

Informix Vista

User's Guide

Version 6.0
November 1999
Part No. 000-6371

Published by Informix® Press

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

© 1999 Informix Corporation. All rights reserved. The following are trademarks of Informix Corporation or its affiliates, one or more of which may be registered in the United States or other jurisdictions:

Answers OnLine™; C-ISAM®; Client SDK™; DataBlade®; Data Director™; Decision Frontier™; Dynamic Scalable Architecture™; Dynamic Server™; Dynamic Server™, Developer Edition™; Dynamic Server™ with Advanced Decision Support Option™; Dynamic Server™ with Extended Parallel Option™; Dynamic Server™ with MetaCube®; Dynamic Server™ with Universal Data Option™; Dynamic Server™ with Web Integration Option™; Dynamic Server™, Workgroup Edition™; Dynamic Virtual Machine™; Enterprise Decision Server™; Formation™; Formation Architect™; Formation Flow Engine™; Gold Mine Data Access®; IIF.2000™; i.Reach™; i.Sell™; Illustra®; Informix®; Informix® 4GL; Informix® InquireSM; Informix® Internet Foundation.2000™; InformixLink®; Informix® Red Brick® Decision Server™; Informix Session Proxy™; Informix® Vista™; InfoShelf™; Interforum™; I-Spy™; Mediazation™; MetaCube®; NewEra™; ON-Bar™; OnLine Dynamic Server™; OnLine/Secure Dynamic Server™; OpenCase®; Orca™; PaVER™; Red Brick® and Design; Red Brick® Data Mine™; Red Brick® Mine Builder™; Red Brick® Decisionscape™; Red Brick® Ready™; Red Brick Systems®; Regency Support®; Rely on Red BrickSM; RISQL®; Solution DesignSM; STARindex™; STARjoin™; SuperView®, TARGETindex™; TARGETjoin™; The Data Warehouse Company®; The one with the smartest data wins.™; The world is being digitized. We're indexing it.SM; Universal Data Warehouse Blueprint™; Universal Database Components™; Universal Web Connect™; ViewPoint®; Visionary™; Web Integration Suite™. The Informix logo is registered with the United States Patent and Trademark Office. The DataBlade logo is registered with the United States Patent and Trademark Office.

Documentation Team: Twila Booth, Laura Kremers, Jerry Tattershall

GOVERNMENT LICENSE RIGHTS

Software and documentation acquired by or for the US Government are provided with rights as follows:

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 of Defense use, with rights as restricted by vendor's standard license, unless superseded by a negotiated vendor license, as prescribed in DFARS 227.7202.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

Table of Contents

Introduction

In This Introduction	3
About This Guide	3
Types of Users	3
Software Dependencies	4
New Features	5
Documentation Conventions	5
Syntax Notation	6
Syntax Diagrams	7
Keywords and Punctuation	9
Identifiers and Names	9
Icon Conventions	10
Customer Support	10
New Cases	11
Existing Cases	12
Troubleshooting Tips.	12
Related Documentation	13
Additional Documentation	15
Online Manuals	15
Printed Manuals	16
Informix Welcomes Your Comments	16

Chapter 1

Introduction to Informix Vista

In This Chapter	1-3
Aggregates in Decision Support Queries	1-4
The Vista Solution	1-5
The Query Rewrite System.	1-5
The Advisor	1-7
Summary	1-8

Chapter 2	Key Concepts of Query Rewriting	
	In This Chapter	2-3
	Precomputed Views	2-4
	Aggregate Tables	2-6
	Aggregate Query Rewrites	2-7
	Rollups and Hierarchies	2-10
	Functional Dependencies	2-10
	Derived Dimensions	2-13
 Chapter 3	 Using the Query Rewrite System	
	In This Chapter	3-3
	Creating Aggregate Tables	3-4
	Populating Aggregate Tables.	3-4
	Example of an Aggregate Table	3-5
	CREATE TABLE Statement	3-6
	INSERT INTO...SELECT Statement	3-6
	Creating Precomputed Views	3-7
	CREATE VIEW...USING Command	3-7
	Example View Definition	3-10
	Cost-Based Analysis of Precomputed Views	3-11
	Using Hierarchies	3-13
	Explicit Hierarchies	3-14
	Implicit Hierarchies	3-18
	Optimizing Query Rewrites	3-19
	Creating Derived Dimensions	3-20
	Creating Indexes	3-25
	Setting Up the Query-Rewriting Environment	3-26
	Marking Precomputed Views Valid	3-26
	Granting Select Privileges on Precomputed Views	3-27
	Turning On the Query Rewrite System	3-28
	Generating Statistics.	3-28
	Querying the RBW_VIEWS System Table	3-28
	Making Precomputed Views Invisible to Client Tools	3-30
	Checklist of Query-Rewriting Tasks	3-31

Chapter 4

Query Rewrite Case Studies

In This Chapter	4-3
General Instructions	4-4
Troubleshooting	4-4
Case 1. Rewriting a Basic Query	4-5
The Query	4-5
Case 2. Making Use of Explicit Hierarchies	4-10
The Query	4-11
Case 3. Defining Hierarchies on Date	4-13
The Queries	4-14
Extend the Hierarchy to Month	4-17
Fiscal Periods	4-18
Case 4. Optimizing with Derived Dimensions	4-22
The Queries	4-22
Case 5. Using Implicit Hierarchies	4-27
The Queries	4-27
Case 6. Rewriting Subqueries	4-30
The Query and Result Set	4-30
Case 7. Rewriting a Query That Calculates Averages	4-31
The Query and Result Set	4-32
Case 8. Using the EXPORT Command	4-33
The First Method	4-33
An Alternative Method	4-35

Chapter 5

Using the Advisor

In This Chapter	5-3
Advisor Overview	5-4
Analysis of Query Patterns	5-4
Advisor System Tables	5-4
Advisor Log Files	5-5
Configuring the Advisor Logging System	5-5
Creating the Advisor Log Files	5-5
Logging Queries	5-6
Setting the ACCESS_ADVISOR_INFO Task Authorization	5-9
Defining Valid Hierarchies	5-10
Querying the Advisor	5-10
Inserting the Results of an Advisor Query into a Table	5-10
Querying the RBW_PRECOMPVIEW_UTILIZATION Table	5-13
Querying the RBW_PRECOMPVIEW_CANDIDATES Table	5-15

Interpreting the Results of Advisor Queries	5-21
BENEFIT Column	5-21
SIZE and REDUCTION_FACTOR Columns	5-21
REFERENCE_COUNT Column.	5-22
Combining the Results	5-23
Understanding the BENEFIT Column	5-23
How the BENEFIT Column Is Calculated	5-24
What the Numbers Mean	5-25
Uniform Probability.	5-26
Advisor System Table Column Descriptions	5-29
RBW_PRECOMPVIEW_CANDIDATES Table.	5-29
RBW_PRECOMPVIEW_UTILIZATION Table.	5-30
Checklist of Advisor Tasks	5-32

Chapter 6

Managing Vista with the Administrator

In This Chapter	6-3
Getting Started	6-4
Managing Vista	6-6
Creating an Aggregate Table and Its View	6-9
Creating and Verifying Hierarchies	6-22
The Store to District Hierarchy	6-23
Using the Advisor	6-26
Precomputed View Utilization	6-26
Candidate View Generation	6-31

Glossary

Index

Introduction

In This Introduction	3
About This Guide	3
Types of Users	3
Software Dependencies	4
New Features.	5
Documentation Conventions	5
Syntax Notation	6
Syntax Diagrams	7
Keywords and Punctuation	9
Identifiers and Names	9
Icon Conventions	10
Customer Support	10
New Cases	11
Existing Cases	12
Troubleshooting Tips	12
Related Documentation	13
Additional Documentation	15
Online Manuals	15
Printed Manuals	16
Informix Welcomes Your Comments.	16

In This Introduction

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

About This Guide

This guide describes the Informix Vista aggregate navigation and advisory system. Illustrates how Vista improves the performance of queries by automatically rewriting queries using aggregates, describes how the Advisor recommends the best set of aggregates based on data collected daily, and shows how the system operates in a versioned environment.

Types of Users

This guide is written for the following users:

- Database administrators
- Database-application programmers
- Database architects
- Database designers
- Database developers
- Performance engineers

This guide assumes that you have the following background:

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

Software Dependencies

This guide assumes that you are using Informix Red Brick Decision Server, Version 6.0, as your database server.

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

For information about how to create and populate the demonstration database, see the [Administrator's Guide](#). For a description of the database and its contents, see the [SQL Self-Study Guide](#).

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

New Features

The following section describes new database server features relevant to this document. For a comprehensive list of new features, see the release notes.

- Informix Red Brick JDBC Driver, which allows Java programs to access database management systems
- Support for the VARCHAR (variable-length character) data type
- Performance improvement to DELETE and UPDATE operations
- Enhancements to BREAK BY and RESET BY functionality
- Ability to freeze a versioned database at one revision for user queries but allow update activities to continue generating new revisions
- Versioned invalidation of views in Vista

Documentation Conventions

Informix Red Brick documentation uses the following notation and syntax conventions:

- 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 *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).







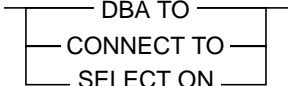
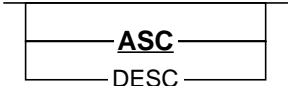
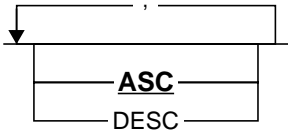
Syntax Notation

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

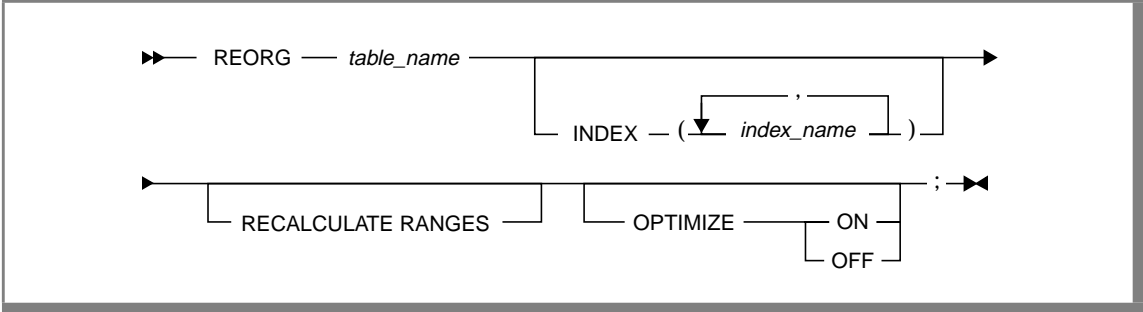
Command Element	Example	Convention
Values and parameters	<i>table_name</i>	Items that you replace with an appropriate name, value, or expression are in <i>italic</i> type style.
Optional items	[]	Optional items are enclosed by square brackets. Do not type the brackets.
Choices	ONE TWO	Choices are separated 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

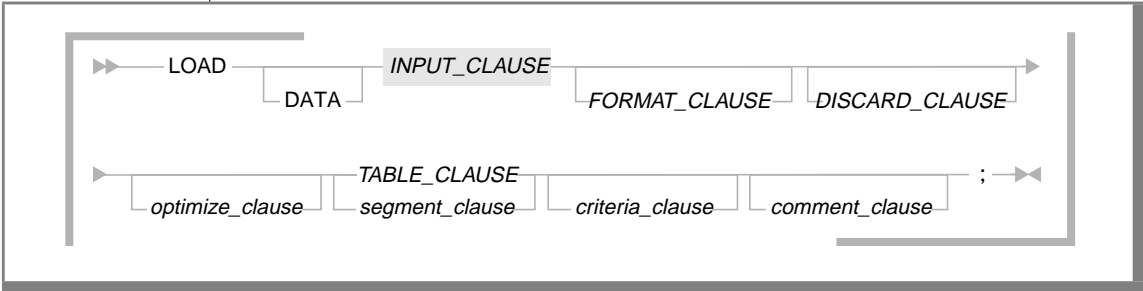
This guide uses diagrams built with the following components to describe the syntax for statements and all commands other than system-level commands.

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.

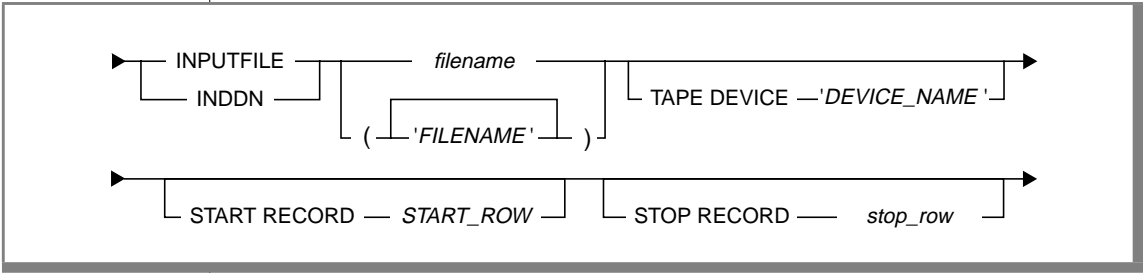
The preceding syntax elements are combined to form a diagram as follows.



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



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



Keywords and Punctuation

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

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

Identifiers and Names

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




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

```
►► — SELECT — column_name — FROM — table_name —◄◄
```

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

Icon Conventions

Throughout the documentation, you will find text that is identified by comment icons. Comment icons identify three types of information, as the following table describes. This information always appears in *italics*.

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

Customer Support

Please review the following information before contacting Informix Customer Support.

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

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

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

Telephone 1-800-274-8184
 (7 A.M. to 7 P.M. CST, Monday through Friday)

Internet access <http://www.informix.com/services>

New Cases

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

- Red Brick Decision Server version
- Platform and operating-system version
- Error messages returned by Red Brick Decision Server or the operating system
- Concise description of the problem, including any commands or operations performed before you received the error message
- List of Red Brick Decision Server or operating-system configuration changes made before you received the error message

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

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

Existing Cases

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

Troubleshooting Tips

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

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

Related Documentation

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

Document	Description
<i>Administrator's Guide</i>	Describes warehouse architecture, supported schemas, and other concepts relevant to databases. Procedural information for designing and implementing a database, maintaining a database, and tuning a database for performance. Includes a description of the system tables and the configuration file.
<i>Installation and Configuration Guide</i>	Provides installation and configuration information, as well as platform-specific material, about Red Brick Decision Server and related products. Customized for either UNIX or Windows NT.
<i>Messages and Codes Reference Guide</i>	Contains a complete listing of all informational, warning, and error messages generated by Informix Red Brick Decision Server products, including probable causes and recommended responses. Also includes event log messages that are written to the log files.
<i>The release notes</i>	Contains information pertinent to the current release that was unavailable when the documents were printed.
<i>RISQL Entry Tool and RISQL Reporter User's Guide</i>	Is a complete guide to the RISQL Entry Tool, a command-line tool used to enter SQL statements, and the RISQL Reporter, an enhanced version of the RISQL Entry Tool with report-formatting capabilities.

