDATE... command is set to ON. The administrator populates the Store_Sales aggregate table with an INSERT statement that selects from the Sales table; creates the Store_Sales_View, using the same query expression as the INSERT statement; and marks the view valid. If some rows in the Sales table are subsequently updated, the precomputed view will be marked invalid and will not be used by the query rewrite system.

Turning On the Query Rewrite System

Turn on the query rewrite system by using the following SET command or its equivalent configuration parameter:

```
set precomputed view query rewrite on;
```

The default behavior of the warehouse server is to rewrite queries. If you do not want queries to be rewritten, you must set this command (or the configuration parameter) to OFF.

Generating Statistics

Generate detailed statistics for queries by using the SET STATS INFO command and note the performance gain when queries are rewritten. When a query is rewritten, the following message is displayed:

```
** INFORMATION ** (1461) SQL statement was rewritten to use one or  
more precomputed views.
```

Use the EXPLAIN command to capture more detailed information about the execution of rewritten queries. See page 4-21 for an example of the EXPLAIN output.

Querying the RBW_VIEWS System Table

To get detailed information about precomputed views and their associated aggregate tables and detail tables, the administrator can query the RBW_VIEWS system table.

Note: Aggregate tables and precomputed views are not distinguished from detail tables (base tables) and regular views in the Type column of the RBW_TABLES system table; they are marked TABLE and VIEW, respectively.

Examples

The following query lists all the aggregate tables that are associated with precomputed views, then identifies the detail table for each aggregate table:

```
select name as view_name,  
       precompview_table as aggregate_table,  
       detail_table  
from rbw_views  
order by 3;
```

VIEW_NAME	AGGREGATE_TABLE	DETAIL_TABLE
PERIOD_QTR_VIEW	PERIOD_QTR	PERIOD
STORE_SALES_VIEW	STORE_SALES	SALES
SALES_RANK_VIEW	SALES_CONSTANT	SALES
...		

In this context, the *detail table* is the referencing table named in the FROM clause of the view definition (or the detail dimension table in the case of a derived dimension).

The following query lists all the *valid* precomputed views. Regular views contain NULLs in the Valid column of the RBW_VIEWS table; precomputed views are marked Y or N.

```
select name from rbw_views where valid = 'Y';  
  
NAME  
  
PERIOD_QTR_VIEW  
QTR_STORE_SALES2_VIEW  
QTR_STORE_SALES1_VIEW  
STORE_SALES_VIEW  
...
```

For information about marking views valid and invalid, refer to page 3-23.

The following query lists the precomputed views associated with the Sales table:

```
select name, precompview_table
from rbw_views
where detail_table = 'SALES';
```

NAME	PRECOMPVIEW_TABLE
QTR_STORE_SALES2_VIEW	QTR_STORE_SALES2
QTR_STORE_SALES1_VIEW	QTR_STORE_SALES1
STORE_SALES_VIEW	STORE_SALES
...	

For detailed information about the contents of view-related system tables, refer to the *Warehouse Administrator's Guide*.

Making Precomputed Views Invisible to Client Tools

When ODBC client applications are used to query a Red Brick Warehouse database, the Red Brick ODBC Driver first queries a view of the RBW_TABLES system table, called RBW_TABLES_VIEW, not the system table itself. If the view does not exist, the driver queries RBW_TABLES.

This behavior is a mechanism for customizing the list of tables and views visible to users of client tools. To make aggregate tables (that is, tables associated with precomputed views) invisible to client tools, define a view named RBW_TABLES_VIEW as follows:

```
create view RBW_TABLES_VIEW as
select * from rbw_tables
where rbw_tables.name not in
  (select rbw_views.precompview_table
   from rbw_views
   where rbw_views.precompview_table is not null);
```

To make both aggregate tables and their precomputed views invisible, define the view like this:

```
create view RBW_TABLES_VIEW as
select * from rbw_tables
where rbw_tables.name not in
  (select rbw_views.precompview_table
   from rbw_views
   where rbw_views.precompview_table is not null)
and rbw_tables.name not in
  (select rbw_views.name
   from rbw_views
   where rbw_views.precompview_table is not null);
```

This procedure is recommended because the query rewrite system rewrites queries that users submit against detail tables. Users are not intended to query precomputed views and their aggregate tables directly.

Note: This procedure works only with client tools that use the ODBC function `SQLTables()` to query the RBW_TABLES system table.

Checklist of Query-Rewriting Tasks

To use the query rewrite system:

Action	Page
1. Enable the Red Brick Vista option with a license key.	Note: Refer to the <i>Installation and Configuration Guide</i> for details.
2. Create and populate aggregate tables: <ul style="list-style-type: none">– Use the CREATE TABLE command to create the tables.– Use the INSERT command or Table Management Utility (TMU) LOAD DATA operations to populate the tables.	3-2
3. Create precomputed views with the CREATE VIEW...USING command.	3-5
4. Declare functional dependencies by using the CREATE HIERARCHY command.	3-11
5. Optimize query-rewriting performance by: <ul style="list-style-type: none">– Creating derived dimensions.– Creating indexes on aggregate tables.	3-17
6. Set up the query-rewriting environment: <ul style="list-style-type: none">– Mark precomputed views valid.– Turn on the query rewrite system.– Generate statistics for rewritten queries.	3-23

Query Rewrite Case Studies

This section presents several examples of query rewriting. Each case study begins by presenting either a specific query or a type of query routinely submitted by users, then shows how to use the query rewrite system to accelerate query performance. Performance statistics can be compared by using the SET STATS INFO command and turning the query rewrite system on and off before running each query.

This chapter is divided into the following sections:

- General Instructions
- Case 1—Rewriting a STARjoin Query
- Case 2—Making Use of Explicit Hierarchies
- Case 3—Optimizing Query Rewrites with Derived Dimensions
- Case 4—Using Implicit Hierarchies to Rewrite Queries
- Case 5—Rewriting Subqueries
- Case 6—Rewriting a Query That Calculates Averages

General Instructions

Each case in this chapter is presented as a tutorial that you can work through by using the RISQL Entry Tool to connect to the sample Aroma database. The following general steps apply to all the cases:

1. Start by setting up the database—by creating aggregate tables, precomputed views, indexes, and hierarchies. (Use the EDIT command to create and save your DDL files.)
2. Populate the aggregate tables by using INSERT statements.
3. Mark the precomputed views as valid by using one of the following SET commands:

```
set precomputed view view_name valid;  
set precomputed views for detail_table valid;
```

4. Generate detailed statistics for queries:

```
set stats info;
```

The output of this command indicates whether the queries you submit have been rewritten. (To get more information about the tables and indexes used to rewrite queries, use the EXPLAIN command.)

5. Run some queries with the query rewrite system turned on:

```
set precomputed view query rewrite on;
```

6. Run the same queries with the query rewrite system turned off and compare the performance:

```
set precomputed view query rewrite off;
```

Troubleshooting

If an expected query rewrite does not occur, check the following:

- Is the query rewrite system turned on?
- Are the precomputed views marked valid?
- Does the query constrain on a column “unknown” to the view? (For an example, see page 4-7.)
- Does the query fall into one of the classes of queries that cannot be rewritten? (See page 3-9.)

Case 1—Rewriting a STARjoin Query

The first example demonstrates the simplest case in which query rewriting improves query performance—that is, when the aggregate table contains the exact result set requested by the query. The only database objects the DBA must create are a precomputed view and its associated aggregate table.

The Query

The following query returns quarterly sales figures for each store in the Store table.

```
select store_name, qtr, sum(dollars) as total_sales
from sales, period, store
where sales.perkey = period.perkey
      and sales.storekey = store.storekey
group by store_name, qtr;
```

To answer this query, the server joins three tables over the STARindex built on the detail Sales table. The results are returned in a matter of seconds, but since this kind of report is requested routinely by several users, the administrator wants to obtain the result set even faster.

The administrator precomputes the results into an aggregate table. Using the query rewrite system, the administrator can do this without requesting that the users query the new table. Their view of the database need not change, and the query they submit will be exactly the same.

The following steps show how to make this performance improvement possible.

Step 1—Create the Aggregate Table

Issue a standard CREATE TABLE statement to create an aggregate table that contains three columns equivalent to the columns in the query's select list:

```
create table quarterly_store_sales
(store_name char(30),
qtr char(5),
dollars dec(13,2));
```

Step 2—Populate the Aggregate Table

Issue an INSERT INTO...SELECT statement to load the aggregate table:

```
insert into quarterly_store_sales (store_name, qtr, dollars)
select store_name, qtr, sum(dollars) as dollars
from sales, period, store
where sales.perkey = period.perkey
      and sales.storekey = store.storekey
group by store_name, qtr;
```

Step 3—Create the Precomputed View

Issue a CREATE VIEW statement with a USING clause to create a precomputed view associated with the aggregate table:

```
create view quarterly_store_sales_view as
select store_name, qtr, sum(dollars) as dollars
from sales, period, store
where sales.perkey = period.perkey
      and sales.storekey = store.storekey
group by store_name, qtr
using quarterly_store_sales (store_name, qtr, dollars);
```

Note: The query expression defined in the view *must match* the query expression defined in the INSERT statement.

Step 4—Mark the Precomputed View Valid

Make the precomputed view available to the query rewrite system by using one of the following SET commands:

```
set precomputed views for sales valid;
set precomputed view quarterly_store_sales_view valid;
```

The first command marks *all* views that use the Sales table as the source of their detail data valid; the second marks only the Quarterly_Store_Sales_View valid.

Step 5—Submit the Query and Note the Performance Gain

Run the query with the query rewrite system turned on and use the SET STATS INFO command to capture performance statistics and verify that the query is being rewritten. Note the considerable performance gain when the query is rewritten versus when the query rewrite system is turned off.

The query rewrite system intercepts the user's SQL and substitutes a fast scan of the small aggregate table for the three-table STARjoin normally executed. In this case, the rewritten query is the equivalent of issuing a SELECT * on the aggregate table; however, users do not need to know that the aggregate table exists, and this table can be used to silently rewrite other queries as well.

```
select store_name, qtr, sum(dollars) as total_sales
from sales, period, store
where sales.perkey = period.perkey
      and sales.storekey = store.storekey
group by store_name, qtr;

** STATISTICS ** (500) Compilation = 00:00:00.22 cp time,
00:00:00.21 time, Logical IO count=156

STORE_NAME                QTR    TOTAL_SALES
Roasters, Los Gatos        Q1_94    43011.50
San Jose Roasting Company  Q1_94    55763.25
Cupertino Coffee Supply    Q1_94    44280.75
...
Beans of Boston            Q1_96    46797.25
Olympic Coffee Company     Q1_96    54868.50
Coffee Connection           Q1_96    31697.00

** INFORMATION ** (1461) SQL statement was rewritten to use one
or more precomputed views.
** STATISTICS ** (500) Time = 00:00:00.02 cptime, 00:00:00.02 time,
Logical IO count=164
** INFORMATION ** (256) 156 rows returned.
```

Step 6—Experiment with Other Query Rewrites

Variations on the original query can also be rewritten using the same precomputed view.