(1 of 2)

Document	Description
SQL Reference Guide	Is a complete language reference for the Informix Red Brick SQL implementation and RISQL extensions for warehouse databases.
SQL Self-Study Guide	Provides an example-based review of SQL and introduction to the RISQL extensions, the macro facility, and Aroma, the sample database.
Table Management Utility Reference Guide	Describes the Table Management Utility, including all activities related to loading and maintaining data. Also includes information about data replication and the <i>rb_cm</i> copy management utility.

(2 of 2)

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

Document	Description
Client Connector Pack Installation Guide	Includes procedures for installing and configuring the Informix Red Brick ODBC Driver, the RISQL Entry Tool, and the RISQL Reporter on client systems. Included for sites that purchase the Client Connector Pack.
SQL-BackTrack User's Guide	Is a complete guide to SQL-BackTrack, 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.

(1 of 2)

Document	Description
<i>Informix Vista User's Guide</i>	Describes the Informix Vista aggregate navigation and advisory system. Illustrates how Vista improves the performance of queries by automatically rewriting queries using aggregates, describes how the Advisor recommends the best set of aggregates based on data collected daily, and shows how the system operates in a versioned environment.
<i>JDBC Connectivity Guide</i>	Includes information about Informix Red Brick JDBC Driver and the JDBC API, which allow Java programs to access database management systems.
<i>ODBC Connectivity Guide</i>	Includes information about ODBC conformance levels and instructions for using the Informix Red Brick ODBClib SDK to compile and link an ODBC application.

(2 of 2)

Additional references you might find helpful include:

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

Additional Documentation

For additional information, you might want to refer to the following documents, which are available as online and printed manuals.

Online Manuals

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

Printed Manuals

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

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

Informix Welcomes Your Comments

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

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

Write to us at the following address:

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

If you prefer to send electronic mail, our address is:

doc@informix.com

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

We appreciate your suggestions.

Introduction to Informix Vista

In This Chapter	1-3
Aggregates in Decision Support Queries	1-4
The Vista Solution	1-5
The Query Rewrite System	1-5
How Aggregate Queries Are Rewritten	1-6
The Advisor	1-7
Summary	1-8

In This Chapter

This chapter introduces Vista, a component of the Informix Red Brick Decision Server that optimizes queries using precomputed aggregates. This component also captures and stores statistics on how often the engine rewrites queries using aggregates. This information simplifies the management of aggregate tables for the database administrator (DBA).

This chapter contains the following sections.

- [Aggregates in Decision Support Queries](#)
- [The Vista Solution](#)
 - [The Query Rewrite System](#)
 - [The Advisor](#)
- [Summary](#)

Aggregates in Decision Support Queries

Decision-support queries frequently calculate aggregate totals such as monthly and quarterly revenues by product over last year. Such queries, called “aggregate queries,” can be expensive when they have to calculate aggregate totals from detail data. For example, when a query calculates quarterly sales averages from daily data, it must retrieve, group, and calculate aggregate totals from an enormous number of rows. Meanwhile, the user waits.

The quickest way to run this kind of query is not to run it at all. Instead, the DBA can precompute values for the aggregates (SUM, AVG, MIN, MAX, COUNT, and others), store these aggregate values in tables, and then notify the users and developers that the aggregate tables are available. Queries can then fetch aggregate values directly from a table rather than rolling them up from the details. The user queries will generally run much faster.

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.

The Vista Solution

The Vista solution provides a systematic approach to precomputing aggregate values 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 Vista solution consists of two components:

- A query rewrite system that intercepts and rewrites queries to use aggregate tables. This process is completely transparent to the query whether issued by a user or an application.
- 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.

Vista operates within the context of versioned or nonversioned databases.

The Query Rewrite System

The Vista solution is fully integrated with the Red Brick Decision Server. Queries are intercepted and rewritten by the database 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 relational database management system (RDBMS) and the query tool. The integrated approach has two distinct advantages:

- 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 an aggregate table data is updated, Vista invalidates precomputed views based on this aggregate table and does not employ them to rewrite queries.
- 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 Decision Engine. Everything that the administrator needs to do to make query rewriting possible can be done using 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 know. When a user submits a query, the query rewrite system evaluates the precomputed views the database 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 Vista solution. 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.

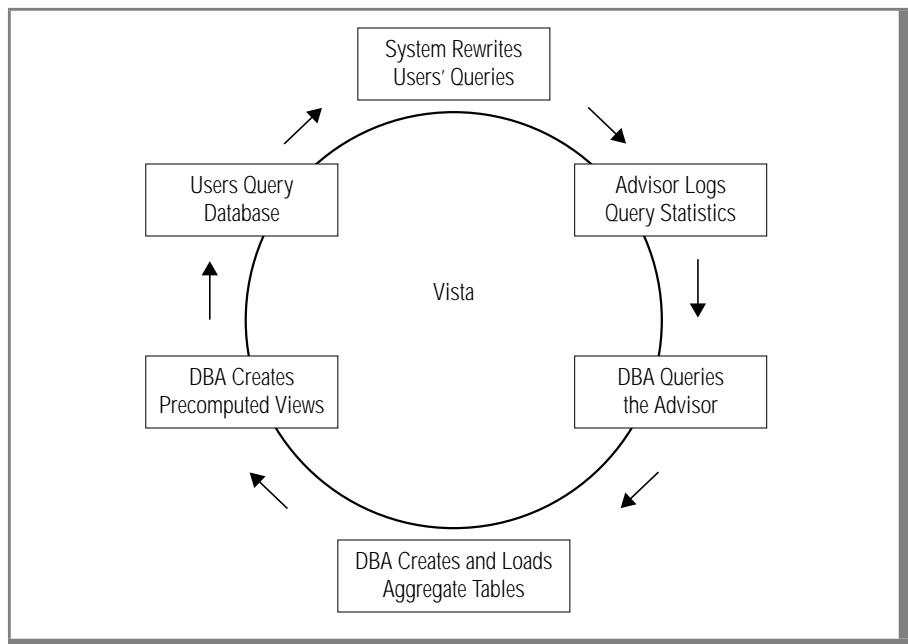
Queries against the Advisor system tables perform detailed analysis of the information stored in the logs. The analysis is based on sophisticated algorithms that determine the best set of aggregate tables for the actual query history. As part of the analysis, a benefit is assigned to each existing view 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 on 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 Decision Server techniques for accelerating query performance, such as STARjoin and TARGETjoin processing, the Vista offers the database administrator a flexible set of features for optimizing aggregation queries.

The process of getting advice, creating precomputed views, and using the query rewrite system is iterative, as the following diagram shows.



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 database server software and is described in detail in the [SQL Self-Study Guide](#).

Key Concepts of Query Rewriting

In This Chapter	2-3
Precomputed Views	2-4
View Validity	2-5
Versioned Databases	2-5
Aggregate Tables	2-6
Aggregate Query Rewrites	2-7
Rollups and Hierarchies	2-10
Functional Dependencies	2-10
Dependencies Declared by the Administrator	2-11
Dependencies Known Implicitly to the Query Rewrite System	2-13
Derived Dimensions	2-13

In This Chapter

The query rewrite system, a major component of Vista, inspects each query during compilation, compares the tables it references with a list of aggregate tables, and then, whenever possible, rewrites the query to retrieve rows from an aggregate rather than a detail table. The rewritten query generally retrieves far fewer rows than the original query and thereby runs much faster.

To take full advantage of the query rewrite system, you need to understand how it uses precomputed or materialized views to rewrite queries. Additionally, you need to understand how detail data can be systematically rolled up into summarized data.

This chapter describes how the query rewrite system can improve query performance and illustrates the conceptual details.

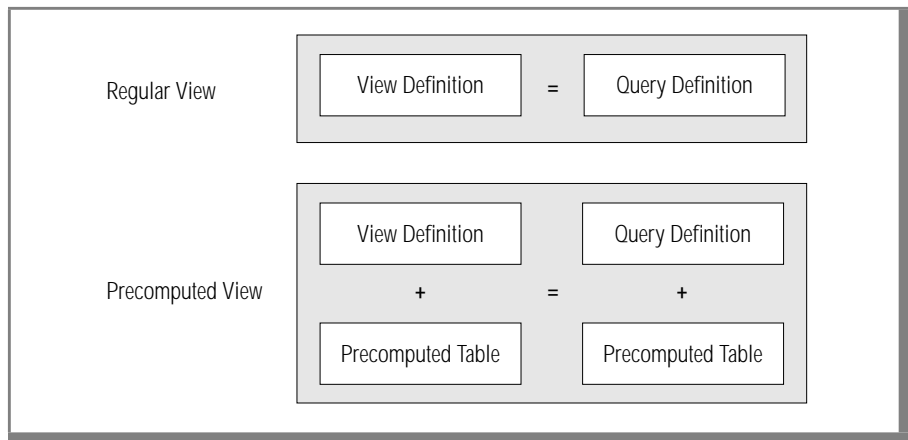
This chapter contains the following sections.

- [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, the result set for 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 Vista.

For a view to be precomputed, both of the following statements must be true:

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

Furthermore, Vista does not *validate* the data used to populate a precomputed table. That is the sole responsibility of the database administrator.

View Validity

A precomputed view can be *valid* or *invalid*. Vista uses only valid views to rewrite a query: the query rewrite system excludes invalid views from its list of candidates.

Initially, Vista marks a precomputed view as invalid. Before Vista can use this view to rewrite a query, the database administrator must mark it as valid using a SET PRECOMPUTED VIEW command. Only the database administrator can mark a view as valid; Vista can only invalidate precomputed views, not validate them.

Vista invalidates a precomputed view when:

- The view is initially created
- Tables referenced by the view are modified by:
 - Server commands that insert, update, or delete rows
 - Table Management Utility load commands
 - Table Management Utility reorganize commands

For example, when the contents of a detail table change, Vista automatically invalidates all the precomputed views based on that table. Or, if a row is modified in a dimension referenced by the precomputed view, Vista invalidates the precomputed view. The view remains invalid until the administrator marks it valid using a SET command. Typically, the administrator marks precomputed views valid after verifying their correctness.



Warning: The database administrator must ensure that the precomputed table contains the exact result set that would be returned by the query defined in the view. Otherwise, rewritten queries can return incorrect result sets.

Versioned Databases

Vista manages precomputed views and their associated aggregate tables in the same way for versioned and nonversioned databases. The rules used to mark a versioned precomputed view invalid are the same as those used to mark a nonversioned one invalid.

A versioned database provides a way for user queries to take advantage of precomputed aggregates even while the administrator modifies detail and dimension tables referenced by the precomputed views. This mode of operation is possible when the DBA freezes the database at a given revision.

For example, the DBA can freeze the database early in the morning before the users arrive. Afterwards, users query the frozen version of the database while others modify the database tables and views. Late in the evening, after the users have departed, the DBA can unfreeze the database. At this time, the updated revisions are merged into the database and precomputed views are marked invalid. The DBA can refresh the aggregate tables at this time, mark them as valid, and the next day users work with fresh data.

For additional information on versioned databases, refer to the [Administrator's Guide](#). For information on controlling how the Table Management Utility invalidates precomputed views, refer to the [Table Management Utility Reference Guide](#).

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 aggregate function such as SUM or COUNT to aggregate detail data. For example, 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 within Vista a precomputed view definition always contains an aggregate query.

Aggregate Query Rewrites

Precomputing query results is a powerful means of speeding up query performance. Vista realizes this performance gain by intercepting and rewriting 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.

Vista 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 [“Case 1. Rewriting a Basic Query” on page 4-5](#).
 - Queries that apply additional calculations to the view data (by using a RSQL display function, for example) can be rewritten. See [page 4-5](#).
 - 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 2-10](#), and examples are presented on [page 4-10](#).
- By logging rewritten query activity and offering advice on the optimal set of precomputed views to create. For details, refer to [Chapter 5, “Using the Advisor.”](#)

In these ways, and in combination with other techniques for accelerating query performance (such as STARjoin and TARGETjoin processing), Vista improves query performance while minimizing both space requirements and the administrator’s maintenance workload.

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 Vista, the database administrator creates and populates a 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;
```



Important: This example uses an `INSERT INTO` command to load the aggregate table. Aggregate tables can be quite large, and in practice, most sites would load aggregate tables using the Table Management Utility for reasons of efficiency.

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);
```

Vista creates this precomputed view and marks it as invalid. Before the view can be used for query rewrites, the database administrator must mark it valid using a SET PRECOMPUTED VIEW command.

Once the Product_Sales_View has been marked valid, 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.

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

```
select * from product_sales;
```



Important: 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, Informix 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 Vista is its 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 Vista component 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 as well, 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 the 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 in 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*). 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 database 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 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. The first column named in a CREATE HIERARCHY statement, the *from* column, must be a NOT NULL column; that is, the column must have been declared NOT NULL when the table was created. (In the context of Vista, the terms *functional dependency* and *hierarchy* are synonymous.)



Warning: Functional dependencies declared with `CREATE HIERARCHY` statements are not validated by the database 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 3-17](#). 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 4-27](#).

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 3-19](#) and the case study that begins on [page 4-22](#).

Using the Query Rewrite System

In This Chapter	3-3
Creating Aggregate Tables	3-4
Populating Aggregate Tables	3-4
Example of an Aggregate Table	3-5
CREATE TABLE Statement.	3-6
INSERT INTO...SELECT Statement	3-6
Creating Precomputed Views	3-7
CREATE VIEW...USING Command.	3-7
Aggregation Columns	3-8
Example View Definition	3-10
Cost-Based Analysis of Precomputed Views	3-11
Indexes Not Considered by the Cost Model.	3-11
Rewritten INSERT INTO...SELECT Statements	3-12
Queries That Cannot Be Rewritten.	3-12
Using Hierarchies	3-13
Explicit Hierarchies	3-14
CREATE HIERARCHY Command.	3-14
Verifying the Validity of Hierarchies	3-17
Implicit Hierarchies	3-18
Optimizing Query Rewrites.	3-19
Creating Derived Dimensions.	3-20
Simplified SQL Generation	3-20
CREATE TABLE Statements	3-23
INSERT Statements	3-23
Precomputed View Definition	3-24
Creating Indexes	3-25

Setting Up the Query-Rewriting Environment	3-26
Marking Precomputed Views Valid	3-26
Using Invalid Views	3-26
Granting Select Privileges on Precomputed Views	3-27
Turning On the Query Rewrite System	3-28
Generating Statistics	3-28
Querying the RBW_VIEWS System Table	3-28
Making Precomputed Views Invisible to Client Tools	3-30
Checklist of Query-Rewriting Tasks	3-31

In This Chapter

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 contains the following sections.

- [Creating Aggregate Tables](#)
- [Creating Precomputed Views](#)
- [Using Hierarchies](#)
- [Optimizing Query Rewrites](#)
- [Setting Up the Query-Rewriting Environment](#)
- [Querying the RBW_VIEWS System Table](#)
- [Making Precomputed Views Invisible to Client Tools](#)
- [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 other tables. For example, in a retail database, the transaction-level data drawn from the online transaction processing (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 various time periods to produce a set of aggregate tables (Sales_Daily, Sales_Monthly, and so on).

In the context of Vista, 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 3-20](#).

Populating Aggregate Tables

Aggregate tables can be loaded with data using two methods:

- The Table Management Utility (TMU) with a LOAD command.
- The INSERT INTO...SELECT command.

Typically, the TMU will be the faster, more efficient method for two reasons. The TMU loads data into tables in a bulk mode; and, selected load commands exploit the auto-aggregate mode.