Variation 1—Add an ORDER BY Clause

```
select store_name, qtr, sum(dollars) as total_sales
from sales, period, store
where sales.perkey = period.perkey
      and sales.storekey = store.storekey
group by store_name, qtr
order by total_sales desc;

STORE_NAME                QTR    TOTAL_SALES
Miami Espresso            Q2_95    61974.40
San Jose Roasting Company  Q2_95    60340.70
Olympic Coffee Company     Q3_95    60070.50
Olympic Coffee Company     Q4_95    59998.60
Beaches Brew               Q2_95    59505.75
```

Query Rewrite Case Studies

Case 1—Rewriting a STARjoin Query

This query can be rewritten to use the same table scan as the original query. The ORDER BY clause does not compute a different result set, just the same result set sorted and displayed in the specified order.

Variation 2—Group the Result Set by Fewer Columns

The following query returns the quarterly sales figures for all stores. The results are grouped by one column only—the Qtr column from the Period table. (This query also retains the ORDER BY clause introduced in the previous example.)

```
select qtr, sum(dollars) as total_sales
from sales, period
where sales.perkey = period.perkey
group by qtr
order by total_sales desc;
```

QTR	TOTAL_SALES
Q2_95	837699.75
Q4_95	822765.35
Q3_95	822268.35
Q1_96	807390.40
Q1_95	797257.60
Q4_94	782359.05
Q3_94	778795.20
Q2_94	756282.05
Q1_94	723532.35

In this case, the query rewrite system must do some GROUP BY processing to roll up the precomputed result set defined by the view (156 rows) to 9 rows—one row per quarter. However, no join of the Sales and Period tables is required; the result set can be calculated directly from the data in the aggregate table.

Variation 3—Add a Predicate to the WHERE Clause

```
select qtr, sum(dollars) as total_sales
from sales, period
where sales.perkey = period.perkey
and qtr like '%95%'
group by qtr
order by total_sales desc;
```

QTR	TOTAL_SALES
Q2_95	837699.75
Q4_95	822765.35
Q3_95	822268.35
Q1_95	797257.60

Because the WHERE clause predicate is on one of the grouping columns used to define the contents of the aggregate table, the query can be rewritten to use the aggregate table instead of the Sales and Period tables.

Variation 4—Add a RSQL Display Function and a WHEN Clause

This variation removes the WHERE clause predicate introduced in the previous example and adds an expression in the select list that uses the RANK display function. The results of the RANK function are constrained by the WHEN clause to return only the top three quarters.

```
select qtr, sum(dollars) as total_sales
       rank(total_sales) as sales_rank
from sales, period
where sales.perkey = period.perkey
group by qtr
when sales_rank <=3
order by total_sales desc;
```

QTR	DOLLARS	SALES_RANK
Q2_95	837699.75	1
Q4_95	822765.35	2
Q3_95	822268.35	3

Variation 5—Add Constraints on “Unknown” Columns

This variation is a “negative test.” The following queries *cannot be rewritten* to use the precomputed view for the Quarterly_Store_Sales table because they refer to columns not defined by the view. These “unknown” columns are shown in bold:

```
select month, qtr, sum(dollars) as total_sales
from sales, period, store
where sales.perkey = period.perkey
       and sales.storekey = store.storekey
group by month, qtr;
```

```
select store_name, store_type, sum(dollars) as total_sales
from sales, period, store
where sales.perkey = period.perkey
       and sales.storekey = store.storekey
group by store_name, store_type;
```

```
select store_name, qtr, sum(dollars) as total_sales
from sales, period, store
where sales.perkey = period.perkey
       and sales.storekey = store.storekey
       and store_type = 'Large'
group by store_name, qtr;
```

Case 2—Making Use of Explicit Hierarchies

Case 1 presented a simple example of query rewriting in which minor variations in the original query could be handled by the same aggregate table and view. The following example demonstrates the case where the existence of a functional dependency extends the range of queries that a single aggregate table can be used to rewrite.

When the precomputed view is grouped by *non-key columns* (such as Store_Name and Qtr, as in Case 1), but the user's query constrains on a column of coarser granularity than one of the grouping columns, the query rewrite system makes use of functional dependencies between the grouping column in the view and the column specified in the query. However, unless these dependencies have been declared as explicit hierarchies, the query rewrite system is not aware of them and the query cannot be rewritten.

The Query

Except for the substitution of the Year column for the Qtr column, the kind of query the administrator wants to rewrite is the same as the original query for Case 1 (see page 4-3).

```
select store_name, year, sum(dollars) as total_sales
from sales, period, store
where sales.perkey = period.perkey
      and sales.storekey = store.storekey
group by store_name, year;
```

Step 1—Create the Aggregate Table and Precomputed View

Create and load the aggregate table and create the precomputed view (grouped by the Store_Name and Qtr columns) as described in Case 1 on page 4-3.

Step 2—Create the Hierarchy

Declare that a functional dependency exists between the Qtr and Year columns in the *detail* Period table.

```
create hierarchy qtr_year
(from period(qtr) to period(year));
```

Ensure that the precomputed view is marked valid, then turn on the query rewrite system and run the query.

Step 3—Create Additional Hierarchies

Create some additional hierarchies to extend the range of query rewrites possible with the `Quarterly_Store_Sales` aggregate table. For example, create the following hierarchy, which defines *two* legal relationships:

```
create hierarchy store_region
  (from store(store_name) to store(city),
   from store (city) to market (district));
```

Now many more queries become candidates for query rewriting. Specifically, queries that group by any combination of the following columns can be rewritten:

- Period table: `Qtr` and `Year` columns
- Store table: `Store_Name` and `City` columns
- Market table (an outboard table referenced by the Store table): All columns

All of the Market table columns can be constrained because the hierarchy from `Store.City` to `Market.District` is internally defined as `Store.City` to `Store.Mktkey` (see page 3-12). Therefore, using the primary-key/foreign-key relationship between the Store and Market tables, rollups are possible from `Store.City` to any column in the Market table.

Step 4—Experiment with Other Query Rewrites

For example, all of the following queries can be rewritten:

- Sales per city per quarter:

```
select city, quarter, sum(dollars) as total_sales
from sales, period, store
where sales.perkey = period.perkey
      and sales.storekey = store.storekey
group by city, quarter;
```

- Sales per store for the Southern region:

```
select store_name, sum(dollars) as total_sales
from sales, period, store, market
where sales.perkey = period.perkey
      and sales.storekey = store.storekey
      and market.mktkey = store.mktkey
      and market.region = 'South'
group by store_name
order by total_sales desc;
```

Query Rewrite Case Studies

Case 2—Making Use of Explicit Hierarchies

- Sales per district per quarter:

```
select district, qtr, sum(dollars) as total_sales
from sales, period, store, market
where sales.perkey = period.perkey
      and sales.storekey = store.storekey
      and store.mktkey = market.mktkey
group by district, qtr;
```

- Sales per district per year:

```
select district, year, sum(dollars) as dollars
from sales, period, store, market
...;
```

- Sales per region per quarter:

```
select region, qtr, sum(dollars) as dollars
from sales, period, store, market
...;
```

- Sales per region per year:

```
select region, year, sum(dollars) as dollars
from sales, period, store, market
...;
```


Case 3—Optimizing Query Rewrites with Derived Dimensions

Case 2 showed how declared functional dependencies extend the range of query rewriting that can be done with a single precomputed view. However, to ensure that the types of queries rewritten in Case 2 achieve *optimal* rewriting performance, the administrator should create derived dimension tables, as explained in the following example.

The Queries

The following queries will be rewritten using explicit hierarchies and a precomputed view grouped by non-key columns; therefore, they both require derived dimensions to ensure optimal performance.

The first query is the same as the query presented at the beginning of Case 2; it requires a derived dimension on the Period table:

```
select store_name, year, sum(dollars) as total_sales
from sales, period, store
where sales.perkey = period.perkey
      and sales.storekey = store.storekey
group by store_name, year;
```

The second query requires derived dimensions on the Period table and Store tables:

```
select city, year, sum(dollars) as total_sales
from sales, period, store
where sales.perkey = period.perkey
      and sales.storekey = store.storekey
group by city, year;
```

Step 1—Create the Aggregate Tables

Issue CREATE TABLE statements to create derived dimensions and an aggregate table that references them. *Because the derived dimensions are the referenced tables, create them first.*

Derived Dimension for the Period Table

```
create table period_qtr
  (qtr char(5) not null,
  year int,
  primary key (qtr))
maxrows per segment 1000;
```

Note: The Qtr column is the primary key of the derived dimension.

Derived Dimension for the Store Table

```
create table derived_store (
  store_name char(30) not null,
  city char(20),
  state char(5),
  zip char(10),
  mktkey integer not null,
  primary key (store_name),
  foreign key (mktkey) references market (mktkey))
maxrows per segment 2000;
```

Note: The foreign-key/primary-key relationship between the Store table and the outboard Market table is retained in the derived Store dimension.

Aggregate Fact Table

```
create table derived_quarterly_store_sales
  (storekey char(30) not null,
  qtrkey char(5) not null,
  dollars dec(13,2),
  primary key (storekey, qtrkey),
  foreign key (qtrkey) references period_qtr (qtr),
  foreign key (storekey) references derived_store (store_name))
maxrows per segment 2000;
```

Note: The foreign keys reference the two derived dimensions. Although the same aggregate data is used in this example as in Cases 1 and 2, this version of the aggregate fact table must reference the derived dimensions instead of the detail Store and Period dimensions.

Step 2—Load the Aggregate Tables

Derived Dimensions

```
insert into period_qtr
select qtr, year from period
group by qtr, year;
```

```
insert into derived_store
select store_name, city, state, zip, mktkey
from store
group by store_name, city, state, zip, mktkey;
```

Note: Although SELECT DISTINCT queries would insert the same set of rows into these tables as the GROUP BY queries used here, the SELECT DISTINCT approach cannot be used because such query expressions are not allowed in precomputed view definitions.

Aggregate Fact Table

```
insert into derived_quarterly_store_sales
(storekey, qtrkey, dollars)
select store_name, qtr, sum(dollars) as dollars
from sales, period, store
where sales.perkey = period.perkey
and sales.storekey = store.storekey
group by store_name, qtr;
```

Step 3—Create the Precomputed Views

Derived dimensions are by definition aggregate tables; they must be linked to precomputed views.

Derived Dimensions

```
create view period_qtr_view as
select qtr, year from period
group by qtr, year
using period_qtr (qtr, year);
```

```
create view derived_store_view as
select store_name, city, state, zip, mktkey
from store
group by store_name, city, state, zip, mktkey
using derived_store (store_name, city, state, zip, mktkey);
```

Aggregate Fact Table

```
create view derived_quarterly_store_sales_view as
  select store_name, qtr, sum(dollars) as dollars
  from sales, period, store
  where sales.perkey = period.perkey
        and sales.storekey = store.storekey
  group by store_name, qtr
  using derived_quarterly_store_sales (storekey, qtrkey, dollars);
```

Step 4—Create a STAR index on the Aggregate Fact Table

Create a STAR index on the foreign-key columns (Qtrkey and Storekey) of the Derived_Quarterly_Store_Sales table.

```
create star index derived_quarterly_store_sales_star
  on derived_quarterly_store_sales (qtrkey, storekey);
```

This STARindex is the equivalent of the STARindex used to join the Sales table to the Period and Store dimensions. The new index will make it possible to STARjoin the Derived_Quarterly_Store_Sales, Period_Qtr, and Derived_Store tables.

Step 5—Create Explicit Hierarchies

Define the same set of hierarchies that were created in Step 4 for Case 2 (see page 4-8).

These functional dependencies must be declared before the aggregate table family can be used to rewrite queries that group by the Year column of the Period table and/or the City column of the Store table.

Step 6—Validate the Precomputed View

Mark the precomputed view valid by issuing the following SET commands:

```
set precomputed view derived_quarterly_store_sales_view valid;
set precomputed view period_qtr_view valid;
set precomputed view derived_store_view valid;
```

Note: Be sure to mark all three views valid (one for each table in the aggregate table family).

Now turn on the query rewrite system and note the performance gain when you run the queries on page 4-11.

Step 7—Note the Simplified SQL

Derived dimensions optimize query-rewriting performance by eliminating some extra GROUP BY processing in the generated SQL. Compare the following query rewrites for the same query.

The first rewrite lacks a derived dimension; therefore, a subquery in the FROM clause (shown in bold) must group the Qtr and Year values from the detail Period table. The results of this subquery are then joined to the aggregate table. The second rewrite uses a simple join to the Period_Qtr derived dimension, in which the grouped Qtr and Year values are precomputed.

For more information about performance gains with derived dimensions, refer to “Simplified SQL Generation” on page 3-18.

Query

```
select store_name, year, sum(dollars) as total_sales
from sales, period, store
where sales.perkey = period.perkey
      and sales.storekey = store.storekey
group by store_name, year
order by total_sales desc;
```

Generated SQL—Without a Derived Dimension

```
SELECT TABLE_2.STORE_NAME AS RBW_2, TABLE_1.COL2 AS RBW_3,
       SUM(TABLE_2.DOLLARS) AS RBW_4
FROM
  (SELECT TABLE_0.QTR AS RBW_0, TABLE_0.YEAR AS RBW_1
   FROM PERIOD AS TABLE_0
   GROUP BY TABLE_0.QTR, TABLE_0.YEAR) AS TABLE_1(COL1, COL2),
  QTR_STORE_SALES1 AS TABLE_2
WHERE TABLE_2.QTR = TABLE_1.COL1
GROUP BY TABLE_2.STORE_NAME, RBW_3
ORDER BY 3 DESC NULL FIRST;
```

Generated SQL—With a Derived Dimension

```
SELECT TABLE_0.STORE_NAME AS RBW_0, TABLE_1.YEAR AS RBW_1,
       SUM(TABLE_0.DOLLARS) AS RBW_2
FROM QTR_STORE_SALES1 AS TABLE_0, PERIOD_QTR AS TABLE_1
WHERE TABLE_0.QTR = TABLE_1.QTR
GROUP BY TABLE_0.STORE_NAME, TABLE_1.YEAR
ORDER BY 3 DESC NULL FIRST;
```

Case 4—Using Implicit Hierarchies to Rewrite Queries

Cases 1, 2, and 3 use aggregate data grouped by non-key columns. These aggregate tables perform very well for specific queries or groups of similar queries. The following example takes a different approach to query rewriting by showing the case where the administrator groups the aggregate data by foreign-key columns.

This approach is simpler—no hierarchies need be declared or derived dimensions defined, and a broad range of queries can be rewritten with the same aggregate table. However, aggregate tables of this kind are usually larger than aggregate tables grouped by non-key columns. As a result, query rewrites will improve performance but often not as much as they would if the aggregate table were grouped by specific columns.

The following example demonstrates a case where a group of users run various queries that constrain on two dimension tables—Period and Store—to calculate sales revenues over different periods of time and in different locations. This case also demonstrates the ability to rewrite queries that reference columns in an outboard table (the Market table), even though those columns are not explicitly defined in the precomputed view.

The Queries

- Sales per day for a given week:

```
select day, sum(dollars) as total_sales
from sales, period
where sales.perkey = period.perkey
      and week = 13
      and year = 1996
group by day
order by sales.perkey;
```

- Sales for each store on each day of a given month:

```
select date, store_name, sum(dollars) as total_sales
from sales, period, store
where sales.perkey = period.perkey
      and sales.storekey = store.storekey
      and month = 'MAR'
      and year = 1996
group by date, store_name
order by sales.perkey;
```

- Sales for each store type for a given month:

```
select store_type, month, sum(dollars) as total_sales
from sales, store, period
where sales.storekey = store.storekey
      and sales.perkey = period.perkey
      and month = 'MAR' and year = 1996
group by store_name, month
order by total_sales desc;
```

- Sales per city per month:

```
select city, month, sum(dollars) as total_sales
from sales, store, period
...
```

- Sales per district per year:

```
select district, year, sum(dollars) as total_sales
from sales, store, period, market
...
```

To answer all these queries, the server would normally join the tables by using the STAR index on the Sales table. The query rewrite system will substitute the aggregate fact table and its STAR index. Because the aggregate fact table is smaller, all the rewritten STARjoin queries will run faster.

Step 1—Create the Aggregate Table

Issue a standard CREATE TABLE statement to create an aggregate table that contains three columns—Perkey, Storekey, and Dollars:

```
create table store_sales
(perkey int not null, storekey int not null, dollars dec(13,2),
primary key (perkey, storekey),
foreign key (perkey) references period (perkey),
foreign key (storekey) references store (storekey))
maxrows per segment 50000;
```

Note: This aggregate table has foreign-key references to the detail Period and Store tables. The two foreign keys make up the primary key.

Step 2—Populate the Aggregate Table

Issue an INSERT INTO...SELECT statement to load the aggregate table:

```
insert into store_sales
select perkey, storekey, sum(dollars)
from sales
group by perkey, storekey;
```

Step 3—Create a STAR Index for the Aggregate Table

Create a STAR index on the key columns of the Store_Sales table:

```
create star index store_sales_star
on store_sales (perkey, storekey);
```

Note: When the Store_Sales table is used instead of the Sales table in a rewritten query that requires a join to the Period and Store tables, this STAR index will be used instead of the STAR index on the Sales table.

Step 4—Create the Precomputed View

Create a precomputed view associated with the aggregate table:

```
create view store_sales_view (perkey, storekey, dollars) as
(select perkey, storekey, sum(dollars)
from sales
group by perkey, storekey)
using store_sales (perkey, storekey, dollars);
```

Because the key columns Perkey and Storekey are used as grouping columns in the precomputed view definition, the query rewrite system can rewrite queries that constrain on *any column* in the Period and Store tables. The use of key columns as grouping columns results in rollups that use implicit hierarchies—for example, from Perkey to Year in the Period table and from Storekey to City in the Store table.

Step 5—Mark the Precomputed View Valid

Inform the query rewrite system that the precomputed view is valid by using one of the following SET commands:

```
set precomputed views for sales valid;  
set precomputed view store_sales_view valid;
```

The first command marks *all* views that use the Sales table as the source of their detail data valid; the second marks only the Store_Sales_View valid.

Step 6—Submit the Queries and Note the Performance Gain

Turn on the query rewrite system, and run the queries listed on page 4-16, as well as any other aggregate query that constrains on columns from the Period, Store, and Market dimensions.

Case 5—Rewriting Subqueries

The following example shows how a query that contains multiple query expressions (or query blocks) is rewritten.

This query is a comparison query that contains two subqueries in the FROM clause. You can use the query rewrite system to accelerate the performance of other types of subqueries as well, including correlated subqueries.

The Query and Result Set

```
select sales1.name, sales_q195, sales_q196
from
  (select e1.store_name, sum(dollars)
   from sales, store e1, period
   where sales.storekey = e1.storekey
     and sales.perkey = period.perkey
     and qtr = 'Q1_95'
   group by e1.store_name) as sales1(name, sales_q195),
  (select e2.store_name, sum(dollars)
   from sales, store e2, period
   where sales.storekey = e2.storekey
     and sales.perkey = period.perkey
     and qtr = 'Q1_96'
   group by e2.store_name) as sales2(name, sales_q196)
where sales1.name = sales2.name
order by sales_q196 desc;
```

NAME	SALES_Q195	SALES_Q196
San Jose Roasting Company	53188.70	57129.30
Beaches Brew	57152.85	55662.90
Olympic Coffee Company	52544.80	54868.50
Miami Espresso	58498.50	52043.70
East Coast Roast	41007.75	48222.50
Java Judy's	48760.50	48213.25
Cupertino Coffee Supply	48155.50	47686.50
Texas Teahouse	46484.50	47013.00
Beans of Boston	46764.50	46797.25
Moulin Rouge Roasting	46671.25	46143.75
Instant Coffee	42400.50	45642.50
Moroccan Moods	43947.50	43617.75
Coffee Brewers	42531.00	43212.50
Roasters, Los Gatos	40725.50	42045.00
Moon Pennies	30203.00	33933.00
Minnesota Roaster	33992.50	33205.00
Coffee Connection	34605.50	31697.00
The Coffee Club	29623.25	30257.00

Step 1—Create the Aggregate Table and View

This example uses the same aggregate table and precomputed view as Case 1; follow steps 1 through 4 on page 4-3.

Step 2—Run the Query

Submit the query with the query rewrite system turned on and note the performance gain.

Step 3—Run the EXPLAIN Command for the Query

Submit the query again, but this time place the EXPLAIN keyword in front of the SQL statement. The EXPLAIN output describes how the rewritten query is executed, showing the tables and indexes that are used:

```

EXPLANATION
[
- EXECUTE (ID: 0) 1 Table locks (table, type):
  (QUARTERLY_STORE_SALES, Read)
--- MERGE SORT (ID: 1) Distinct: FALSE
----- EXCHANGE (ID: 2) Exchange type: Upper Hash 1-1 Match
----- HASH 1-1 MATCH (ID: 3) Join type: InnerJoin;
----- EXCHANGE (ID: 4) Exchange type: Lower Hash 1-1 Match
----- HASH AVL AGGR (ID: 5) Log Advisor Info: TRUE, Grouping:
TRUE, Distinct: FALSE;
----- FUNCTIONAL JOIN (ID: 6) 1 tables: TABLE_0
----- BTREE SCAN (ID: 7) Table: TABLE_0, Index:
QUARTERLY_STORE_SALES_PK_IDX ,Reverse order: FALSE; Start-stop
predicate: <none>; Predicate: (TABLE_0.QTR) = ('Q1_96')
----- EXCHANGE (ID: 8) Exchange type: Lower Hash 1-1 Match
----- HASH AVL AGGR (ID: 9) Log Advisor Info: TRUE, Grouping:
TRUE, Distinct: FALSE;
----- FUNCTIONAL JOIN (ID: 10) 1 tables: TABLE_2
----- BTREE SCAN (ID: 11) Table: TABLE_2, Index:
QUARTERLY_STORE_SALES_PK_IDX ,Reverse order: FALSE; Start-stop
predicate: <none>; Predicate: (TABLE_2.QTR) = ('Q1_95')
]

```

The output reveals that the only table and index used to execute this rewritten query are the Quarterly_Store_Sales aggregate table and its primary key index. The names TABLE_0 and TABLE_2 in this output are aliases for the Quarterly_Store_Sales table that the query rewrite system assigns when it generates the rewritten SQL for each subquery.