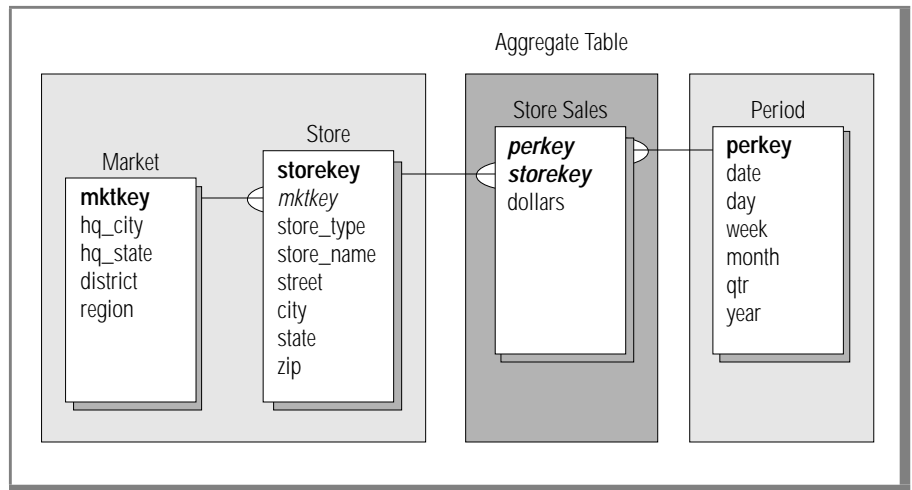
For information about the TMU and auto-aggregation, refer to the [Table Management Utility Reference Guide](#). For the syntax of the INSERT command, refer to the [SQL Reference Guide](#).

Example of an Aggregate Table

The following example is based on the sample Aroma database, which the [SQL Self-Study Guide](#) describes in detail.

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 17, 1999, the San Jose Roasting Company sold \$87.00 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 17, 1999, total sales at the San Jose Roasting Company were \$429.75.

CREATE TABLE Statement

The CREATE TABLE statement for Store_Sales looks 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 Vista; 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 3-7](#)).

INSERT INTO...SELECT Statement

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

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

This is a 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. If so, the administrator can mark the precomputed view valid, and the query rewrite system can begin using it. (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);
```

Before Vista can rewrite queries using this precomputed view, the administrator must mark the view valid using the SET PRECOMPUTED VIEW command.

Aggregation Columns

Aggregation columns in precomputed view definitions must be of the form

```
set_function(expression)
```

where *set_function* is one of the following:

- 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, constants, or both.

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 7. Rewriting a Query That Calculates Averages](#)” on page 4-31.
- 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

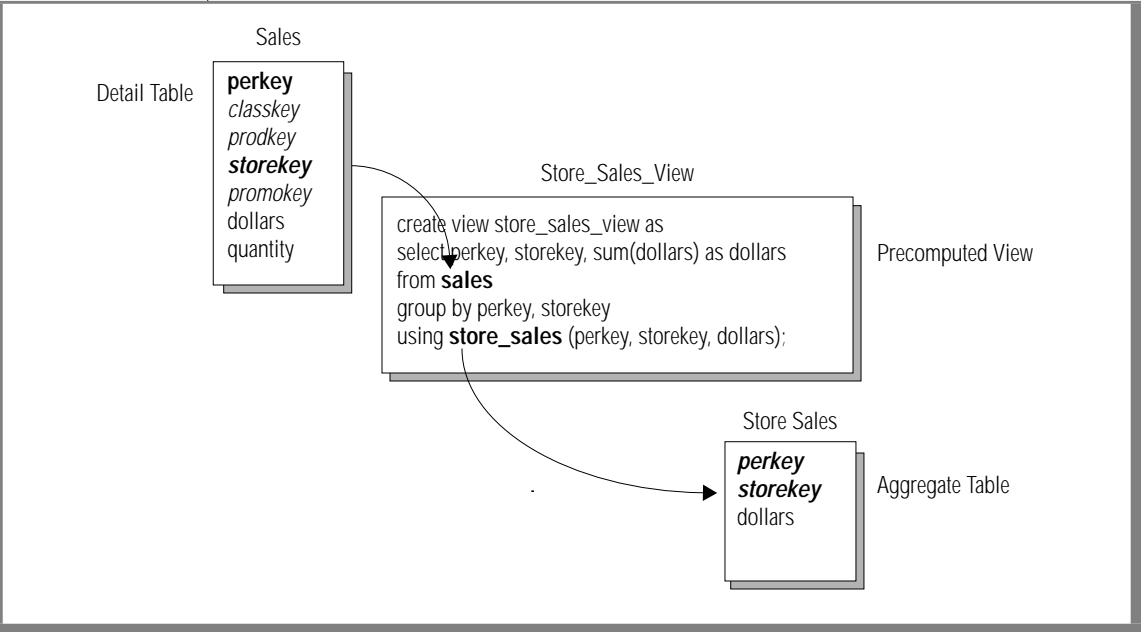
The query expression defined for a precomputed view is also restricted in the following ways:

- 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 5. Using Implicit Hierarchies” on page 4-27](#).

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 3-26.

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 Vista. For example, query rewrites that involve aggregate table families in a star schema must make use of STARindexes and TARGETindexes equivalent to those defined on the detail tables they replace. For examples of rewritten queries optimized by the use of indexes, refer to “[Case 4. Optimizing with Derived Dimensions](#)” on page 4-22.

For information about standard approaches to indexing database tables, refer to the [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 3-26](#)). 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 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](#).

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. This translates to a reduced set of aggregate tables, less maintenance for the administrator, and less storage space.

The key to this query-rewriting flexibility is the exploitation of functional dependencies inherent in the data (see [page 2-10](#)). These dependencies follow the path of primary key/foreign key relationships that are an integral part of the database 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 Vista, in much the same way that defining primary key/foreign key relationships is a routine prerequisite for creating database tables.

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.



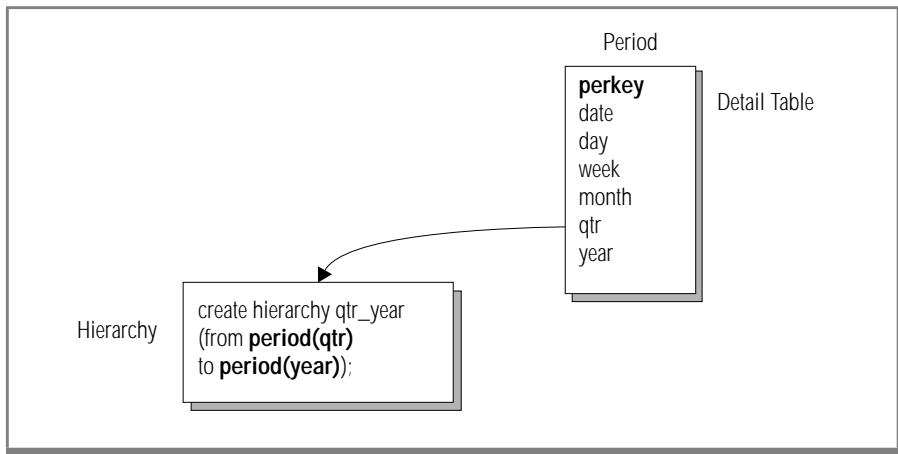
Warning: *Hierarchies must be defined with great care. The declaration of hierarchies on columns whose values do not satisfy a many-to-one relationship might cause rewritten queries to return incorrect results, without warning. The database server does not validate hierarchies when they are declared; 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 hierarchies if they become invalid.*

For examples of valid and invalid functional dependencies, refer to [“Dependencies Declared by the Administrator”](#) on page 2-11.

CREATE HIERARCHY Command

This command assigns a unique name to the hierarchy, 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. The column in the FROM clause of a hierarchy statement must be a column that has been declared NOT NULL.

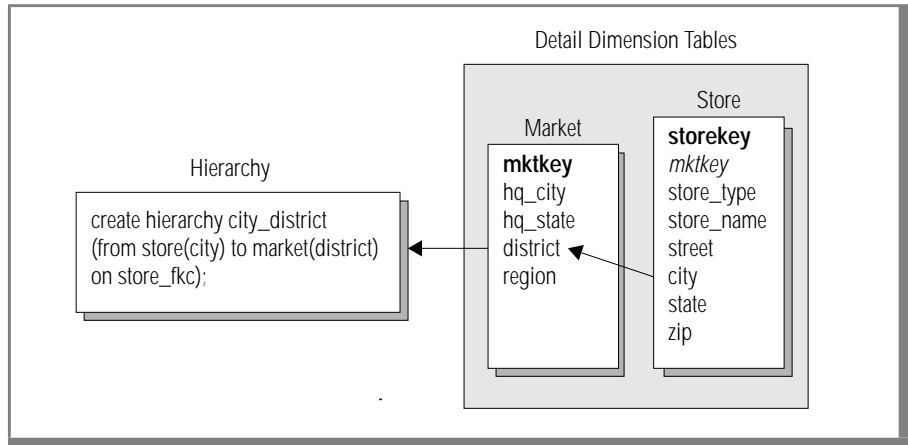
For example, the following hierarchy declares that a many-to-one relationship exists between the Qtr and Year columns in the Period table. This relationship is valid because every row in the Period table that contains a given value in the Qtr column has the same value in the Year column. For example, when the Qtr value is Q1_98, the Year value is always 1998.



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 4-10](#).

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. Based on 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).

A hierarchy can be dropped with the DROP HIERARCHY command. 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 [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) as totals
from table_name
group by from_column;
```

If the result of the Count column 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) as totals
from period
group by year, qtr
order by year;
```

QTR	TOTALS
Q1_98	1
Q2_98	1
Q3_98	1
Q4_98	1
Q1_99	1
Q2_99	1
Q3_99	1
Q4_99	1
Q1_00	1

If you are running this 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 totals
from period
group by qtr
having count_col <> 1;
```

QTR	TOTALS
-----	--------

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) as totals
from store
group by store_type;
```

STORE_TYPE	TOTALS
Large	4
Medium	5
Small	8

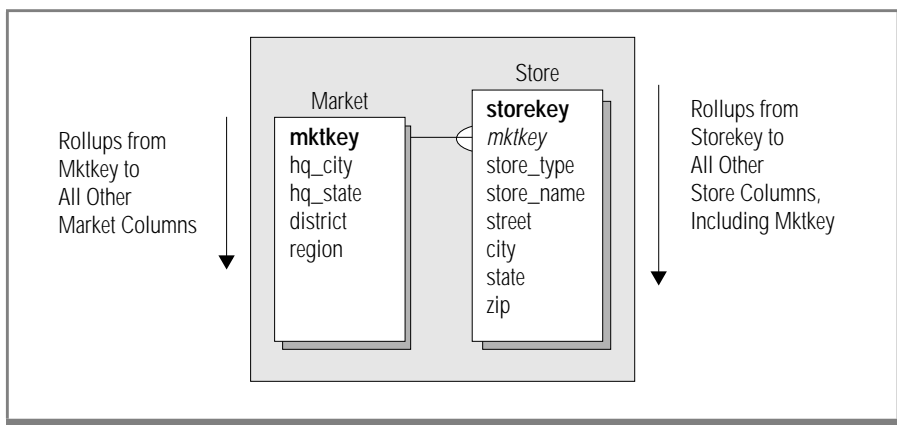
Note that the Mktkey column, which is a foreign key in the Store table and the primary key of the Market table, is the argument of the COUNT(DISTINCT) function in this case. The results of this query demonstrate that a hierarchy from the Store_Type column to *any column* in the Market table would not 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 3-10](#) 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, *any column* in the Store table, or both 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 4-27](#).

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 logically “aggregated.” 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, say 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.



Important: 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 STARindex that can join the family of aggregate tables; this index is equivalent to the STARindex that joins the detail tables in the same schema.

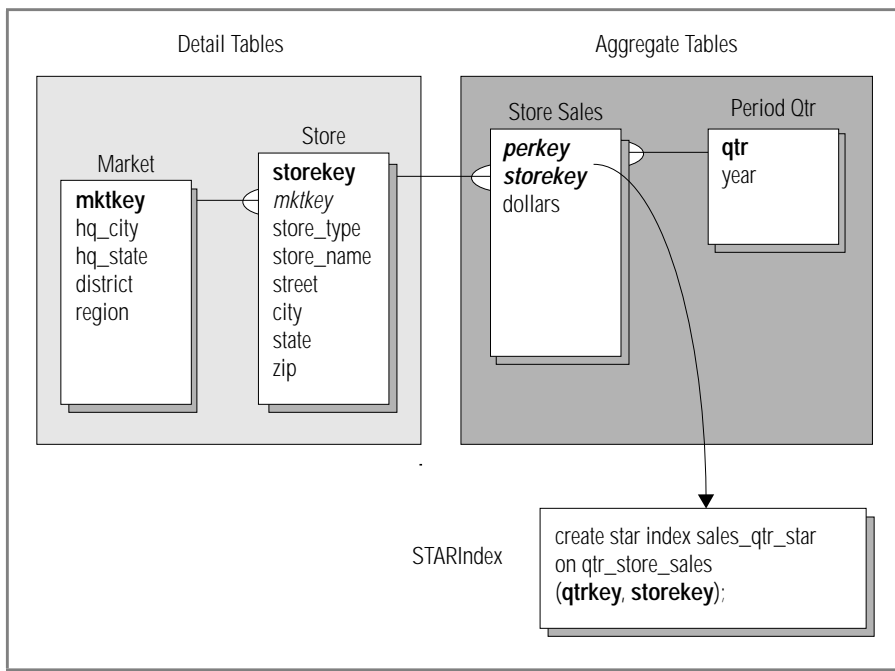
To summarize, if the administrator creates a precomputed view linked to an aggregate Sales table grouped by the Qtr column, the following additional objects will optimize rewrites of queries grouped by Year:

- 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 3-14](#).
- A STARindex on the aggregate Sales table.

Without the derived dimension and the STARindex, queries that group by Qtr and Year would still be rewritten, but queries grouped by Year might be rewritten with suboptimal 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 4. Optimizing with Derived Dimensions](#)” on [page 4-22](#) 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;
```



Important: 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);
```



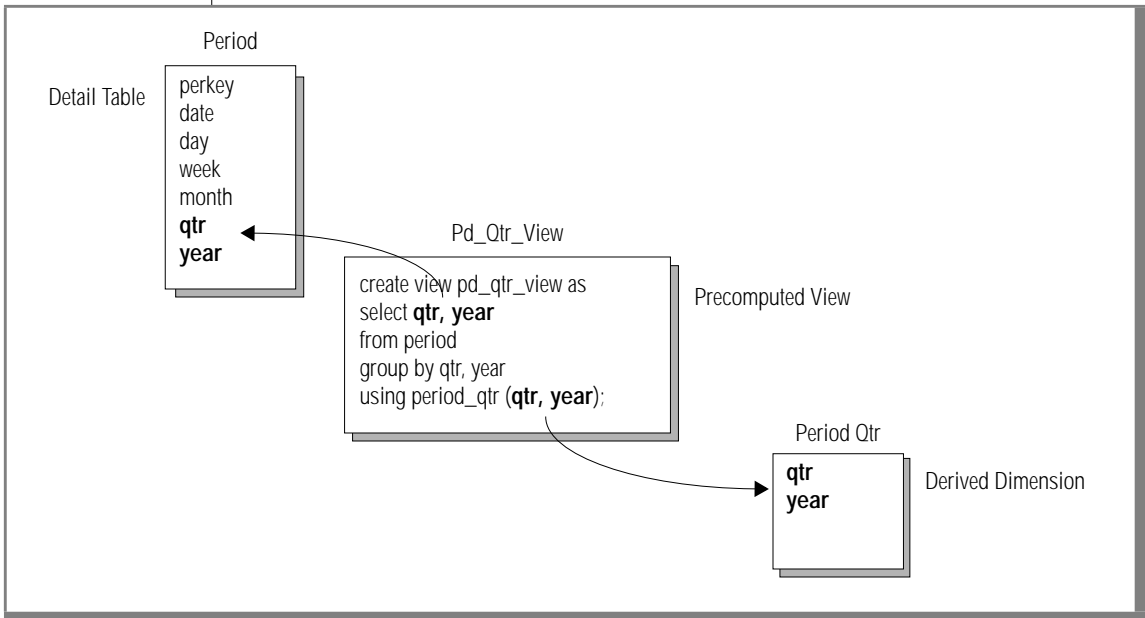
Warning: This kind of precomputed view must be defined with a GROUP BY clause. The following equivalent query expression uses the DISTINCT function in its select list and will return a syntax error despite the fact that it computes the same result set.

```
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 Period_Qtr:



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, Vista must be used in conjunction with standard Red Brick performance and tuning techniques, especially STARjoin and TARGETjoin query processing. Indexes are particularly important when families of aggregate tables are used.

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
- STARindexes
- TARGETindexes, including those for TARGETjoin processing.

For general indexing instructions, see the [Administrator's Guide](#).

Setting Up the Query-Rewriting Environment

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

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 a specific 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 views associated with a 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. For additional information on SET commands and their associated configuration parameters, refer to the [Administrator's Guide](#).

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 database 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 then:

1. Populates the Store_Sales aggregate table with an INSERT statement that selects from the Sales table.
2. Creates the Store_Sales_View, using the same query expression as the INSERT statement.
3. 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.

Granting Select Privileges on Precomputed Views

The DBA must grant Select privilege on a precomputed view to users before their queries can take advantage of that precomputed view. Specifically, Vista qualifies a user query for rewrite only if the user has Select privilege on:

- each table referenced by the query, and
- a precomputed view that could be used to rewrite the query.

The DBA *must* grant Select privileges on precomputed views to any user whose queries might benefit from automatic rewrite by Vista. The DBA grants this privilege using the GRANT command. For additional information on this command, refer to the [SQL Reference Guide](#).

Turning On the Query Rewrite System

The administrator turns on the query rewrite system by using the following SET command or its corresponding configuration parameter:

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

The default behavior of the database server is to rewrite queries. If you do not want queries to be rewritten, you must set this command (or the corresponding 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 ** (1462) 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.

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.



Important: *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, and 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
...
```

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

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

NAME                                PRECOMPCVIEW_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 [Administrator's Guide](#).

Making Precomputed Views Invisible to Client Tools

When Open Database Connectivity (ODBC) client applications are used to query a Red Brick Decision Engine 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.

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



Checklist of Query-Rewriting Tasks

To use the query rewrite system:

1. Enable the Vista with a license key.
2. Create and populate aggregate tables:
 - a. Use the CREATE TABLE command to create the tables.
 - b. Use the TABLE MANAGEMENT UTILITY or an SQL insert statement to populate the tables.
3. Create precomputed views with the CREATE VIEW...USING command.
4. Declare any necessary functional dependencies by using the CREATE HIERARCHY command.
5. Optimize query-rewriting performance:
 - a. Create derived dimensions.
 - b. Create indexes on aggregate tables.
6. Set up the query-rewriting environment:
 - a. Mark precomputed views valid.
 - b. Grant select privilege on precomputed views to users.
 - c. Turn on the query rewrite system.
 - d. Generate statistics for rewritten queries.

Query Rewrite Case Studies

In This Chapter	4-3
General Instructions	4-4
Troubleshooting	4-4
Case 1. Rewriting a Basic Query	4-5
The Query	4-5
Step 1. Create the Aggregate Table	4-5
Step 2. Populate the Aggregate Table	4-6
Step 3. Create the Precomputed View	4-6
Step 4. Mark the Precomputed View Valid	4-6
Step 5. Submit the Query and Note the Performance Gain.	4-7
Step 6. Experiment with Other Query Rewrites	4-8
Case 2. Making Use of Explicit Hierarchies	4-10
The Query	4-11
Step 1. Create the Aggregate Table and Precomputed View	4-11
Step 2. Create the Hierarchy	4-11
Step 3. Create Additional Hierarchies.	4-11
Step 4. Experiment with Other Query Rewrites	4-12
Case 3. Defining Hierarchies on Date	4-13
The Queries	4-14
Step 1. Create the Aggregate Table and Precomputed View	4-14
Step 2. Create the Calendar Hierarchy	4-15
Step 3. Mark the View Valid and Grant Select Privilege.	4-15
Step 4. Watch Vista Rewrite Queries	4-16
Extend the Hierarchy to Month	4-17
Fiscal Periods	4-18
Step 1. Include the Functional Dependencies in the Hierarchy	4-19
Step 2. Watch Vista Rewrite Queries	4-20

Case 4. Optimizing with Derived Dimensions.	4-22
The Queries	4-22
Step 1. Create the Aggregate Tables	4-23
Step 2. Load the Aggregate Tables	4-24
Step 3. Create the Precomputed Views	4-24
Step 4. Create a STARindex on the Aggregate Fact Table	4-25
Step 5. Create Explicit Hierarchies	4-25
Step 6. Validate the Precomputed View	4-25
Step 7. Note the Simplified SQL	4-26
Case 5. Using Implicit Hierarchies.	4-27
The Queries	4-27
Step 1. Create the Aggregate Table	4-28
Step 2. Populate the Aggregate Table	4-29
Step 3. Create a STARindex on the Aggregate Table	4-29
Step 4. Create the Precomputed View	4-29
Step 5. Mark the Precomputed View Valid	4-29
Step 6. Submit the Queries and Note the Performance Gain	4-30
Case 6. Rewriting Subqueries	4-30
The Query and Result Set	4-30
Step 1. Create the Aggregate Table and View	4-31
Step 2. Run the Query	4-31
Case 7. Rewriting a Query That Calculates Averages	4-31
The Query and Result Set	4-32
Step 1. Create the Aggregate Table	4-32
Step 2. Populate the Aggregate Table	4-32
Step 3. Create the Precomputed View	4-32
Step 4. Mark the Precomputed View Valid	4-33
Step 5. Submit the Queries and Note the Performance Gain	4-33
Case 8. Using the EXPORT Command	4-33
The First Method	4-33
Step 1. Unload the Summarized Data	4-34
Step 2. Create the Aggregate Table	4-34
Step 3. Load the Summary Table with the TMU	4-35
An Alternative Method	4-35
Step 1. Unload the Summarized Data	4-35

In This Chapter

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 contains the following sections.

- [General Instructions](#)
- [Case 1. Rewriting a Basic Query](#)
- [Case 2. Making Use of Explicit Hierarchies](#)
- [Case 3. Defining Hierarchies on Date](#)
- [Case 4. Optimizing with Derived Dimensions](#)
- [Case 5. Using Implicit Hierarchies](#)
- [Case 6. Rewriting Subqueries](#)
- [Case 7. Rewriting a Query That Calculates Averages](#)
- [Case 8. Using the EXPORT Command](#)

General Instructions

Each case in this chapter is a tutorial you can work through using the RISQL Entry Tool connected to the Aroma database. These steps apply to each case.

1. Start by setting up the database: create aggregate tables, precomputed views, indexes, and hierarchies.
2. Populate the aggregate tables using INSERT INTO statements.
3. Mark the precomputed views as valid by using a SET command and grant select privileges to those users who would like to use Vista.
4. Generate detailed statistics for queries with the `set stats info` statement. The statistics indicate whether a query has been rewritten.
5. Run a few queries with the query rewrite system turned on.
6. Run the same queries with the query rewrite system turned off and compare their performance.

Scripts for these example are included as part of the Aroma database. A separate script for each example is located in the directory:

```
/redbrick_dir/sample_input/vista_examples
```

Each script contains SQL statements to create and insert data into the tables, SET statements to enable the query rewriting environment, and a SELECT statement for each query. These examples are intended to be run in the order that they occur in this guide.

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 user have select privilege on the precomputed views?
- Does the query constrain on a column “unknown” to the view? (For an example, see [page 4-10](#).)
- Does the query fall into one of the classes of queries that cannot be rewritten? (See [page 3-12](#).)

Case 1. Rewriting a Basic Query

The first example demonstrates the simplest case where 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 aggregate table.

The Query

The following query returns quarterly sales 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. The result set is returned in a matter of seconds, but because this kind of report is requested routinely by several users, the administrator wants to obtain the result set faster and reduce demand on the database server.

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 illustrate how the administrator achieves the objective.

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)not null,
qtr         char(5)not null,
dollars     dec(13,2) not null,
constraint qt_pk primary key (store_name, qtr));
```

Step 2. Populate the Aggregate Table

Compared with an actual database, Aroma is small, so the aggregate table can be loaded with an INSERT INTO...SELECT statement:

```
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);
```

Step 4. Mark the Precomputed View Valid

Make the precomputed view available to the query rewrite system by using the following SET command:

```
set precomputed view quarterly_store_sales_view valid;
```

This statement marks only the Quarterly_Store_Sales_View valid. At this point, Vista will only rewrite queries using this precomputed view for users who have select privilege on this view.

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 a small aggregate table. 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
Moulin Rouge Roasting	Q1_99	46671.25
Moon Pennies	Q1_99	30203.00
The Coffee Club	Q1_99	29623.25
...		
Miami Espresso	Q4_98	53709.50
Coffee Brewers	Q4_98	45573.75
Beans of Boston	Q4_98	44972.25
Olympic Coffee Company	Q4_98	58932.20

```
** INFORMATION ** (1462) SQL statement was rewritten to use one or
more precomputed views.
```

```
** STATISTICS ** (500) Time = 00:00:00.02 cp time, 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_99	61974.40
San Jose Roasting Company	Q2_99	60340.70
Olympic Coffee Company	Q3_99	60070.50
Olympic Coffee Company	Q4_99	59998.60
...		

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.

```
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_99	837699.75
Q4_99	822765.35
Q3_99	822268.35
Q1_00	807390.40
Q1_99	797257.60
Q4_98	782359.05
Q3_98	778795.20
Q2_98	756282.05
Q1_98	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 '%98%'
group by qtr
order by total_sales desc;
```

QTR	TOTAL_SALES
Q4_98	782359.05
Q3_98	778795.20
Q2_98	756282.05
Q1_98	723532.35

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 RANK 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	TOTAL_SALES	SALES_RANK
Q2_99	837699.75	1
Q4_99	822765.35	2
Q3_99	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, year, 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, year;
```

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 4-5](#)).

```
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 4-5](#).

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 *three* legal relationships:

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

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

- **Period.** Qtr and Year columns
- **Store table.** Store_Name, City, and Store_Type columns
- **Market table.** All columns (outboard table referenced by Store)

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 3-14](#)). 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, qtr, sum(dollars) as total_sales
from sales, period, store
where sales.perkey = period.perkey
      and sales.storekey = store.storekey
group by city, qtr;
```

- **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;
```

- **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. Defining Hierarchies on Date

The Period dimension in Aroma represents each quarter of the year as a character string (Q1_99). When users talk about quarters, however, they usually say the first, second, third and fourth quarter. A query can exclude any possible ambiguity by grouping on quarter *and* year. This section shows you how to define a hierarchy on dates when quarter is represented by a simple integer value.

The base example defines a calendar hierarchy on the Date column. Using this hierarchy, Vista can rewrite a query that groups by quarter *and* year to use an aggregate table that groups by Date. The example illustrates a more general principle that if a set of columns A, B, ..., K individually depend on the same column X, then Vista can rewrite queries that group any subset of those columns. That is, A functionally depends on X, B on X, ..., K on X. After the base calendar hierarchy has been established, the example is extended to include months. The example concludes by adding fiscal periods that are defined by weeks to the hierarchy.

To illustrate this case, two new columns (Cal_Qtr and Fis_Qtr) are added to the Period table. The script necessary to add these columns is included in the script `vista_case3.txt`.

The Queries

The base calendar hierarchy allows Vista to rewrite queries that group by calendar quarter (Cal_Qtr) and Year to use an aggregate table grouped by Date. For example, Vista can rewrite queries such as:

```
select cal_qtr, store_name, sum(dollars) as total_sales
from sales natural join period natural join store
      natural join market
where   market.region = 'South'
      and year = 1998
group by year, cal_qtr, store_name
order by total_sales desc;
```

Vista rewrites this kind of query to use the Store_Summaries aggregate table rather than the Sales detail table.

Step 1. Create the Aggregate Table and Precomputed View

The aggregate table contains daily totals for each store.

```
create table store_summaries
(date          date not null,
 store_name    char(30) not null,
 dollars       dec(13,2) not null,
 quantity      integer not null,
 primary key (date, store_name));
```

This table has a finer granularity than the Quarterly_Store_Sales table but Vista can use it to rewrite a much larger class of queries. The reduction factor for this aggregate table is also favorable.

You can insert aggregate totals into this table using the following statement.

```
insert into store_summaries
select date, store_name, sum(dollars), sum(quantity)
from sales natural join store natural join period
group by date, store_name;
** INFORMATION ** (209) Rows inserted: 13871.
```

You can create the precomputed view for the Store_Summaries aggregate table using the following statement.

```
create view store_summaries_view
(date, store_name, dollars, quantity) as
(select date, store_name, sum(dollars), sum(quantity)
from sales, store, period
where sales.storekey = store.storekey
and sales.perkey = period.perkey
group by date, store_name)
using store_summaries
(date, store_name, dollars, quantity);
```

The next step creates the base calendar hierarchy.

Step 2. Create the Calendar Hierarchy

Vista can rewrite queries that group by quarter and year provided that the functional dependencies of Cal_Qtr on Date and Year on Date are defined by a formal hierarchy. The following statement creates the necessary hierarchy.