Query Rewrite Case Studies
Case 5—Rewriting Subqueries

In other words, the rewritten query still contains subqueries in the FROM clause, but each subquery has been rewritten as a B-TREE index scan of the Quarterly_Store_Sales table with the appropriate predicate on the Qtr column. The only join required is a join of the two intermediate tables that result from the rewritten subqueries. The three-table joins within the subqueries have been eliminated.

For more information about the output of the EXPLAIN command, refer to the *Warehouse Administrator's Guide*.

Case 6—Rewriting a Query That Calculates Averages

Although you cannot use the AVG set function in a precomputed view definition (see page 3-6), queries that calculate averages can be rewritten as long as a precomputed view exists that stores SUM and COUNT values for the column to be averaged.

This case presents a very simple example of a rewritten query that contains an AVG function.

The Query and Result Set

```
select perkey, int(avg(dollars)) as day_avg
from sales
where perkey between 1 and 31
group by perkey
order by perkey;
```

PERKEY	DAY_AVG
2	105
3	93
4	96
5	101
6	115
7	87
8	103
9	100
10	78
...	

Step 1—Create the Aggregate Table

Create an aggregate table that contains three columns:

```
create table sum_count_sales
(perkey int,
 sum_sales dec(13,2),
 count_sales dec(13,2));
```

Step 2—Populate the Aggregate Table

Populate the aggregate table with an INSERT INTO...SELECT statement:

```
insert into sum_count_sales
select perkey, sum(dollars), count(dollars)
from sales
group by perkey;
```

Query Rewrite Case Studies

Case 6—Rewriting a Query That Calculates Averages

Step 3—Create the Precomputed View

Create a precomputed view associated with the aggregate table:

```
create view sum_count_view (perkey, sum_sales, count_sales) as
  select perkey, sum(dollars), count(dollars)
  from sales
  group by perkey
  using sum_count_sales (perkey, sum_sales, count_sales);
```

Step 4—Mark the Precomputed View Valid

Inform the query rewrite system that the precomputed view is valid:

```
set precomputed view sum_count_view valid;
```

Step 5—Submit the Queries and Note the Performance Gain

Turn on the query rewrite system and run the query on page 4-23. The query will be rewritten by using the precomputed SUM and COUNT values in the Sum_Count_Sales table to calculate the average sales figures.

Using the Advisor

The Red Brick Warehouse Advisor is used to analyze the usefulness of precomputed views that exist in your database and to suggest new precomputed views that can increase the query performance of your system. This chapter contains the following sections:

- Advisor Overview
- Configuring the Advisor Logging System
- Querying the Advisor
- Interpreting the Results of Advisor Queries
- Understanding the BENEFIT Column
- Advisor System Table Column Descriptions
- Checklist of Advisor Tasks

Advisor Overview

The Advisor—an integral part of the Red Brick Vista option—aims in figuring out the best aggregate tables for your database, whether those tables currently exist or not. Since the Advisor knows exactly what types of queries can be rewritten with the Red Brick Vista query rewriter, it suggests the exact precomputed views to build in your database. This is a powerful tool in gaining the best performance from your Red Brick Warehouse database.

There is a cost to every precomputed view, as well as a benefit to having them exist in your database. The Advisor helps with the cost-benefit analysis of improving the query performance of your database with precomputed views.

The Advisor provides a facility to log activity of aggregate queries against a database. From the logged queries, you can analyze two categories: 1) the use of existing aggregates in the database and 2) evaluate potential new aggregates to create that, with the query rewriter, can improve query performance.

Analysis of Query Patterns

The goal of the Advisor is to analyze query patterns and see if you have created the appropriate precomputed views for your database. The longer and more representative a sample of query patterns that are logged, the more accurate the results of Advisor queries.

Advisor System Tables

You analyze the information in the Advisor log by querying the Advisor system tables. The Advisor system tables are created with the other system tables when a database is created. They provide information necessary to understand the use of existing precomputed views and also guide you in the creation of new precomputed views. The two Advisor system tables are:

- RBW_PRECOMPVIEW_CANDIDATES Table
- RBW_PRECOMPVIEW_UTILIZATION Table

For descriptions of each column in the Advisor system tables, refer to “Advisor System Table Column Descriptions” on page 5-24.

Advisor Log Files

The Advisor log files store information about the precomputed views in a database. They are created when logging is started, either at system startup or manually, depending on the configuration.

The log files store two types of information about precomputed views:

- Information about precomputed views that exist in your database.
- Information about precomputed views that do not exist but would provide query-performance benefits if created.

The Advisor analyzes the log files when you query the Advisor system tables, as explained on page 5-8.

Configuring the Advisor Logging System

There are two steps to configuring your system to log queries for the Advisor:

1. Create the Advisor log file (ADMIN ADVISOR_LOGGING ON or ALTER SYSTEM START ADVISOR_LOGGING).
2. Enable Advisor query logging (OPTION ADVISOR_LOGGING ON or SET ADVISOR_LOGGING ON).

For the syntax of the various Advisor logging commands, refer to the *SQL Reference Guide*. For more information about logging, refer to the *Warehouse Administrator's Guide*.

Creating the Advisor Log Files

If the ADMIN ADVISOR_LOGGING parameter is set to ON in the *rbw.config* file, the Advisor log file is created upon system startup. You can also create the log file manually with the ALTER SYSTEM START ADVISOR_LOGGING command. The log file is created in the directory specified by the ADMIN ADVISOR_LOG_DIRECTORY parameter or in the *redbrick_dir/logs* directory if that parameter is not specified.

Logging Queries

After the Advisor log file is created, you can enable query logging by setting the `OPTION ADVISOR_LOGGING` parameter to `ON` or `ON_WITH_CORR_SUB` in the `rbw.config` file. You can also use the `SET ADVISOR LOGGING` command to enable or disable Advisor query logging for a session.

Rewritten Queries

When Advisor query logging is enabled, the Advisor logs all queries that are rewritten to access data in precomputed views. The purpose of logging these queries is to provide data to help you analyze the benefit of your aggregate tables. When you query the `RBW_PRECOMPVIEW_UTILIZATION` table, the query reads the log files and provides statistics on the actual use during the time period you specify in the query.

Note: The Advisor logs queries that are rewritten against any valid precomputed views, including views that are forced into a valid state with `SET PRECOMPUTED VIEW...VALID` commands. Similarly, the Advisor also logs queries rewritten against views that are marked invalid with `SET PRECOMPUTED VIEW...INVALID` commands if `USE INVALID PRECOMPUTED VIEWS` is set to `ON`.

Candidate Views

The Advisor also logs views that, if they existed, would have been used for rewriting queries. These potential views are called candidate views, and can be seen by querying the `RBW_PRECOMPVIEW_CANDIDATES` table.

Correlated Subqueries

The `ON_WITH_CORR_SUB` state of the `SET ADVISOR LOGGING` command and `OPTION ADVISOR_LOGGING` parameter logs correlated subqueries to the Advisor log files. It is useful to log correlated subqueries because they can be rewritten by the Red Brick Vista query rewrite system. Correlated subqueries execute the same query with different values multiple times (potentially a very large number of repeated queries). Therefore, if such a query can be rewritten, the performance improvement is multiplied by the number of times the correlated subquery is executed. If that number is, for example, 1,000,000, that can be a huge performance boost.

In general, it is a good idea to log correlated subqueries. There are, however, two things to consider when deciding whether to log correlated subqueries:

- Log file size
- Data skew

Log File Size

If correlated subqueries are common in your database, then logging with the ON_WITH_CORR_SUB state will cause your log file to grow much larger much faster. This not only takes extra space, but also adds processing time when querying the Advisor system tables.

Data Skew

Correlated subqueries can potentially skew your Advisor results because each subquery of the correlated subquery is logged as a separate REFERENCE_COUNT, and a correlated subquery executes a separate subquery for each row in the outer query that is input into the correlated subquery.

Queries That Are Rewritten But Not Logged

When a query is rewritten, a record is sent to the log file. The record is then used to keep track of which precomputed views have been used and to log candidate views.

There is, however, a class of queries that could be rewritten if the proper precomputed view existed, but do not get logged: queries that contain a table or a subquery that is not related (via a primary key/foreign key relationship) to the other tables in the query. The following is an example of such a query from the Aroma database:

```
set cross join on;
select market.hq_state as hq_state,
       sum(sales.dollars) as sum_dollars,
       sum_dollars/sum_x.total_sales
from sales, market, store, (select sum(dollars)
                             from sales) as sum_x(total_sales)
where sales.storekey = store.storekey
and   store.mktkey = market.mktkey
and   market.hq_state <> 'CA'
group by hq_state, sum_x.total_sales;
```

The subquery in the FROM clause of this query has no primary key/foreign key relationship with the other tables in the query, so this would not be logged by the Advisor. If you had a precomputed view defined with the following CREATE TABLE, INSERT INTO...SELECT, and CREATE VIEW...USING statements, however, this query could be rewritten to use that precomputed view:

```
create table simple_table (hq_state char(20),
    hq_city char(20),
    year integer,
    month character(5),
    sum_dollars dec(13,2));

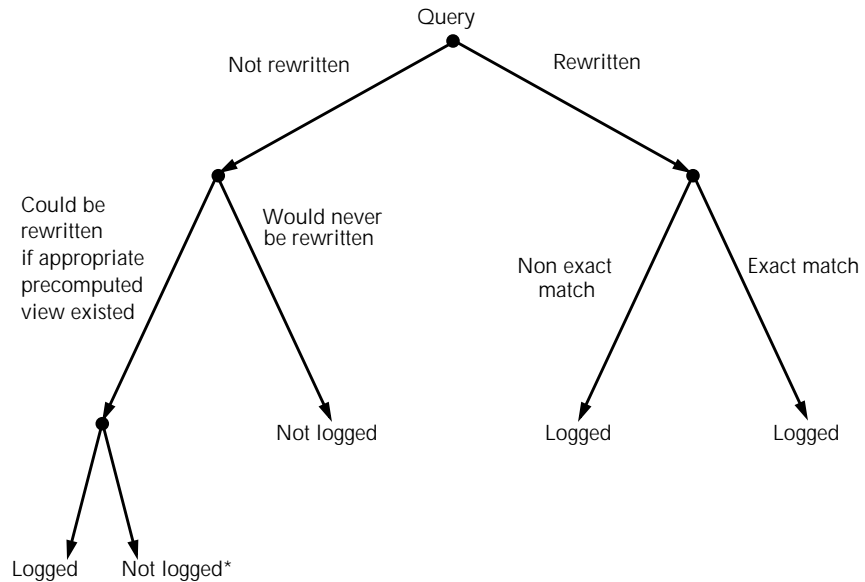
create view simple_view as
    (select market.hq_state as hq_state,
        market.hq_city as hq_city,
        period.year as year, period.month as month,
        sum(sales.dollars) as sum_dollars
    from sales, market, period, store
    where sales.perkey = period.perkey
        and sales.storekey = store.storekey
        and store.mktkey = market.mktkey
    group by hq_city, hq_state, month, year)
using simple_table (hq_state, hq_city, year, month, sum_dollars);

insert into simple_table
    (hq_state, hq_city, year, month, sum_dollars)
(select market.hq_state as hq_state, market.hq_city as hq_city,
    period.year as year, period.month as month,
    sum(sales.dollars) as sum_dollars
from sales, market, period, store
where sales.perkey = period.perkey
    and sales.storekey = store.storekey
    and store.mktkey = market.mktkey
group by hq_city, hq_state, month, year);

set precomputed view simple_view valid;
set precomputed view query rewrite on;
```

Note: This class of query is only not logged if no precomputed view for it exists; if the proper precomputed views exist and are valid, the Advisor logs their use.