```
create hierarchy calendar
(from period(date) to period(cal_qtr),
from period(date) to period(year));
```

Each clause declares a functional dependency known by the DBA to obtain in the physical data. You can use the Administrator Tool to verify that the hierarchy actually obtains.

Step 3. Mark the View Valid and Grant Select Privilege

The DBA can mark the precomputed view valid using:

```
set precomputed view store_summaries_view valid;
```

This statement marks the only Store_Summaries view valid. After testing the precomputed view, the DBA needs to grant select privileges on the precomputed view either to users in general or to a selected few.

Step 4. Watch Vista Rewrite Queries

Vista can now rewrite queries that group Sales data by Date, Cal_Qtr, Year, and combinations of these columns. The following examples illustrate the kind of queries Vista can now rewrite to use the Store_Summaries aggregate table. These examples also depend on the store_name and qtr_region hierarchies created for “[Case 2. Making Use of Explicit Hierarchies](#)” that begins on [page 4-10](#).

Group by Calendar Quarter

```
select cal_qtr, 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'
      and year = 1998
group by year, cal_qtr, store_name
order by total_sales desc;
```

CAL_QTR	STORE_NAME	TOTAL_SALES
1	Moulin Rouge Roasting	44353.25
1	Texas Teahouse	46899.25
1	Miami Espresso	52785.75
1	Olympic Coffee Company	55934.35
2	Texas Teahouse	46719.00
2	Moulin Rouge Roasting	47421.00
...		

Group by Quarter and Year with a Case Function

Reports often pivot a report using a CASE function *within* an aggregate function. For example, this report displays quarterly aggregate totals for different cities.

YEAR	CAL_QTR	BOSTON	NORLEANS	HOUSTON
1998	1	43551.50	44353.25	46899.25
1998	2	49419.50	47421.00	46719.00
1999	1	46764.50	46671.25	46484.50
1999	2	48986.25	45378.00	48356.25
2000	1	46797.25	46143.75	47013.00

The following query returns the above report.

```
select  year, cal_qtr,
        sum(case when city = 'Boston' then
              dollars else 0 end) as Boston,
        sum(case when city = 'New Orleans' then
              dollars else 0 end) as NOOrleans,
        sum(case when city = 'Houston' then
              dollars else 0 end) as Houston
from    sales natural join store natural join period
where   cal_qtr in (1, 2)
        and store_type = 'Medium'
group by year, cal_qtr
order by year, cal_qtr;
```

Vista rewrites this query to use the `Store_Summaries` table.

Extend the Hierarchy to Month

You can modify the calendar hierarchy so that Vista also rewrites queries that group by month. The following statements create the revised hierarchy.

```
drop hierarchy calendar;
create hierarchy calendar
  (from period(date) to period(month),
   from period(month) to period(cal_qtr),
   from period(date) to period(year));
```

This hierarchy declares that `Cal_Qtr` depends on `Month` rather than on `Date`; but because `Month` depends on `Date`, so does `Cal_Qtr`. The dependency relationship is transitive.

Using the revised hierarchy, Vista can rewrite all the previous queries plus those that group by Month. For example, Vista rewrites the following query (a variation on the previous query), which groups by Cal_Qtr and Month.

```
select  cal_qtr, month,
        sum(case when city = 'Boston' then
                dollars else 0 end) as Boston,
        sum(case when city = 'New Orleans' then
                dollars else 0 end) as NOrleans,
        sum(case when city = 'Houston' then
                dollars else 0 end) as Houston
from    sales natural join store natural join period
where   cal_qtr in (1, 2)
        and year = 1999
        and store_type = 'Medium'
group by cal_qtr, month
order by cal_qtr;
```

CAL_QT	MONTH	BOSTON	NORLEANS	HOUSTON
1	MAR	16669.00	17526.25	14634.25
1	FEB	13657.75	13109.00	14581.00
1	JAN	16437.75	16036.00	17269.25
2	JUN	15392.50	14117.25	15494.75
2	MAY	17043.00	16132.75	17479.75
2	APR	16550.75	15128.00	15381.75

Vista can now rewrite a large number of queries to use the Store_Summaries aggregate table rather than the Sales table.

Fiscal Periods

Some users analyze by fiscal rather than calendar periods. Fiscal periods are based on weeks instead of months, a fiscal quarter consisting of thirteen weeks rather than three months. Although weeks fail to strictly align with months, and fiscal quarters with calendar quarters, the calendar hierarchy can be readily extended to cover fiscal periods by defining the necessary functional dependencies of Week on Date, and Fis_Qtr on Week.

Step 1. Include the Functional Dependencies in the Hierarchy

To extend the calendar hierarchy, you need to include only two additional dependencies: Week on Date and Fis_Qtr on Week.

```
drop hierarchy calendar;
create hierarchy calendar
  (from period(date) to period(month),
   from period(month) to period(cal_qtr),
   from period(date) to period(year),
   from period(date) to period(week),
   from period(week) to period(fis_qtr));
```



Warning: The administrator must verify that the physical relationships obtain; otherwise, rewritten queries can return incorrect result sets.

You can look at how the dependencies are defined by writing a query to access the system table that maintains hierarchies.

```
select from_table, from_column, to_table, to_column
from rbw_hierarchies
where name = 'CALENDAR';
```

FROM_TABLE	FROM_COLUMN	TO_TABLE	TO_COLUMN
PERIOD	DATE	PERIOD	MONTH
PERIOD	MONTH	PERIOD	CAL_QTR
PERIOD	DATE	PERIOD	YEAR
PERIOD	DATE	PERIOD	WEEK
PERIOD	WEEK	PERIOD	FIS_QTR

The Administrator Tool can retrieve the same information and display it in a much nicer format. Connect with Aroma, expand the Aroma folder, the Hierarchies folder, and finally the folder for the Calendar hierarchy. The Administrator graphs the relationships from coarser to finer granularity.

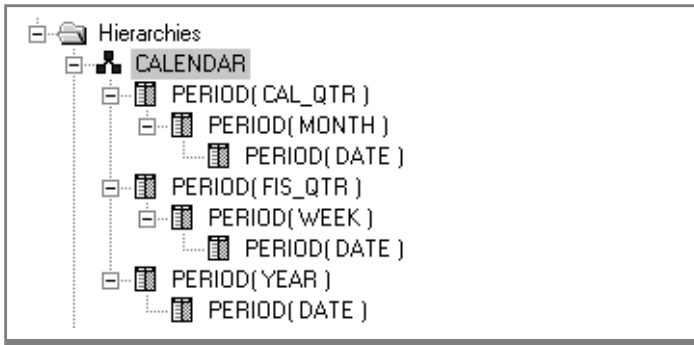


Figure 4-1
Calendar Hierarchy

Step 2. Watch Vista Rewrite Queries

Vista can now rewrite queries that group data in the Sales fact table by the following columns:

- Date
- Month
- Cal_Qtr
- Week
- Fis_Qtr
- Year

Whenever a user query groups Sales data by one of these columns, Vista will attempt to rewrite the query to use the Sales_Summaries aggregate table.

For example, a user query can calculate aggregate totals for the first two calendar quarters of 1998 and 1999 by grouping the Sales table by Cal_Qtr and Year. Vista will intercept and rewrite the query. Expect the rewritten queries to run much faster.

Group by Fis_Qtr

This query compares first quarter totals for three separate years. Vista rewrites the query to use the Store_Summaries aggregate table.

```
select  year, fis_qtr,
        sum(case when city = 'Boston' then
                dollars else 0 end) as Boston,
        sum(case when city = 'New Orleans' then
                dollars else 0 end) as NOlleans,
        sum(case when city = 'Houston' then
                dollars else 0 end) as Houston
from    sales natural join store natural join period
where   fis_qtr = 1
        and year in (1998, 1999, 2000)
        and store_type = 'Medium'
group by year, fis_qtr
order by year, fis_qtr ;
```


Query Result Set

YEAR	FIS_QTR	BOSTON	NORLEANS	HOUSTON
1998	1	42451.00	43036.75	44686.25
1999	1	44873.00	44786.50	44396.25
2000	1	43729.25	43460.75	42790.50

Group by Week

```

select  week,
        sum(case when city = 'Boston' then
                dollars else 0 end) as Boston,
        sum(case when city = 'New Orleans' then
                dollars else 0 end) as NOrleans,
        sum(case when city = 'Houston' then
                dollars else 0 end) as Houston,
        sum(case when city = 'Phoenix' then
                dollars else 0 end) as Phoenix
from    sales natural join store natural join period
where   month = 'JAN'
        and year = 1999
        and store_type = 'Medium'
group by week, year
order by week, year;

```

Query Result Set

WEEK	BOSTON	NORLEANS	HOUSTON	PHOENIX
1	647.50	227.75	522.75	456.00
2	4185.25	3197.25	4537.00	4036.00
3	2930.00	4283.25	4112.00	3501.50
4	4378.25	3870.50	4129.75	3446.75
5	4014.50	3685.00	3582.00	2798.50
6	282.25	772.25	385.75	663.00

Case 4. Optimizing 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;
```

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;
```

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;
```



Important: *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;
```

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 STARindex on the Aggregate Fact Table

Create a STARindex 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 and should be created to improve query performance.

Step 5. Create Explicit Hierarchies

Define the same set of hierarchies that were created in Steps 2 and 3 for Case 2 (see [page 4-11](#)).



Important: 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, the City column of the Store table, or both.

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;
```

Now turn on the query rewrite system and note the performance gain when you run the queries on [page 4-22](#).

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.

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 5. Using Implicit Hierarchies

Cases 1 through 4 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 = 1999
group by day;
```

- 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 = 1999
group by date, store_name;
```

■ 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 = 1999
group by store_type, month
order by total_sales desc;
```

■ Sales per city per month:

```
select city, month, sum(dollars) as total_sales
from sales, store, period
where sales.storekey = store.storekey
      and sales.perkey = period.perkey
      and year = 1999
group by city, month;
```

■ Sales per district per year:

```
select district, year, sum(dollars) as total_sales
from sales, store, period, market
where sales.storekey = store.storekey
      and store.mktkey = market.mktkey
      and sales.perkey = period.perkey
      and year = 1999
group by district, year;
```

To answer all these queries, the server would normally join the tables by using the STARindex on the Sales table. The query rewrite system will substitute the aggregate fact table and its STARindex. 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;
```



Important: 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 STARindex on the Aggregate Table

Create a STARindex on the key columns of the Store_Sales table:

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

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 the following SET command:

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

This SET command marks *all* views that use the Sales table valid.

Step 6. Submit the Queries and Note the Performance Gain

Turn on the query rewrite system, and run the queries listed on [page 4-27](#), as well as any other aggregate query that constrains on columns from the Period, Store, and Market dimensions.

Case 6. 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_q198, sales_q199
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_98'
   group by e1.store_name) as sales1(name, sales_q198),
  (select e2.store_name, sum(dollars)
   from sales, store e2, period
   where sales.storekey = e2.storekey
        and sales.perkey = period.perkey
        and qtr = 'Q1_99'
   group by e2.store_name) as sales2(name, sales_q199)
where sales1.name = sales2.name
order by sales_q198 desc;
```

NAME	SALES_Q198	SALES_Q199
Beaches Brew	57893.00	57152.85
Olympic Coffee Company	55934.35	52544.80
San Jose Roasting Company	55763.25	53188.70
Miami Espresso	52785.75	58498.50
Texas Teahouse	46899.25	46484.50
Java Judy's	46458.00	48760.50
Moulin Rouge Roasting	44353.25	46671.25
Cupertino Coffee Supply	44280.75	48155.50
Moroccan Moods	44065.00	43947.50
Beans of Boston	43551.50	46764.50
Instant Coffee	43129.50	42400.50
Roasters, Los Gatos	43011.50	40725.50
Coffee Brewers	40955.00	42531.00
East Coast Roast	38062.75	41007.75
Moon Pennies	35427.25	30203.00
The Coffee Club	30962.25	29623.25

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 4-5](#).

Step 2. Run the Query

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

Case 7. Rewriting a Query That Calculates Averages

Although you cannot use the AVG set function in a precomputed view definition (see [page 3-8](#)), 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;
```

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 4-32](#). 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.

Case 8. Using the EXPORT Command

All the previous examples in this chapter load aggregate tables using the SQL INSERT statement. You can also summarize and export data from a database using the EXPORT command (a server command) but load the exported data using the Table Management Utility.

The following example shows how you can summarize data with a query and then load the summarized data using the TMU. Two methods are illustrated; the second one can be more efficient than the first.

The First Method

The first method requires three basic steps.

1. Unload the summarized data.
2. Create the aggregate table.
3. Load the summary data using the TMU.

The Store_Summaries aggregate table of case three ([page 4-15](#)) is used to illustrate this method.

Step 1. Unload the Summarized Data

The first step uses an EXPORT statement to summarize and unload data from the Sales table. This statement is executed by the Red Brick server.

```
set export_default_path '/dba/scripts/';
export to 'data' DDLFILE 'create' TMUFILE 'load'
format internal
  (select date, store_name, sum(dollars), sum(quantity)
   from sales natural join store natural join period
   group by date, store_name);
```

The server writes out three files: the first (data) contains the summarized data, the second (create) contains a DDL script that can create a table for the exported data, and the third (load) contains a TMU script that can load the exported data. Both scripts must be modified before they can be used.

For additional information on the EXPORT command, refer to the [SQL Reference Guide](#). The [Table Management Utility Reference Guide](#) covers the TMU.

Step 2. Create the Aggregate Table

The DDL script generated by the EXPORT statement can be used to create the Store_Summaries table. This script contains three dummy names and each must be modified before the script is usable. The dummy names begin with the word 'GENERATED'.

```
CREATE TABLE GENERATED_TABLE (
  DATE DATE,
  STORE_NAME CHARACTER(30),
  GENERATED_COLNAME_3 DECIMAL(13,2),
  GENERATED_COLNAME_4 DECIMAL(16,0));
```

Replace the dummy table name with the name of the aggregate table (Store_Summaries), replace the dummy column names with names of its respective aggregate total columns (Dollars and Quantity), and include a primary key constraint on Date and Store_Name.

You can now use this statement to create the aggregate table.

Step 3. Load the Summary Table with the TMU

The TMU script generated by the EXPORT statement can be used to load the Store_Summaries table. This script contains a single dummy name (GENERATED_TABLE) that must be modified before the script is usable.

```
LOAD DATA INPUTFILE
  'data'
  INSERT
  FORMAT UNLOAD
  NLS_LOCALE 'English_UnitedStates.US-ASCII@Binary'
  INTO TABLE GENERATED_TABLE;
```

Replace the dummy name with the name of the aggregate table (Store_Summaries).

You can now use the TMU to load the table.

```
rb_tmu -d AROMA load
```

Before Vista can use this aggregate table to rewrite queries, you must first create a precomputed view for it, mark it valid, and grant select privilege on it to users. “Case 3. Defining Hierarchies on Date” covered these steps in detail. See [page 4-13](#).

An Alternative Method

The previous method exports data by writing it to a disk file, and then turns around, reads the data from that disk file, and loads it into the aggregate table. You can eliminate the disk reads and writes by piping the exported data directly to the TMU. This approach is simpler and can make the process more efficient.

Step 1. Unload the Summarized Data

The first step uses an EXPORT statement to summarize and export data from the Sales table and then pipe the summarized data directly to the TMU. The following EXPORT statement does the work.

```
export to '|/redbrick/bin/rb_tmu -d AROMA /dba/scripts/pipe'
format internal
  (select date, store_name, sum(dollars), sum(quantity)
   from sales natural join store natural join period
   group by date, store_name);
```

The location of the TMU and its modified control file (/dba/scripts/pipe) must be specified with fully qualified path names. The redbrick directory contains the rbw.config file and the binaries for Red Brick Decision Server.

The Modified TMU Script

Before you can use the above EXPORT statement, you must modify the TMU control file to read its input from a pipe instead of a file. The modification is simple: replace the input file name (data) with a dash (-).

```
LOAD DATA
  INPUTFILE '-'
  INSERT
  FORMAT UNLOAD
  NLS_LOCALE 'English_UnitedStates.US-ASCII@Binary'
  INTO TABLE store_summaries;
```

This script is a modification of the load script on [page 4-35](#).

Before Vista can use this aggregate table to rewrite queries, you must first create a precomputed view for it, mark it valid, and grant select privilege on it to users. “Case 3. Defining Hierarchies on Date” covered these steps in detail. See [page 4-13](#).

Using the Advisor

In This Chapter	5-3
Advisor Overview	5-4
Analysis of Query Patterns	5-4
Advisor System Tables	5-4
Advisor Log Files	5-5
Configuring the Advisor Logging System	5-5
Creating the Advisor Log Files	5-5
Logging Queries	5-6
Rewritten Queries	5-6
Candidate Views	5-6
Correlated Subqueries	5-6
Queries That Are Rewritten But Not Logged	5-7
Setting the ACCESS_ADVISOR_INFO Task Authorization	5-9
Defining Valid Hierarchies	5-10
Querying the Advisor	5-10
Inserting the Results of an Advisor Query into a Table	5-10
Creating and Populating the CANDIDATE_TEMP Table	5-12
Creating and Populating the UTILIZATION_TEMP Table	5-13
Querying the RBW_PRECOMPVIEW_UTILIZATION Table	5-13
Rules for Querying the Utilization Table	5-14
NON_EXACT_MATCH_COUNT Column	5-14
Querying the RBW_PRECOMPVIEW_CANDIDATES Table	5-15
Rules for Querying the Candidates Table	5-15
SAMPLE_VIEW_NAME Column	5-16

Interpreting the Results of Advisor Queries	5-21
BENEFIT Column	5-21
SIZE and REDUCTION_FACTOR Columns	5-21
REFERENCE_COUNT Column	5-22
Combining the Results	5-23
Understanding the BENEFIT Column	5-23
How the BENEFIT Column Is Calculated	5-24
What the Numbers Mean	5-25
Uniform Probability	5-26
Example	5-27
Advisor System Table Column Descriptions	5-29
RBW_PRECOMPVIEW_CANDIDATES Table	5-29
RBW_PRECOMPVIEW_UTILIZATION Table	5-30
Checklist of Advisor Tasks	5-32

In This Chapter

The Informix Vista 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 Vista, aids in figuring out the best aggregate tables for your database, whether those tables currently exist or not. Because the Advisor knows exactly what types of queries can be rewritten with the 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 data.

There is a cost to every precomputed view, as well as a benefit to having precomputed views 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 5-29.](#)

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 5-10](#).

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 [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 on 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.



Important: 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 Vista 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);

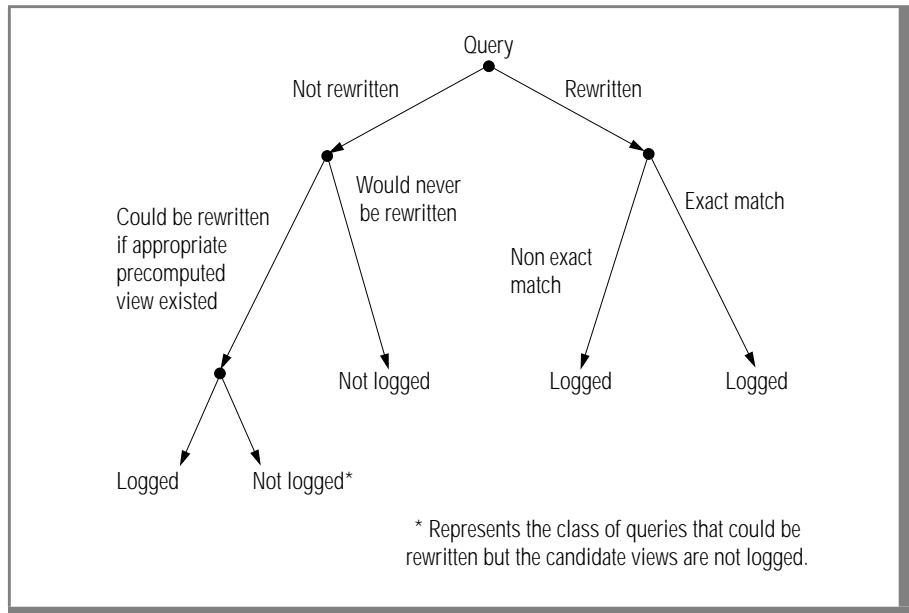
insert into simple_table
  (hq_state, hq_city, year, month, sum_dollars)
  (select market.hq_state as hq_state, market.hq_city as
hq_city,
   period.year as year, period.month as month,
   sum(sales.dollars) as sum_dollars
  from sales, market, period, store
 where sales.perkey = period.perkey
     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



Important: 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:



For information on what constitutes an exact match, refer to [“NON_EXACT_MATCH_COUNT Column” on page 5-14](#).

Setting the ACCESS_ADVISOR_INFO Task Authorization

All users with the DBA system role have the necessary privileges to query the Advisor system tables. 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;
```



Important: 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.

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 2-10](#).

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 on 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.

You can further reduce the analysis time by constraining the extraction query to a time period defined by start and end dates. The date can be a date or a time stamp.

For example, the following query limits the extraction of data to a week in July of 1999.

```
insert into temporary_table
select
---
where  detail_table_name = 'SALES'
       and start_date = '1999-07-11'
       and end_date = '1999-07-18';
```

When the query is processed, the server converts its dates to timestamps so that '1999-07-11' is converted to '1999-07-11 00:00:00.000000'.

The conversion of dates to timestamps can yield unexpected results. For example, a query that constrains its start and end dates to the same date defines an empty interval (starts and ends at 00 hours on the morning of July the seventh).

You can define an interval that spans a single day ('1999-07-11') by using the dates '1999-07-11' and '1999-07-12'. Or, you can specify two timestamps on the same date ('1999-07-11 00:00:00.000000', '1999-07-11 23:59:59.999999').

Timestamps can be used to define characteristic workload intervals, such as the morning, afternoon, and evening shifts. The following statement uses start and end timestamps to define a morning workload.

```
Insert into temporary_table
select
---
where  detail_table_name = 'SALES'
       and start_date  = '1999-07-11 09:00:00:000000'
       and end_date    = '1999-07-11 11:00:00:000000';
```

The granularity of the interval depends on the purpose of the analysis.

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),  
    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 detail_table_name,  
    aggr_elapsed_time, reference_count,  
    sample_view_name, reduction_factor,  
    size, benefit, name, seq, text  
from rbw_precompview_candidates  
where detail_table_name = 'SALES'  
    and start_date = date '1999-07-01'  
    and end_date = date '1999-08-01';
```

This query restricts the interval to July 1999 and the operation can potentially take a long time to execute.

Typically, queries will run faster against the Candidate_Temp table than against the RBW_PRECOMPVIEW_CANDIDATES table because there is no need to scan the log files. Also, the amount of data is smaller because the interval has been restricted to one month.

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),  
    name char (128),  
    size int,  
    reduction_factor float,  
    benefit float,  
    reference_count int,  
    non_exact_match_count int ) ;
```

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

```
insert into utilization_temp  
select detail_table_name, name, size,  
       reduction_factor, benefit,  
       reference_count, non_exact_match_count  
from rbw_precompview_utilization  
where detail_table_name = 'SALES'  
       and start_date = date '1999-07-01'  
       and end_date = date '1999-08-01';
```

This query restricts the interval to July 1999.

Typically, queries will run faster against the Utilization_Table than against the RBW_PRECOMPVIEW_UTILIZATION table because there is no need to scan the log files. Also, the amount of data is smaller because the interval has been restricted to one month.

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 5-30](#).

Rules for Querying the 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.



Important: If you insert the results of an Advisor query into a table like the Utilization_Temp table shown on [page 5-13](#), 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 column in the RBW_PRECOMPVIEW_CANDIDATES table, refer to [page 5-29](#).

Rules for Querying the 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.



Important: If you insert the results of an Advisor query into a table like the Candidate_Temp table shown on [page 5-12](#), there are no restrictions as to what columns can be constrained on that table; any columns can be constrained.

SAMPLE_VIEW_NAME Column

You can restrict an Advisor candidate analysis to a subset of the detail data. This allows you to base an analysis on a more *representative* sample of data and to decrease the often considerable time required for an analysis.

To Perform an Analysis on a sample of data:

1. Define the sample as a view of the detail data.
 - The sample view must map to a subset of rows in the detail table and include, for each column of the detail table, a corresponding column of exactly the same datatype.
 - The view should proscribe a representative sample of data. Otherwise, the analysis will lack balance.
2. Constrain the SAMPLE_VIEW_NAME COLUMN of the precomputed view candidates table.

This constraint directs the Advisor to use the sample defined by the view instead of the detail data.

The example on [page 5-20](#) shows how to use a sample of data instead of using all the detail data.



Important: When a query constrains the SAMPLE_VIEW_NAME column, the Advisor computes the SIZE and REDUCTION_FACTOR values using the view, not the detail table.

When you define a sample of data as a view, an Advisor query against the view might perform poorly because the definition of the view results in a table scan. For example, if a detail table contains a billion rows of data and the query that defines the view invariably performs a table scan of the billion rows, then expect the Advisor query to perform poorly.

For this kind of case, create a table for the data sample, populate the table with a subset of rows from the detail table, and then create a view for this table. Make sure that you create the appropriate indexes on the sample table, particularly STAR indexes because the Advisor performs numerous query operations when it analyzes the log files. Advisor performance depends on a proper set of indexes.

Example Query Against RBW_PRECOMPVIEW_CANDIDATES

This example queries the Candidate_Temp table that is described on [page 5-12](#). 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:

```
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

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:

```
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  
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.

Example Query Against a Sample of Data

The candidate analysis in the previous example was based on all the existing detail data in the Sales table. The analysis might be biased because the Sales table contains data that spans three years. A smaller, more representative sample of data can yield a more balanced and timely analysis.

To restrict the analysis to a representative sample of data, you need to create a view that defines the sample and constrain the Advisor query to the view. This is straightforward.

1. Create a view that defines a representative sample of data.

Assume that the third week of July provides a realistic picture of the workload. The following view defines this sample.

```
create view sample_week as
select sales.perkey, classkey, prodkey,
       storekey, promokey, quantity, dollars
from sales natural join period
where year = 1999 and week = 29 ;
```

The column names in the view must match those of the Sales detail table exactly.

2. Constrain the Advisor query to the view when you perform an analysis. The following query inserts the results of an analysis into a temporary table.