The following figure shows the relationship of queries that are rewritten and not rewritten to queries that are logged:



* Represents the class of queries that could be rewritten but the candidate views are not logged.

For information on what constitutes an exact match, refer to “NON_EXACT_MATCH_COUNT Column” on page 5-11.

Setting the ACCESS_ADVISOR_INFO Task Authorization

All users with the DBA system role have the necessary privileges to query the Advisor system tables. With the Enterprise Control and Coordination option, you can authorize a user or role to query the Advisor system tables by granting the ACCESS_ADVISOR_INFO task authorization with the GRANT command.

The following example grants authorization to query the Advisor system tables to the database user Rieko:

```
grant access_advisor_info to rieko;
```

Note: To query the Advisor, you also need read privileges on all tables that are referenced in the log records for the detail table you are constraining on.

For more information about privileges and role-based security, refer to the *Warehouse Administrator's Guide*.

Defining Valid Hierarchies

It is important to define any valid hierarchies in your schema before you begin logging queries with the Advisor. This is especially important for the RBW_PRECOMPVIEW_CANDIDATES table. If hierarchies are defined before you begin logging, the Advisor can recommend candidate views that take advantage of these hierarchies. For example, if the Period table has a hierarchy from the Qtr column to the Year column, the Advisor can recommend views with the knowledge that queries that can be rewritten with a precomputed view grouped by Year can also be rewritten with a precomputed view grouped by Qtr.

For information on hierarchies, refer to “Rollups and Hierarchies” on page 2-6.

Querying the Advisor

To analyze the queries that are logged in the Advisor log files, you submit SQL queries against the Advisor system tables. When queries are issued against the RBW_PRECOMPVIEW_CANDIDATES and RBW_PRECOMPVIEW_UTILIZATION tables, the Advisor analyzes the information in the log files and collects statistics on the value of each view or candidate view.

Inserting the Results of an Advisor Query Into a Table

Each Advisor query examines the log files and performs extensive analysis on the information in them. These queries can take a considerable amount of time to process, particularly queries of the RBW_PRECOMPVIEW_CANDIDATES table. If your database is extremely large and if there are a large number of candidate views, they can take many hours or even days. One way to avoid doing the processing multiple times is to create a table or a temporary table and insert the results of a query into the new table. Then you can query the new table to get the Advisor information.

To insert results of an Advisor query into a table, first you must create the table or temporary table to insert into and then run an Advisor query with an INSERT statement.

Note: A temporary table is created with the CREATE TEMPORARY TABLE command. If you create a temporary table to insert into, the table and its contents are only accessible in the session in which the temporary table is created; the temporary table and its contents are automatically dropped when you end your session. For more information on temporary tables, refer to the *SQL Reference Guide*.

Creating and Populating the CANDIDATE_TEMP Table

Use the following procedure to create a table to store the data from a query against the RBW_PRECOMPVIEW_CANDIDATES Advisor system table, insert data from the Advisor query into the new table, and query the new table to view the results of the Advisor query.

1. Create a table that contains all of the columns of the Advisor system table as in the following example:

```
create table candidate_temp (  
    detail_table_name char (128),  
    start_date timestamp,  
    end_date timestamp,  
    aggr_elapsed_time int,  
    reference_count int,  
    sample_view_name char (128),  
    size int,  
    reduction_factor float,  
    benefit float,  
    name char (128),  
    seq int,  
    text char (1024) ) ;
```

2. Insert data from an Advisor query into the new table as in the following example:

```
insert into candidate_temp  
select * from rbw_precompview_candidates  
where detail_table_name = 'SALES';
```

This operation inserts all of the information about candidate views that can be created on the Sales table into the table Candidate_Temp. This INSERT operation might take a large amount of time, depending on the size of the database and the Advisor log files being analyzed.

Creating and Populating the UTILIZATION_TEMP Table

Use the following procedure to create a table to store the data from a query against the RBW_PRECOMPVIEW_UTILIZATION Advisor system table, insert data from the Advisor query into the new table, and query the new table to view the results of the Advisor query.

1. Create a table that contains all of the columns of the Advisor system table as in the following example:

```
create table utilization_temp (  
    detail_table_name char (128),  
    start_date timestamp,  
    end_date timestamp,  
    name char (128),  
    size int,  
    reduction_factor float,  
    benefit float,  
    non_exact_match_count int,  
    rollup_count int) ;
```

2. Insert data from an Advisor query into the new table as in the following example:

```
insert into utilization_temp  
    select * from rbw_precompview_utilization  
    where detail_table_name = 'SALES';
```

This operation inserts all of the information about views currently in your database that reference the Sales table into the table Utilization_Temp.

After the Utilization_Temp table is populated, you can query it to find out how well your existing views are being used.

Querying the RBW_PRECOMPVIEW_UTILIZATION Table

You query the RBW_PRECOMPVIEW_UTILIZATION table to determine how often the precomputed views that exist in your database are being used and to determine the overall benefit they are providing. This section describes some rules you need to know when querying the table and some details about the NON_EXACT_MATCH_COUNT column.

For a description of each column in the RBW_PRECOMPVIEW_UTILIZATION table, refer to page 5-26.

Rules for Querying the RBW_PRECOMPVIEW_UTILIZATION Table

The following rules apply to queries of the RBW_PRECOMPVIEW_UTILIZATION table:

- The `DETAIL_TABLE_NAME` column must be constrained to indicate which table the precomputed views reference. Exactly one detail table must be specified per query.
- You can (optionally) constrain on the `START_DATE` and `END_DATE` columns to limit the scope of the query to a particular time period.
- No other columns can be constrained.

Note: If you insert the results of an Advisor query into a table like the `Utilization_Temp` table shown on page 5-10, there are no restrictions as to what columns can be constrained on that table; any columns can be constrained.

NON_EXACT_MATCH_COUNT Column

The `NON_EXACT_MATCH_COUNT` column tells how many times a view in the database was used to calculate answers to questions where some additional aggregation was needed. If the count in this column is high, it suggests that other precomputed views might help your query performance.

An *exact match* is when a query is answered by a precomputed view without performing additional aggregation on the precomputed view. There can still be some predication on the query (for example, a `WHERE` clause or `HAVING` clause) and there can be some formatting (for example, `ORDER BY` clause), but no extra aggregation (for example, `GROUP BY`, `SUM`, `MIN`, `MAX`). In other words, an exact match is considered to be some subset of the rows in the precomputed view.

Example

Assume you have a detail table with a granularity of days, a precomputed view defined on that table with a granularity of months, and the detail table and the precomputed view both contain the sum of dollars. If you asked many questions about how many dollars were generated for a year, those questions can be answered by the month table. They can be answered, but not directly; a further aggregation needs to be computed first. Each time the precomputed view is accessed to answer a question about the sum of dollars for a year, the `NON_EXACT_MATCH_COUNT` column is incremented by one.

If the answer to the question is not an exact match of what is in the precomputed view, the column is incremented. This includes when additional aggregation is performed and when a join to another table occurs.

Querying the RBW_PRECOMPVIEW_CANDIDATES Table

You query the RBW_PRECOMPVIEW_CANDIDATES table to help determine what precomputed views to create in your database. Queries on the table perform a detailed analysis of the query history stored in your Advisor log files and the recommendations for candidate views are based on that history.

For a description of each columns in the RBW_PRECOMPVIEW_CANDIDATES table, refer to page 5-24.

Rules for Querying the RBW_PRECOMPVIEW_CANDIDATES Table

The following rules apply to queries against the RBW_PRECOMPVIEW_CANDIDATES Advisor system table:

- The DETAIL_TABLE_NAME column must be constrained.
- You can (optionally) constrain on the START_DATE, END_DATE, and SAMPLE_VIEW_NAME columns.
- No other columns can be constrained.

Note: If you insert the results of an Advisor query into a table like the Candidate_Temp table shown on page 5-9, there are no restrictions as to what columns can be constrained on that table; any columns can be constrained.

SAMPLE_VIEW_NAME Column

The purpose of the sample view is to allow you to perform your Advisor analysis on a smaller set of data to improve the performance of Advisor queries.

When you constrain on the SAMPLE_VIEW_NAME column of the RBW_PRECOMPVIEW_CANDIDATES table, the scope of the Advisor query is limited to the view name referenced in the column. A sample view must meet the following requirements:

- The sample view must map to a subset of the rows in the detail table.
- The sample view must have a column corresponding to each of the columns in the detail table.
- The datatypes of the columns in the sample view must exactly match those in the detail table.

Additionally, when creating a sample view, try to create a view that contains a representative sample of your data. Beware of creating a view that has a highly skewed sample of your data. This is very database-specific, and the only way to know if your data is not skewed is to know the data in your database. If your sample view definition does contain a high degree of data skew, the results from your Advisor queries will also be skewed, making some views look better than they really are and others look worse.

In order to constrain on the `SAMPLE_VIEW_NAME` column, a view must exist on the detail (base) table. For example:

```
create view subset_of_detail_table as select * from sales
      where perkey between 1 and 100;
```

Note: This is a regular view, not a precomputed view; there is no `USING` clause on the `CREATE VIEW` statement and there is no precomputed table.

It is possible, however, to create a view that does not perform well. For example, if you have a 1 billion row detail table and you create a view that ends up performing a table scan on the 1 billion row table, then performance of that Advisor query might be poor. In cases like this, you can create a table and then populate the table with an `INSERT` statement that inserts a subset of the data in the detail table, for example, 100,000 rows, into the new table. Then you create a view on the new table and use that view to constrain on in your Advisor query.

Whether you create a table with a subset of the rows in the detail table or if you simply create a view that defines a subset of the rows in the detail table, it is very important that the table has the appropriate indexes defined on it, particularly `STAR` indexes. The Advisor performs many query operations when analyzing the data in the log files, and proper indexing is essential for good performance.

Note: When the `SAMPLE_VIEW_NAME` column is constrained in an Advisor query, the values in the `SIZE` and `REDUCTION_FACTOR` columns are based on the size of the sample view, not the detail table.