```
insert into candidate_temp_sample
select detail_table_name,
       aggr_elapsed_time,
       reference_count,
       sample_view_name,
       reduction_factor,
       size, benefit, name, seq, text
from rbw_precompview_candidates
where detail_table_name = 'SALES'
      and sample_view_name = 'SAMPLE_WEEK'
      and start_date = date '1999-04-30'
      and end_date = date '1999-08-01';
```

You can now interrogate and use the results of the analysis.



Important: The time interval defined by the Advisor query restricts the set of queries selected for an analysis to those that occurred during the months of May, June, and July; the time interval defined by the view restricts the sample of data for the analysis to a week in July. The analysis depends to a large extent on how well these time periods represent the workload.

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 database administrator must evaluate this trade-off and decide what precomputed views to create or remove. 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 5-23](#).

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 the view 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.



Important: 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 data management.

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

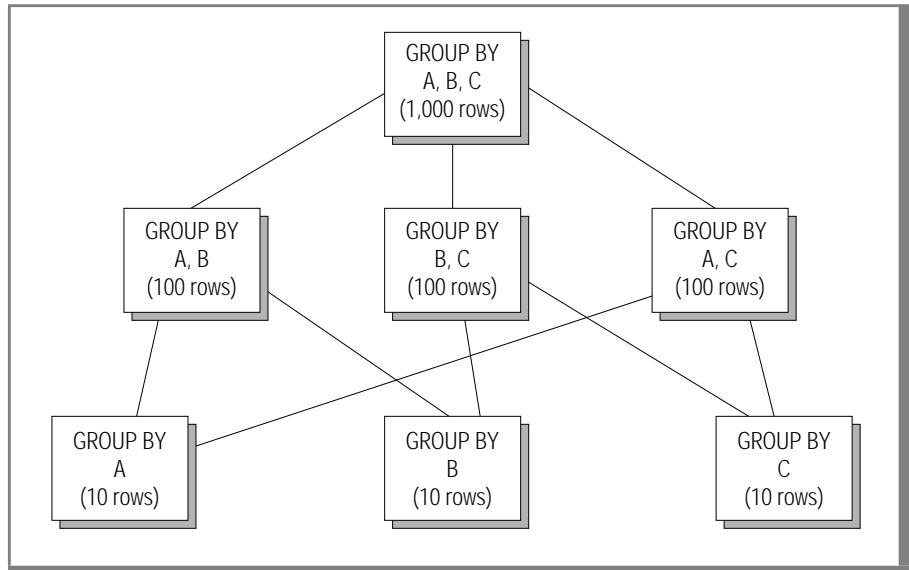
The BENEFIT column of the two Advisor system tables provides a number that indicates the benefit of a precomputed view based on other views that are available. In general, the higher the benefit, the more useful the precomputed view is for your database. The benefit is calculated based on your query 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 following factors:

- 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 other; 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 of that view saves the database from processing 100,000 rows that it would have to process without the view.



Important: 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

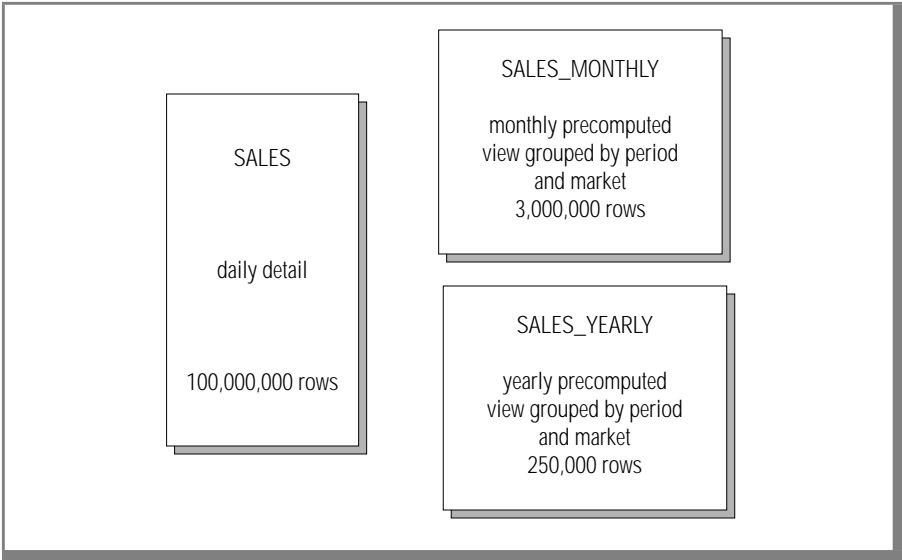
When the SET UNIFORM PROBABILITY FOR ADVISOR command is set to ON, the Advisor does not scan the log files when calculating the value for the BENEFIT column in the RBW_PRECOMPVIEW_UTILIZATION table; instead, it assumes that all precomputed views have been accessed an equal number of times and therefore 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 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.

(1 of 2)

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 the 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.

(2 of 2)

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.
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:

1. Enable the Vista with a license key.
2. Create the Advisor log file (ADMIN ADVISOR_LOGGING ON or ALTER SYSTEM START ADVISOR_LOGGING). See [page 5-5](#).
3. Enable Advisor query logging (OPTION ADVISOR_LOGGING ON or SET ADVISOR LOGGING ON). See [page 5-5](#).
4. Provide authority to access the advisor for the user who will query the Advisor (ACCESS_ADVISOR_INFO task authorization). See [page 5-9](#).
5. Define any explicit hierarchies with CREATE HIERARCHY statements that are valid for your schema. See [page 3-13](#).
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. See [page 5-13](#).
8. Query the RBW_PRECOMPVIEW_CANDIDATES table to analyze precomputed views that, if created, would improve query performance. See [page 5-15](#).
9. Analyze the results of your Advisor system table queries. See [page 5-21](#).
10. Create new precomputed views or remove existing precomputed views, based on your analysis.

Managing Vista with the Administrator

In This Chapter	6-3
Getting Started	6-4
Managing Vista	6-6
Creating an Aggregate Table and Its View	6-9
Creating and Verifying Hierarchies	6-22
The Store to District Hierarchy	6-23
Using the Advisor	6-26
Precomputed View Utilization	6-26
Candidate View Generation	6-31
Generating Candidates from Sampled Data.	6-35

In This Chapter

The Administrator is a tool that provides a graphical user interface to Red Brick Decision Server. This tool simplifies most database administration tasks and efficiently completes these tasks. This tool includes several features that are specific to Vista. This chapter shows you, in a step-by-step fashion, how to use this tool to manage the Vista environment.

This chapter contains the following sections:

- [Getting Started](#)
- [Managing Vista](#)
- [Creating an Aggregate Table and Its View](#)
- [Creating and Verifying Hierarchies](#)
- [Using the Advisor](#)

Getting Started

Choose **Start→Programs→Red Brick→Red Brick Administrator** to start the Administrator Tool. The actual location of the tool depends on options selected by the DBA during installation: ask your DBA for the exact location.

The Administrator Tool returns its primary window, as [Figure 6-1](#) shows.



Figure 6-1
*The
Administrator
Primary Window*

From the toolbar, choose the **Open** file icon, and then select a Database Source Name (DSN) from the popup menu. The Administrator extends its primary toolbar and returns the Source window, as [Figure 6-2](#) shows. You can perform most Vista operations directly from the Source window or the extended toolbar.

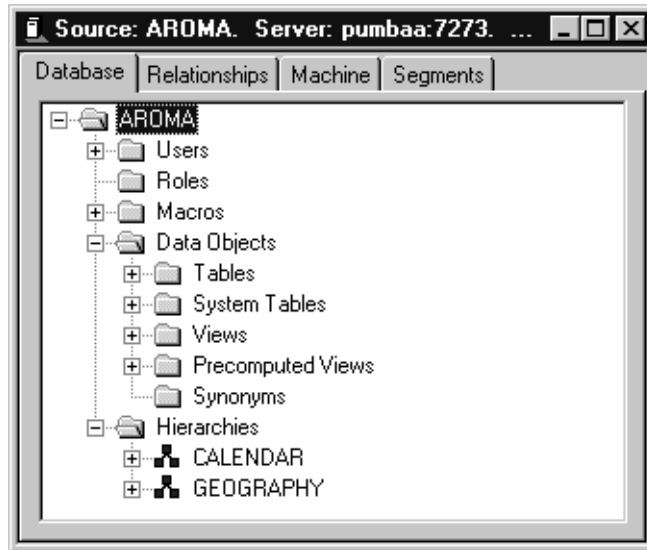


Figure 6-2
*Database Source:
The Initial Window*

Managing Vista

The DBA can perform the following tasks from the primary window.

- Control the Vista execution environment.
- Control the Vista Advisor.
- Set the state of a precomputed view.
- Grant Select privileges to users on precomputed views.

The following examples illustrate the most direct way to perform these tasks.

To Establish the Vista Execution Environment

1. Left-click the **Settings** button on the Administrator tool bar.

The Administrator returns the Vista Settings window. The DBA can control the Vista execution environment from this window, as [Figure 6-3](#) shows.

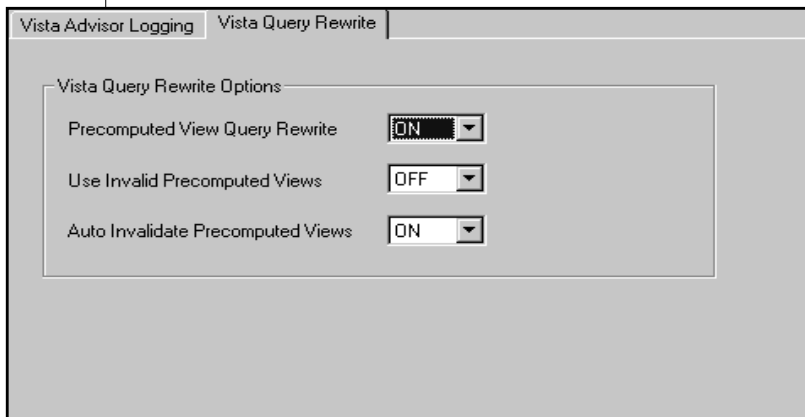


Figure 6-3
Vista Settings

2. Left-click **Vista Query Rewrite** tab to set the state of the rewrite system. From the rewrite panel, click the down arrow and select **On** or **Off** to set an option.
3. Left-click **Vista Advisor Logging** tab to set the state of the Advisor. From the rewrite panel, click the down arrow and select **On** or **Off** to set an option. A discussion of the Advisor begins on [page 6-26](#).

To Set the State of a Precomputed View

You can determine and set the state of a precomputed view from the toolbar or the Source window:

- **Toolbar.** Choose **Manage→PrecomputedViews** from the primary toolbar. The Administrator returns a wizard that lists several options. Select a Validate option and the wizard returns a sequence of pages that prompt you for additional information.
- **Source window.** Expand the **Aroma** and the **Precomputed Views** folder. Right-click on the name of a precomputed view and the Administrator returns a list box. Choose **Validate** or **Invalidate** from the list box.

The most direct method is from the Source window. [Figure 6-4](#) shows the Source window and the popup list box for precomputed views.

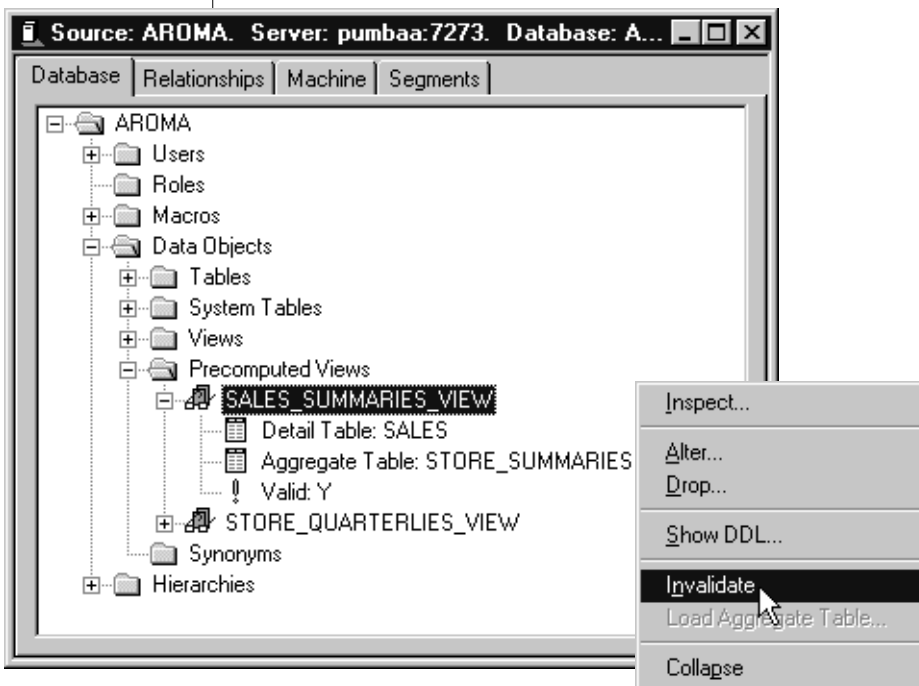


Figure 6-4
Invalidate a
Precomputed View

When you expand the folder for a specific precomputed view, the Administrator lists the name, its detail and aggregate tables, and its current state. If the view is valid, its eyeglasses icon is green; otherwise, it is red.

To Grant Select Privilege on Precomputed View

There are two ways to grant Select privilege on a precomputed view to a user.

- **Toolbar.** Choose **Manage→User** from the primary toolbar. The Administrator returns a wizard that lists several options. Select the Grant option and the wizard returns a sequence of pages that prompt you for additional information.
- **Source window.** Expand the **Aroma** and the **Users** folder. Right-click a user name (for example, **Ralph**) and the Administrator returns a list box. Choose **Grant**. The Administrator returns a wizard that prompts you for specific information.

The second method is the most direct. [Figure 6-5](#) shows the Source window and the list box.

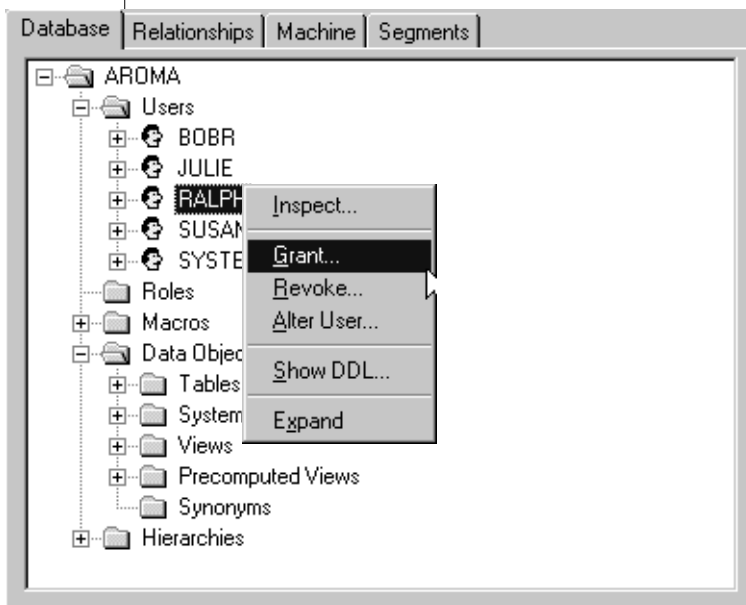


Figure 6-5
*Grant Select
Privilege to a User*

Creating an Aggregate Table and Its View

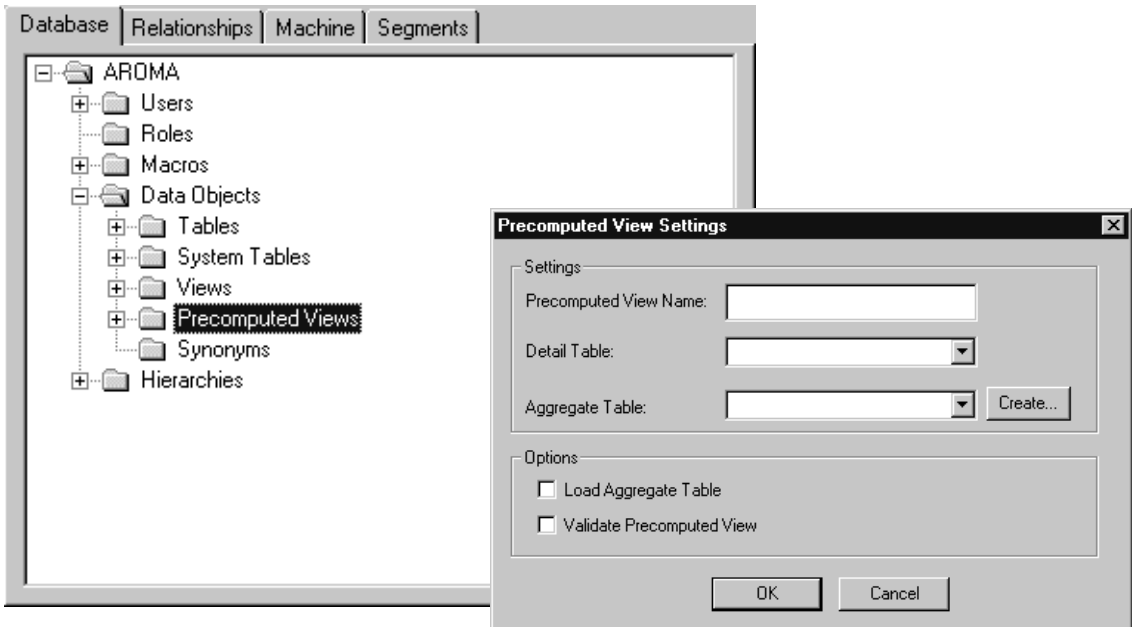
You can use the Administrator Tool to quickly define and create an aggregate table, define a precomputed view for the aggregate table, load the aggregate table, and mark the precomputed view as valid.

The following scenario shows you how to create the `Sales_Summaries_View` described in “[Case 3. Defining Hierarchies on Date](#)” on page 4-13. The aggregate table for this view falls out as an early by-product of the view definition. The process also generates a script that inserts rows into the aggregate table.

To Build the Precomputed View and its Aggregate Table

1. Expand the Aroma folder and right-click **Precomputed Views**. Select **Create** from the popup menu. The Administrator returns a dialog box, as [Figure 6-6](#) shows.

Figure 6-6
Precomputed View Settings Dialog Box



2. Enter the name of the precomputed view, select its detail table from the pull-down list, and check the **Load** and **Validate** check boxes.
You can select an aggregate table from the pull-down list only when the aggregate table exists. This scenario *builds* the aggregate table.
3. Click **OK** (*do not click Create at this time*).

The Administrator returns a window with upper and lower panes, as [Figure 6-7](#) shows. The upper pane contains a placeholder for names of dimension and aggregate columns of the `Sales_Summaries_View`; the lower pane contains a description of the detail table and its columns (`Sales`).

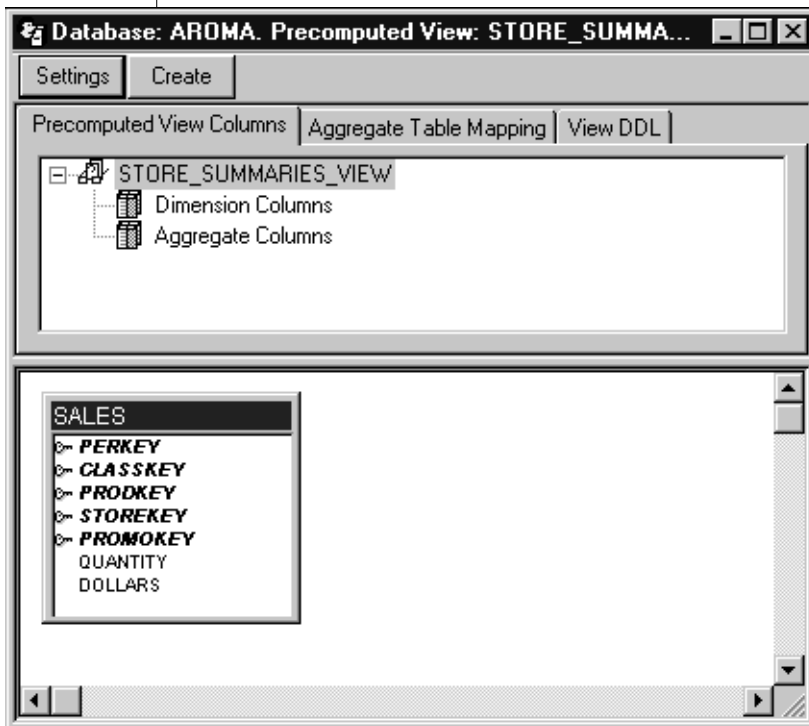


Figure 6-7
*Precomputed View
Window Panes*

Before the columns can be defined for the precomputed view, dimension tables referenced by the detail table must first be displayed in the lower pane.

4. Double-click **Perkey** in the lower pane, and the Administrator returns a popup list of all the tables in Aroma. Select **Period** and **Store** from the list (CTRL-click).

The Administrator includes these tables in the lower pane and graphs the primary and foreign key relationships between the tables, as [Figure 6-8](#) shows.

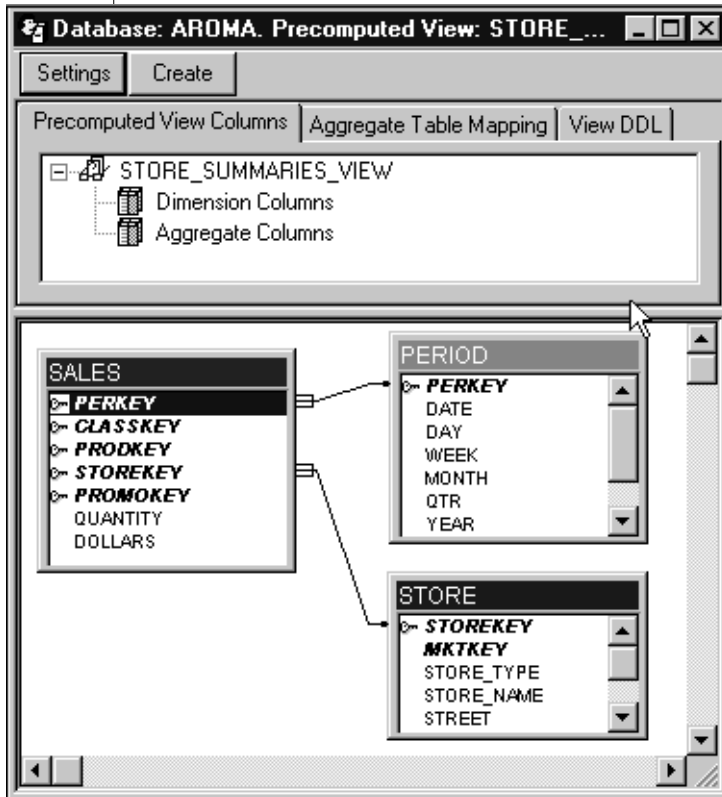


Figure 6-8
Diagrams for the
Dimensions

The following dimension and aggregate columns will be defined for the Store_Summaries aggregate table:

- **Date:** Period.Dateshows
- **Store:** Store.Store_Name
- **Sales:** SUM(Sales.dollars)
- **Qty:** SUM(Sales.quantity)

5. To define a dimension column, drag the column name from the lower pane to the Dimension Column label in the upper pane. The Administrator returns a dialog box that prompts you for additional information, as [Figure 6-9](#) shows.

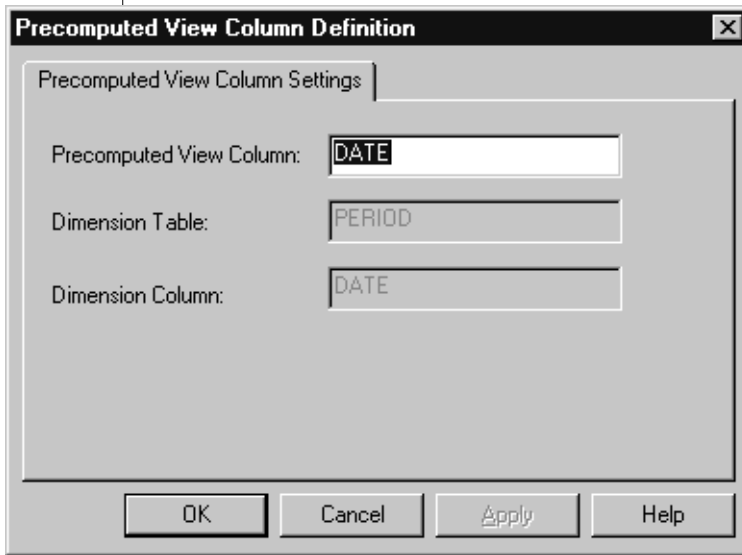


Figure 6-9
*Dialog Box for a
Dimension Column*

The name of a column in a precomputed view can be the same as the corresponding name in the dimension; or it can have a different name. The Administrator returns the dimension column name (Date) as a candidate column name for the precomputed view.

6. Accept the candidate column name or enter a new column name, and then click **OK**.

The Administrator returns a description of the precomputed view column in the upper pane, as [Figure 6-10](#) shows.

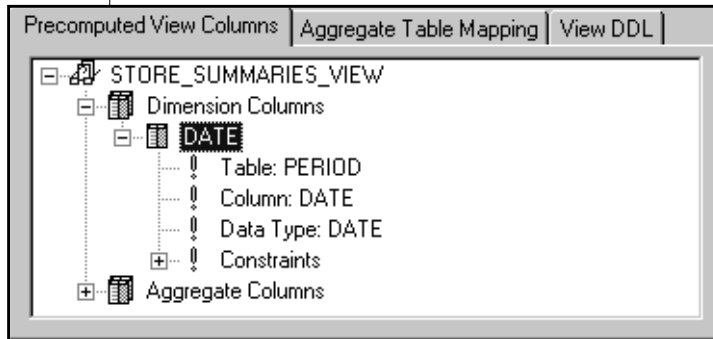


Figure 6-10
*Description of a
Dimension Column*

7. Continue this process until you define all the dimension columns.
8. To define an aggregate column for the precomputed view, right-click **Aggregate Columns** in the upper pane. The Administrator returns another dialog box, as [Figure 6-11](#) shows.

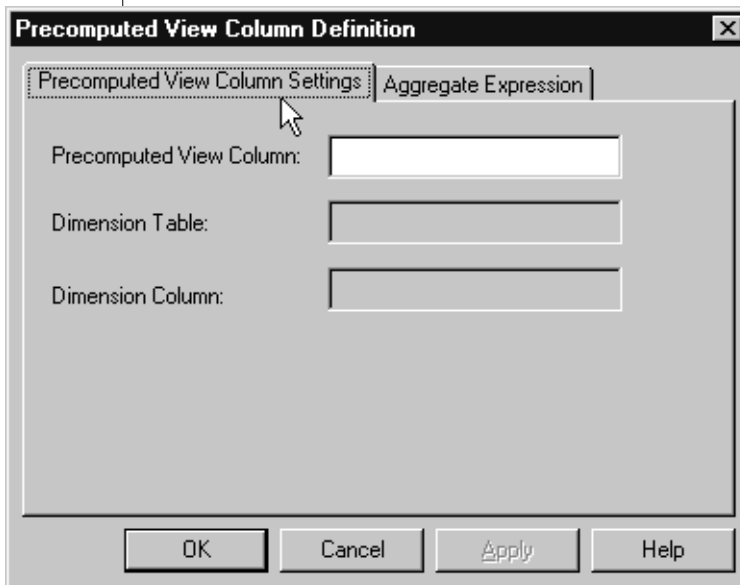


Figure 6-11
*Dialog Box for
Aggregate Column*

9. Enter the column name for the aggregate column and click the **Aggregate Expression** tab. The Administrator returns another dialog box, as [Figure 6-12](#) shows.

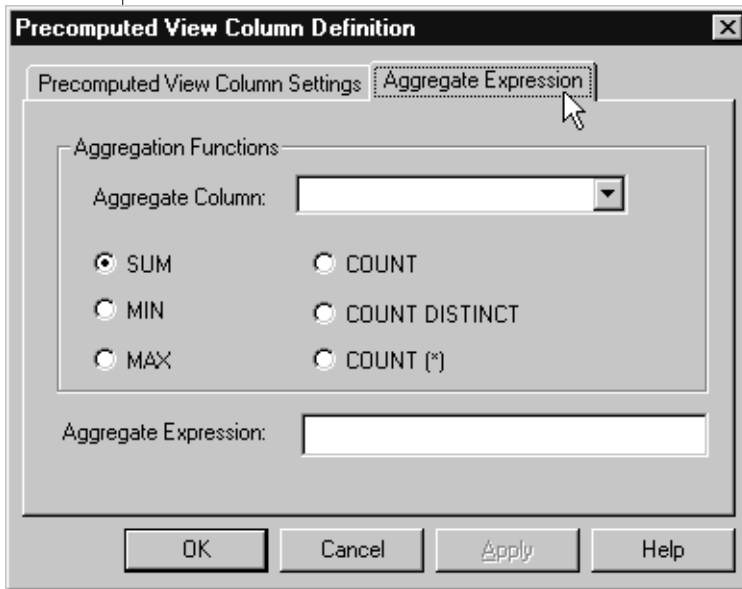


Figure 6-12
*Dialog Box for the
Expression*

10. Click the down arrow, select the name of the detail column to be aggregated (Dollars), and choose **SUM**.
The Administrator enters the aggregate expression in the lower box.

11. Click **OK** to generate the aggregate column.

The Administrator returns a description of the aggregate column in the upper pane, as [Figure 6-13](#) shows.

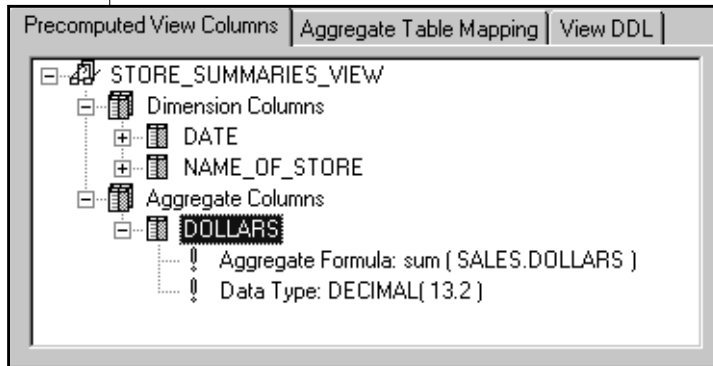


Figure 6-13
Description of an
Aggregate Column

Note that the Administrator promotes the data type of the Dollars column from Decimal (7,2) to Decimal (13,2) to accommodate the larger values of aggregate totals.

12. Continue this process until you have defined all the aggregate columns for the precomputed view.

You have now provided enough information for the Administrator to create the aggregate table. This table must be created before you can finish describing the precomputed view.

13. To create the aggregate table, right-click **Settings** and the Administrator returns a dialog box, as [Figure 6-14](#) shows.

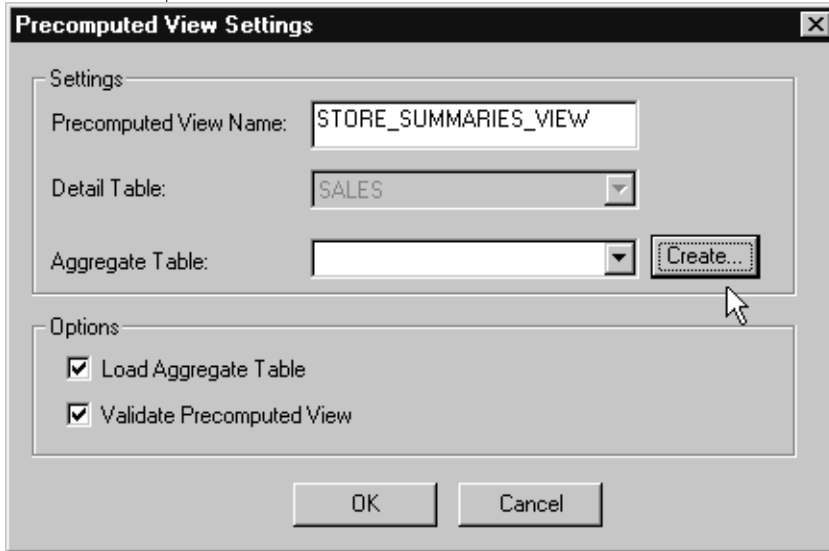


Figure 6-14
Create the
Aggregate Table

14. Left-click **Create** but *without entering the name of an aggregate table*.

The Administrator returns the Create Table wizard, a sequence of eight pages that prompts you for specific information about the table. You need to supply information for only two of the pages.

- ❑ **Enter Table Name:** Enter 'Store_Summaries'.
- ❑ **Define Primary Key:** Enter 'Store_Summaries_PK' and select Date and Store_Name as the primary key columns.

The last window of the sequence, **View SQL**, displays the DDL generated by the Administrator for the aggregate table. You can modify this script before you click **Finish**, but for this example that is not necessary.