Example of RBW_PRECOMPVIEW_CANDIDATES Query

This example queries the Candidate_Temp table that is described on page 5-9. You can query the RBW_PRECOMPVIEW_CANDIDATES table directly, but the processing on it is done for each query. You can use the following query to inspect the relative value of the candidate views that have been generated for the detail table Sales:

```
RISQL> select substr(detail_table_name, 1,10) as TABLE_NAME,  
> size, reference_count, benefit  
> from candidate_temp  
> where detail_table_name = 'SALES';
```

TABLE_NAME	SIZE	REFERENCE_C	BENEFIT
SALES	13871	2	112140.00
SALES	1450	6	547928.00
SALES	30992	2	77898.00
SALES	66759	2	6364.00
SALES	69941	3	0.00
SALES	69941	1	0.00
SALES	69941	0	0.00

```
RISQL>
```

If you want to see the text of the candidate view that would have a size of 1450 rows as seen in the previous Advisor query, enter the following query:

```
RISQL> select text  
> from candidate_temp  
> where detail_table_name = 'SALES'  
> and size = 1450;  
TEXT
```

```
SELECT TABLE_2.PERKEY AS RBW_0, TABLE_0.PROMOKEY AS RBW_1,  
SUM(TABLE_1.DOLLARS) AS RBW_2 FROM PROMOTION AS TABLE_0,  
SALES AS TABLE_1, PERIOD AS TABLE_2  
WHERE TABLE_1.PROMOKEY = TABLE_0.PROMOKEY  
AND TABLE_1.PERKEY = TABLE_2.PERKEY  
GROUP BY TABLE_2.PERKEY, TABLE_0.PROMOKEY;
```

The table that is represented by this query in the Text column represents the contents of a precomputed view. If you created the precomputed view with this information in it, you will speed up the processing of queries that ask for the sum of dollars by promotion and by time period, as well as speeding up the processing on queries that ask for a subset of what comprises the precomputed view.

To create the precomputed view, you must create a table, insert the data into the table (this becomes the precomputed table), create a precomputed view using the precomputed table, and then mark the precomputed view valid.

The following steps show how to create, populate, and make available the precomputed view recommended by the Advisor in this case:

Step 1—Create the Aggregate Table

Issue a CREATE TABLE statement to create the aggregate table:

```
create table sales_promo_period (  
  perkey int not null,  
  promokey int not null,  
  dollars_sum dec(7,2),  
  primary key (perkey, promokey),  
  foreign key (perkey) references period (perkey),  
  foreign key (promokey) references promotion(promokey));
```

Step 2—Populate the Aggregate Table

Issue an INSERT INTO...SELECT statement to insert the results of the query the Advisor identified as a candidate view into the new precomputed table:

```
insert into sales_promo_period (  
  SELECT TABLE_2.PERKEY AS RBW_0, TABLE_0.PROMOKEY AS RBW_1,  
  SUM(TABLE_1.DOLLARS) AS RBW_2  
  FROM PROMOTION AS TABLE_0, SALES AS TABLE_1, PERIOD AS TABLE_2  
  WHERE TABLE_1.PROMOKEY = TABLE_0.PROMOKEY  
  AND TABLE_1.PERKEY = TABLE_2.PERKEY  
  GROUP BY TABLE_2.PERKEY, TABLE_0.PROMOKEY );
```

Step 3—Create Any Needed Indexes

At this point, create any indexes on the new table that you need. This is especially important if the precomputed table is large. For example, create the following STAR index and drop the primary key B-TREE index that was automatically created with the table:

```
create star index sales_promo_period_star  
  on sales_promo_period (perkey, promokey);  
  
drop index sales_promo_period_pk_idx;
```

The STAR index now acts as the primary key index.

Step 4—Create the Precomputed View

Issue a CREATE VIEW statement with a USING clause to create a precomputed view associated with the aggregate table:

```
create view sales_promo_period_view as (  
  SELECT TABLE_2.PERKEY AS RBW_0, TABLE_0.PROMOKEY AS RBW_1,  
         SUM(TABLE_1.DOLLARS) AS RBW_2  
  FROM PROMOTION AS TABLE_0, SALES AS TABLE_1, PERIOD AS TABLE_2  
  WHERE TABLE_1.PROMOKEY = TABLE_0.PROMOKEY  
         AND TABLE_1.PERKEY = TABLE_2.PERKEY  
  GROUP BY TABLE_2.PERKEY, TABLE_0.PROMOKEY )  
using sales_promo_period (perkey, promokey, dollars_sum);
```

Step 5—Mark the Precomputed View Valid

Mark the precomputed view valid and turn on the query rewrite system:

```
set precomputed views for sales valid;  
set precomputed view query rewrite on;
```

Step 6—Submit the Query and Note the Performance Gain

Now the precomputed view is available for query rewriting:

```
RISQL> select sum(dollars)  
from sales natural join promotion natural join period  
where promo_type = 900;  
** STATISTICS ** (500) Compilation = 00:00:00.12 cp time,  
00:00:00.11 time, Logical IO count=99  
  
7200.00  
** INFORMATION ** (1462) SQL statement was rewritten to use one or  
more precomputed views.  
** STATISTICS ** (1457) EXCHANGE (ID: 2) Parallelism over 1 times  
High: 2 Low: 2 Average: 2.  
** STATISTICS ** (1457) EXCHANGE (ID: 5) Parallelism over 1 times  
High: 1 Low: 1 Average: 1.  
** STATISTICS ** (1457) EXCHANGE (ID: 7) Parallelism over 1 times  
High: 1 Low: 1 Average: 1.  
** STATISTICS ** (500) Time = 00:00:00.18 cp time, 00:00:00.23 time,  
Logical IO count=99  
** INFORMATION ** (256) 1 rows returned.  
RISQL>
```

The system automatically used the precomputed view to rewrite this query, improving its performance.

Interpreting the Results of Advisor Queries

There is always a cost-benefit trade-off in creating precomputed views. The cost is in disk space, time to create, time to load, and time to administer. The benefit is better query performance. Users always favor faster performance. The warehouse administrator must evaluate this trade-off and decide what precomputed views to create and/or remove, and the Advisor is a tool to help make those decisions.

The three most important columns for interpreting Advisor query results are BENEFIT, SIZE, and REFERENCE_COUNT.

BENEFIT Column

When you interpret the BENEFIT column, remember that the larger the number, the greater the benefit for that view. Compare the BENEFIT column values with other values in the same Advisor run. The numbers are not normalized, so if you have one run from a year ago that captured one week of data and another run from this month that captured the whole month of data, the numbers are not comparable *with each other*. They are, however, comparable with the numbers for other views from the same run.

For more information on the benefit column, refer to “Understanding the BENEFIT Column” on page 5-19.

SIZE and REDUCTION_FACTOR Columns

The SIZE column specifies how many rows are in the precomputed view or the candidate view. A small number means the precomputed table is small, and small tables are inexpensive. The size in relation to the detail table, however, is very important. The REDUCTION_FACTOR column gives that ratio, but it is also useful to look at the raw numbers that make up the ratio.

For example, suppose your detail (fact) table has one billion rows and a query of the RBW_PRECOMPVIEW_CANDIDATES table shows a candidate view with a high reference count and a size of 1,000,000 rows. The reduction factor is:

$$1,000,000,000 / 1,000,000 = 1,000$$

So even though this precomputed view would contain a million rows, it is 1,000 times smaller than the detail table, and that is an excellent reduction. Any reduction is significant, but in general, a reduction greater than 10 indicates that a view that might is probably worthwhile.

If a precomputed view has no rows (SIZE equals 0), the value in the REDUCTION_FACTOR column is 0. This might be an indication that no rows have been inserted into the aggregate table.

Note: Also, if you are constraining on the SAMPLE_VIEW_NAME column (RBW_PRECOMPVIEW_CANDIDATES table), the REDUCTION_FACTOR and SIZE results are all in relation to the sample view, not the detail table.

REFERENCE_COUNT Column

The REFERENCE_COUNT column specifies how many times a precomputed view was used (RBW_PRECOMPVIEW_UTILIZATION table) or could have been used (RBW_PRECOMPVIEW_CANDIDATES table). If this number is small, it generally indicates that the view is not very good for your database. There might be cases when it is justified, though, even with a small reference count. For example, the queries that use the view might be run by the CEO of your company and she is the one who provides next year's funding for the data warehouse.

If the reference counts are high in the RBW_PRECOMPVIEW_UTILIZATION table, it means the precomputed views are being used. If the reference counts are low, consider dropping some of them, particularly the ones that are particularly large or difficult to maintain.

Combining the Results

A "good" number in any particular column by itself is not a compelling reason to create a candidate view or conclude that an existing view is being well-used. Only when you look at the numbers together can you tell the value of a given view. For example, if a view has a high benefit and a high reference count, but there is another view that can answer the same queries, as well as other queries, and that view is only marginally larger, then that means that the other view is probably the one to create. You need to look at the results together, in the context of all of the views for a given detail table. In general, though, views that have high reference counts and small sizes are probably good, low-cost views to precompute.

It is also important that you have a representative sample of queries in the Advisor log. If the main users of the database were all at an off-site meeting during the time the query history was captured, then it is not a representative sample. The longer the time period logged, the more representative is the sample and the more accurate are the Advisor results.

Understanding the BENEFIT Column

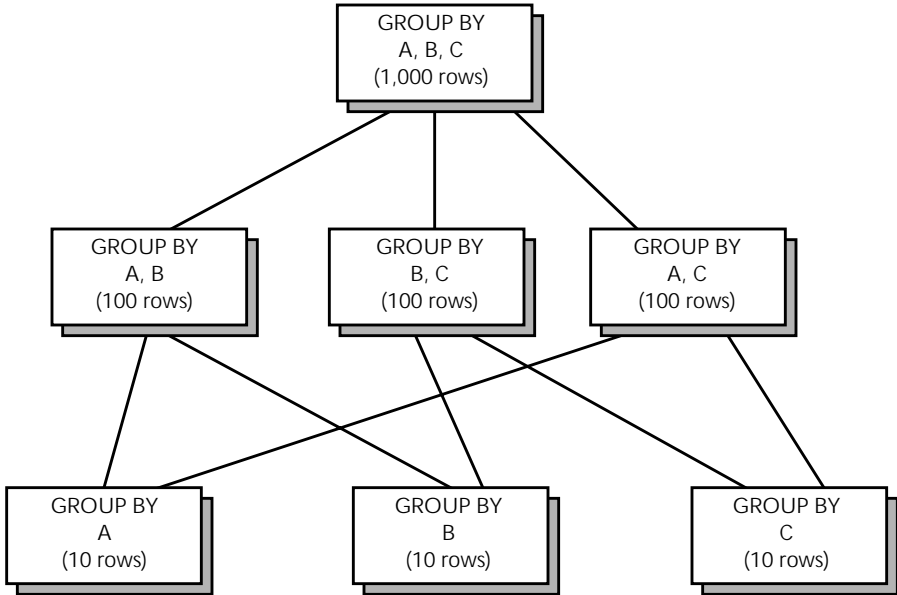
The BENEFIT column of the two Advisor system tables provides a number that indicates the benefit of a precomputed view based on other views that are available. In general, the higher the benefit, the more useful the precomputed view is for your database. The benefit is calculated based on your query history, so it is vital that there be a representative history in the Advisor log files.

How the BENEFIT Column Is Calculated

The BENEFIT column calculation is based on:

- The reference count.
- The number of rows that are saved by using the precomputed view instead of the detail table.
- Other precomputed views that could be used to answer the same questions.

Consider a database with seven precomputed views that all contain the aggregation *sum(dollars)*, but each one groups on the different columns as follows:



Notice that any query that can be answered by A can also be answered by AB, AC, or ABC. Similarly, any query that can be answered by B can also be answered by AB, BC, or ABC; and any query that can be answered by C can also be answered by BC, AC, or ABC. Because the precomputed views on the bottom level (A, B, and C) have fewer rows, these views will provide the best performance. However, they are also the most limited because they will only answer questions that are based on a single grouping column. The precomputed view on the top level (grouped by A, B, and C) can answer the widest range of queries, but it is also 100 times larger than the views on the bottom level.

The algorithm that calculates the benefit (BENEFIT column) considers the sizes of the precomputed views (SIZE column) and the number of rows saved by processing the query using that view; the views' relationships to each another; and the number of times the views could have been used (REFERENCE_COUNT column) to calculate the overall benefit. In the RBW_PRECOMPVIEW_UTILIZATION table, the benefit refers to how often the views that exist in your database could have been used, based on the query history. In the RBW_PRECOMPVIEW_CANDIDATES table, the benefit refers both to views that exist and to views that do not yet exist.

What the Numbers Mean

The numbers output in the BENEFIT column signify the number of rows that do not need to be processed, given the precomputed view corresponding to that row of the Advisor query. In other words, if a view has a benefit of 100,000, then the existence that view saves the database from processing 100,000 rows that it would have to process without the view.

Note: The calculation of the BENEFIT column does not take into account any indexes that exist on the detail table; it assumes a full table scan.

Uniform Probability

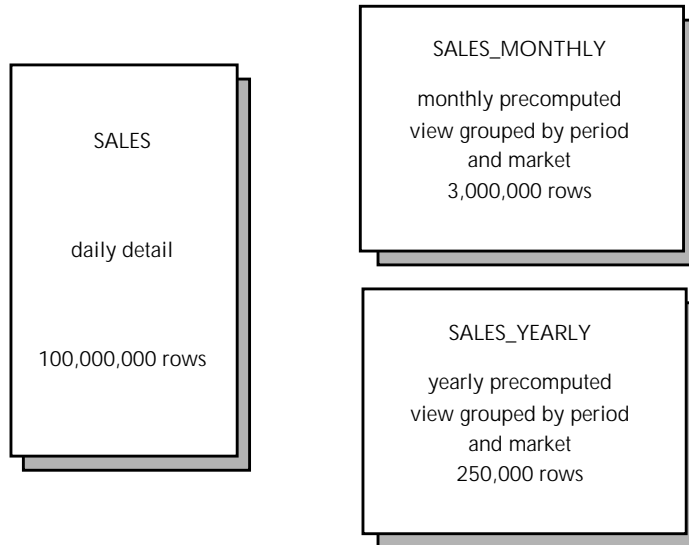
When the SET UNIFORM PROBABILITY FOR ADVISOR command is set to ON, the Advisor does not scan the log files when calculating the value for the BENEFIT column in the RBW_PRECOMPVIEW_UTILIZATION table; instead, it assumes that all precomputed views have been accessed an equal number of times and therefore does not consider the value of REFERENCE_COUNT in determining the value of the BENEFIT column. This saves time when processing an Advisor query of the RBW_PRECOMPVIEW_UTILIZATION table and is useful in the following situations:

- You are not concerned with the number of times the precomputed views have been accessed (REFERENCE_COUNT).
- The log file is excessively large and your Advisor query is taking too long.
- Your log history has skewed data in it.

For example, if you have run a few test queries 1,000 times each in testing (and logging was enabled during that time), the reference counts on the precomputed views accessed in those queries will not reflect normal usage.

Example

Consider a daily sales table with monthly and yearly precomputed views as shown in the following figure:



Consider also the following results from a query of the RBW_PRECOMPVIEW_UTILIZATION table:

```
RISQL> select substr(detail_table_name, 1,10) as TABLE_NAME,  
>      size, reference_count, benefit  
> from rbw_precompview_utilization  
> where detail_table_name = 'SALES';
```

TABLE_NAME	SIZE	REFERENCE_C	BENEFIT
SALES	3000000	1000	998994000000
SALES	250000	2	5500000

Any query that could be answered by the yearly table can also be answered by the monthly table (by adding up all the months in a year, for example), as long as there is a valid hierarchy between month and year. Because the reference count is only 2 on the yearly table, and because the benefit is low compared with the benefit for the monthly table (a factor of 6,000), it would probably be sufficient to create only the monthly table in this case.

When evaluating the benefit numbers in Advisor queries, consider the following:

- The higher the REFERENCE_COUNT, the higher the benefit.
- The longer the period of time covered in the query history, the higher the benefit tends to be.
- The smaller the SIZE, the higher the benefit.

If you analyze the Advisor log over a long period of time, the absolute numbers in the BENEFIT column tend to get larger because the more queries that are run, the more rows that are processed. These numbers provide guidelines to aid in your decisions about which views to create. There are no definitive “best” answers, but instead sets of possible “good” answers that the you, the DBA, must evaluate based on the specific needs and unique environment of your site.

Advisor System Table Column Descriptions

This section provides the names, the datatypes, and descriptions of each column in the Advisor system tables.

RBW_PRECOMPVIEW_CANDIDATES Table

The RBW_PRECOMPVIEW_CANDIDATES table contains information necessary to analyze the benefits of creating new precomputed views to help the performance of certain queries. This information can also be used to make decisions on which precomputed views to create.

This table contains one row for each potential candidate view based on the queries that are logged and one row for each existing view. If the SQL in the TEXT column is more than 1,024 bytes, then there is one row for each 1,024-byte portion of the candidate views. The table contains the following columns:

Column Name	Column Type	Column Description
DETAIL_TABLE_NAME	CHAR (128)	Name of the base (detail) table. This column must be constrained with a single detail table per query.
START_DATE	TIMESTAMP	Start date for aggregate query analysis. Scope of analysis is defined by an equality constraint on the specified date range.
END_DATE	TIMESTAMP	End date for aggregate query analysis. Scope of analysis is defined by an equality constraint on the specified date range.
AGGR_ELAPSED_TIME	INTEGER	Time, in seconds, spent in executing aggregate parts of the query sub-plans for a group of queries that could be represented by a candidate view.
REFERENCE_COUNT	INTEGER	Number of times a candidate view would have been used to answer queries referencing the specified base table.

Column Name	Column Type	Column Description
SAMPLE_VIEW_NAME	CHAR(128)	Name of an existing view defined on the specified detail table that contains a subset of the rows in the detail table. Use this to limit the scope of analysis to a portion of the detail table. This speeds up the processing time of the Advisor analysis.
SIZE	INTEGER	Size (number of rows) of the precomputed view. If SAMPLE_VIEW_NAME column is constrained, the value is the size of the sample view.
REDUCTION_FACTOR	DOUBLE (FLOAT)	(Detail table size / View size). Size is defined as number of rows. This indicator can be used to predict the reduction in average number of rows processed for a query. If SAMPLE_VIEW_NAME column is constrained, the value is (Sample view size / View size).
BENEFIT	DOUBLE (FLOAT)	Benefit of a view with respect to the set of views being analyzed. That is, the benefit of a view is computed by considering how it can improve the cost of evaluating views, including itself.
NAME	CHAR(128)	Name of an existing precomputed view defined on the specified base table. NULL for candidate views.
SEQ	INTEGER	Sequence number of the view text for SQL text greater than 1,024 bytes.
TEXT	CHAR(1024)	SQL text representing the candidate view's definition.

RBW_PRECOMPVIEW_UTILIZATION Table

The RBW_PRECOMPVIEW_UTILIZATION table contains information necessary to analyze the value of precomputed views that were created for a specific detail table. It also provides insight on a specific view's utilization and the costs and benefits of that view with respect to other views answering the same query.

This table has one row for every valid precomputed view defined in the database. This includes views that are set to a valid state with the SET USE INVALID PRECOMPUTED VIEWS ON statement. The table contains the following columns:

Column Name	Column Type	Column Description
DETAIL_TABLE_NAME	CHAR(128)	Name of the base (detail) table. This column must be constrained with a single detail table per query.
START_DATE	TIMESTAMP	Start date for aggregate query analysis. Scope of analysis is defined by an equality constraint on the specified date range.
END_DATE	TIMESTAMP	End date for aggregate query analysis. Scope of analysis is defined by an equality constraint on the specified date range.
NAME	CHAR(128)	Name of precomputed view defined on the specified base table.
SIZE	INTEGER	Size of the precomputed view (number of rows).
REDUCTION_FACTOR	DOUBLE (FLOAT)	(Detail table size / View size). Size is defined as number of rows. This indicator can be used to predict the reduction in average number of rows processed for a query.
BENEFIT	DOUBLE (FLOAT)	Benefit of a view with respect to the set of views being analyzed.

Column Name	Column Type	Column Description
REFERENCE_COUNT	INTEGER	Number of times a view was used to answer queries referencing the specified base table.
NON_EXACT_MATCH_COUNT	INTEGER	Number of times a view was used to retrieve information that is not an exact match of what is stored in the precomputed view (for example, a query that performs another aggregation on the data in the precomputed view).

Checklist of Advisor Tasks

To use the Advisor:

Action	Page
1. Enable the Red Brick Vista option with a license key.	Note: For information about installation, refer to the <i>Installation and Configuration Guide</i> .
2. Create the Advisor log file (ADMIN ADVISOR_LOGGING ON or ALTER SYSTEM START ADVISOR_LOGGING).	5-3
3. Enable Advisor query logging (OPTION ADVISOR_LOGGING ON or SET ADVISOR_LOGGING ON).	5-3
4. Provide authority to access the advisor for the user who will query the Advisor (ACCESS_ADVISOR_INFO task authorization).	5-7
5. Define any explicit hierarchies with CREATE HIERARCHY statements that are valid for your schema.	3-11
6. Log user queries for a significant period of time. The longer and more representative a sample of queries, the better the advice.	
7. Query the RBW_PRECOMPVIEW_UTILIZATION table to analyze the usefulness of existing precomputed views.	5-10
8. Query the RBW_PRECOMPVIEW_CANDIDATES table to analyze precomputed views that, if created, would improve query performance.	5-12
9. Analyze the results of your Advisor system table queries.	5-17
10. Create new precomputed views or remove existing precomputed views, based on your analysis.	

A

Glossary

Advisor

The logging and analysis component of the Red Brick Vista option. The Advisor measures the benefits of existing precomputed views and suggests new precomputed views to create based on query history.

aggregate elapsed time

The total amount of time spent processing the aggregation portion of a query.

aggregate navigator

A layer of software that rewrites SQL statements to access aggregate tables instead of detail-level tables. *See* query rewrite system.

aggregate table

In general, a table that summarizes or consolidates detail-level records from other database tables. In the context of the Red Brick Vista option, an aggregate table is a precomputed table that stores the results of an aggregate query defined in an associated precomputed view.

aggregate view

See precomputed view.

aggregate-aware SQL

SQL that has been rewritten to use aggregate tables, thereby accelerating query performance.

aggregation query

A query that requires the summarization or consolidation of rows in database tables, typically using a set function, such as SUM or COUNT, and a GROUP BY clause.

base table

See detail table.

benefit

A column in the Advisor system tables used to measure the relative benefit of a precomputed view compared to other precomputed views in the database.

candidate view

A precomputed view suggested by the Advisor.

consolidation

Another term for summarization or aggregation. The data in aggregate tables “consolidates” detail-level data.

detail table

A base table that contains the detail-level data that is loaded into the data warehouse from an operational system. For example, the detail records in a data warehouse used to analyze retail sales might derive from a point-of-sales (POS) system.

derived dimension

An aggregate dimension table derived from a detail dimension table; also known as a “shrunk dimension.”

exact match rewrite

A query that can be answered by a precomputed view without performing additional aggregation.

explicit rollup

See rollup.

functional dependency

A many-to-one relationship shared by columns of values in database tables. A functional dependency from column *X* to column *Y* is a constraint that requires two rows to have the same value for the *Y* column if they have the same value for the *X* column. *See also* hierarchy.

generated SQL

SQL as rewritten internally by the Red Brick Vista option for faster query processing.

grain, granularity

The level of detail of the rows in base tables. The grain of an aggregate table is coarser than the grain of the detail table from which it is derived.

hierarchy

A functional dependency declared by the warehouse administrator, using the CREATE HIERARCHY command.

implicit rollup

See rollup.

many-to-one relationship

See functional dependency.

materialized view

See precomputed view.

metadata

System table data that describes database objects and their relationships.

non-exact match rewrite, non-exact match count

A query that is rewritten even though it does not exactly match the data defined in the precomputed view. The query is rewritten by performing aggregation on the data in the precomputed view. The number of times such additional aggregations occur for a given precomputed view is the *non-exact match count*. *See also* exact match.

precomputed table

A table associated with a precomputed view in a CREATE VIEW...USING statement. In the context of the Red Brick Vista option, precomputed tables are always *aggregate tables*.

precomputed view

A view linked to a database table known as a *precomputed table*. The view defines a query, and the table contains its precomputed results. The query rewrite system analyzes existing precomputed views to find the optimal way to rewrite each query.

query rewrite system

An aggregate navigation system that intercepts users' queries and invisibly rewrites them to use aggregate tables associated with precomputed views, thereby accelerating performance.

rollup

The computation of aggregates that are coarser than existing precomputed aggregates—for example, the rollup of monthly sales totals to quarterly and annual sales totals. This rollup capability relies on functional dependencies in the data, as declared explicitly with CREATE HIERARCHY statements or known implicitly to the query rewrite system through primary key/foreign key relationships.

rewritten query

A query that is rewritten to use aggregate tables associated with precomputed views. Query performance is accelerated and the rewrites are transparent to users.

sample view

A view that defines a subset of the rows in a detail table, used to improve performance of queries of the RBW_PRECOMPVIEW_CANDIDATES table.

shrunk dimension

See derived dimension.

summary table

Another name for an aggregate table. The term *prestored summary* is also used to refer to an aggregate table.

uniform probability

Assumption that all precomputed views were referenced an equal number of times—an optional mechanism for speeding up queries of the Advisor system tables.



Index

A

- ACCESS_ADVISOR_INFO task
 - authorization 5-7
- ADMIN_ADVISOR_LOGGING parameter
 - creating Advisor log files 5-3
- Advisor
 - candidate views 3-11
 - checklist of tasks 5-28
 - introduction to 1-5
 - overview 5-2
- Advisor logging
 - candidate views 5-4
 - configuring 5-3
 - correlated subqueries 5-4
 - log files 5-3
 - queries that are not logged 5-5
- aggregate queries, defined 2-3
- aggregate tables
 - creating 3-2
 - defined 2-3
 - example 3-3
 - existing 3-4
 - family of 3-19
 - loading 3-2
 - loading with cascaded inserts 3-8
 - visibility to client tools 3-27
- aggregation columns 3-6
- aggregation functions 3-6
- Aroma database 1-6, 3-3
- averages, rewriting queries that
 - calculate 4-23
- AVG function 3-6

B

- BENEFIT column
 - described 5-17
 - how it is calculated 5-19
 - understanding 5-19 to 5-23
 - what numbers mean 5-20
- BREAK BY queries, not rewritten 3-9

C

- candidate views 3-11
- cascaded inserts 3-8
- cases, tracked by technical support xvi
- compound expressions 3-9
- constraint names, in hierarchy
 - definitions 3-13
- conventions
 - syntax diagrams xii
 - syntax notation xi
- correlated subqueries
 - logging 5-4
 - rewriting 4-20
- cost-based analysis, of precomputed
 - views 3-8
- COUNT function 3-6
- CREATE HIERARCHY command 2-7, 3-12
- CREATE TABLE statements
 - for aggregate tables 3-4
 - for derived dimensions 3-20
- CREATE VIEW...USING command 3-5
- Customer Support Center xv

Index

D

- data skew 5-5
- derived dimensions 2-9
 - as referenced tables 4-11
 - creating 3-17
 - examples of rewritten queries 4-11
 - precomputed views linked to 3-21
- dimension tables, derived 3-17
- DISTINCT function 3-6, 3-21
- documentation
 - list of Red Brick Systems ix
 - support xvii
- DROP HIERARCHY command 3-13

E

- e-mail addresses, for Red Brick Systems xv
- exact match, defined 5-11
- EXCEPT queries 3-10
- EXPLAIN command 3-24, 4-21
- explicit hierarchies 3-11
 - defined 2-7
 - example 2-8
 - examples of rewritten queries 4-8

F

- family of aggregate tables 3-19
- foreign-key constraint names, in hierarchy
 - definitions 3-13
- FROM clause subqueries 4-20
- functional dependencies 3-11, 4-8
 - defined 2-6
 - example 2-8
 - validity of 3-11

G

- generated SQL 3-18
 - simplified with derived dimensions 4-15
- GROUP BY clause, compound expressions in 3-9
- grouping columns 3-5

H

- HAVING clause 3-7

hierarchies

- concept of 2-6
- explicit 3-11, 4-8
- implicit 3-15, 4-16
- use with Advisor 5-8
- validity of 3-11
- verifying validity of 3-14

I

- implicit hierarchies 3-15
 - defined 2-9
 - examples of rewritten queries 4-16
- indexes
 - for rewritten queries 3-8
 - on aggregate tables 4-14
- INSERT command 3-4
- INSERT statements
 - for derived dimensions 3-20
 - rewriting 3-8
- INTERSECT queries 3-10
- invalid precomputed views 3-23
 - effect on Advisor 5-4

J

- join predicates, for precomputed views 3-7

K

- keywords
 - in syntax diagrams xiv

L

- logging, Advisor, *See* Advisor logging

M

- materialized views, *See* precomputed views 3-5
- MAX function 3-6
- metavariables
 - in syntax diagrams xiv
- MIN function 3-6

N

- NON_EXACT_MATCH_COUNT
 - column 5-11
- notation conventions xi

O

- ODBC client applications 3-27
- OPTION ADVISOR_LOGGING parameter
 - enabling Advisor logging 5-4
- ORDER BY clause, in rewritten queries 4-6
- outboard tables, rewriting queries
 - against 2-9, 4-9

P

- precomputed query expressions 3-5
- precomputed tables, *See* aggregate tables
- precomputed views
 - cost-based analysis 3-8
 - creating 3-5
 - defined 2-2
 - example 3-7
 - join predicates 3-7
 - linked to derived dimensions 3-21
 - SET commands 3-23
 - unknown columns 4-7
 - validity of 3-23
 - visibility to client tools 3-27
- primary key/foreign key relationships,
 - functional dependencies in 3-11

Q

- query blocks, rewritten separately 3-8
- query expressions 3-5
- query rewrite system 3-1 to 3-27
 - calculating averages 4-23
 - derived dimensions 4-11
 - detailed examples 4-1 to 4-22
 - exact-match rewrites 4-3
 - explicit hierarchies 4-8
 - implicit hierarchies 4-16
 - introduction to 1-3
 - key concepts 2-1 to 2-9
 - negative tests 4-7
 - optimizing rewritten queries 3-17
 - queries not rewritten 3-9
 - query blocks 3-8
 - rewriting subqueries 4-20
 - summary of use 3-28
 - turning on and off 3-24

R

- RBW_PRECOMPVIEW_CANDIDATES
 - table
 - described 5-24
 - rules for querying 5-12
- RBW_PRECOMPVIEW_UTILIZATION
 - table
 - described 5-26
 - querying 5-10
- RBW_TABLES system table 3-25, 3-27
- RBW_TABLES_VIEW, creating 3-27
- RBW_VIEWS system table 3-25
- Red Brick ODBC Driver 3-27
- REDUCTION_FACTOR column 5-17
- REFERENCE_COUNT column 5-18
- referenced tables, derived dimensions
 - as 4-11
- rewritten queries, *See* query rewrite system
- rewritten SQL 3-18
- RISQL display functions, in rewritten queries 4-7
- rollups
 - defined 2-6
 - hierarchies and 3-11

S

- SAMPLE_VIEW_NAME column 5-12
- SET commands, for precomputed views 3-23
- set functions 3-6
- SET STATS INFO command 3-24, 4-2
- shrunk dimensions, *See* derived dimensions
- SIZE column 5-17
- SQLTables function 3-27
- STARindexes, for aggregate tables 3-18, 4-14, 4-18
- STARjoin query, rewritten 4-3
- statistics messages, for queries 4-2
- subqueries, rewriting 4-20
- SUM function 3-6
- support
 - documentation xvii
 - technical xv

Index

- syntax diagrams
 - conventions for xii
 - keywords in xiv
 - metavariables in xiv
- syntax notation xi
- system tables, querying view-related 3-25

- T**
- technical support xv
- temporary tables, use with Advisor 5-8
- troubleshooting
 - general problems xvii

- U**
- UNIFORM PROBABILITY FOR ADVISOR
 - command 5-21
- UNION queries
 - separate query blocks for 3-8
 - when rewritten 3-10

- USING clause, in CREATE VIEW
 - command 3-5

- V**
- validity
 - of hierarchies 3-11
 - of precomputed views 3-23
 - view of RBW_TABLES 3-27
 - views, *See* precomputed views

- W**
- WHEN clause
 - disallowed in view definitions 3-7
 - in rewritten queries 4-7
- WHERE clause predicates, in rewritten queries 4-6
- World Wide Web address, for Red Brick Systems xv



RED BRICK®

**USA SALES
OFFICES**

1040 Crowne Point Parkway, Suite 250, Atlanta, GA 30338 +1 770 804 2440
2215 York Road, Suite 409, Oak Brook, IL 60521 +1 630 472 8500
1120 Avenue of the Americas, 4th Floor, New York, NY 10036 +1 212 626 6815
5314 Arapaho Road, Dallas, TX 75248 +1 972 702 1750
2010 Corporate Ridge, 7th Floor, McLean, VA 22102 +1 703 883 9310

**UK SALES
OFFICE**

Red Brick Systems UK Ltd., 45 Berkeley Square, Mayfair, London W1A 1EB
United Kingdom +44 171 290 8373

**AUSTRALASIA
HEADQUARTERS**

Red Brick Systems Australasia Pty. Ltd., Level 20, 99 Walker Street,
North Sydney, NSW 2060 Australia +61 02 9911 7744

**JAPAN
HEADQUARTERS**

Red Brick Japan Co. Ltd., Level 16 Shiroyama Hills, 4-3-1 Toranomom,
Minato-ku, Tokyo 105 Japan +81 3 5403 4638