```
create table STORE_SUMMARIES
( DATE DATE not null,
  STORE_NAME CHARACTER( 30 ) not null,
  QUANTITY DECIMAL( 16, 0 ) not null,
  DOLLARS DECIMAL( 13, 2 ) not null,
  primary key ( DATE, STORE_NAME ) );
```


15. Click **Finish**, and the Administrator creates the aggregate table.
After creating the aggregate table, the Administrator returns the **Precomputed View Settings** dialog box with the name of the Store_Summaries table in the Aggregate Table box.
16. Click **OK** and the Administrator diagrams the Store_Summaries table in the lower pane of the Source window, as [Figure 6-15](#) shows.

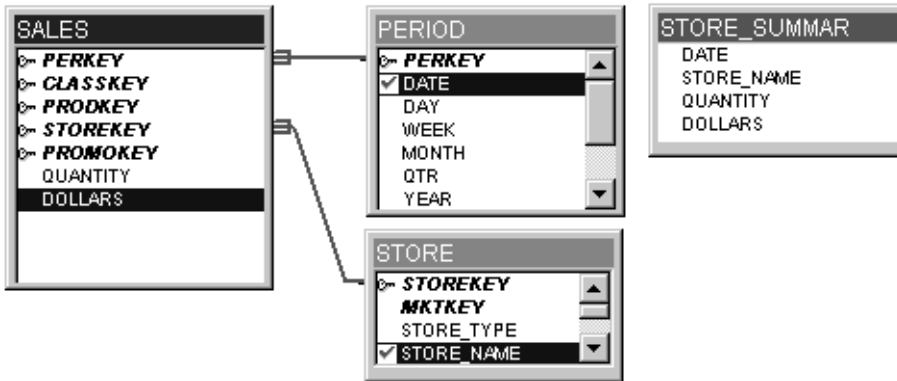


Figure 6-15
Diagram of the
Aggregate Table

The Administrator diagrams the precomputed view with a red highlight and the aggregate table with a green highlight. Now that the aggregate table exists, you are ready to map aggregate table columns to precomputed view columns.

17. Click **Aggregate Table Mapping** to start the mapping process.
The Administrator returns the column names of the view and the aggregate table in the upper pane, as [Figure 6-16](#) shows.

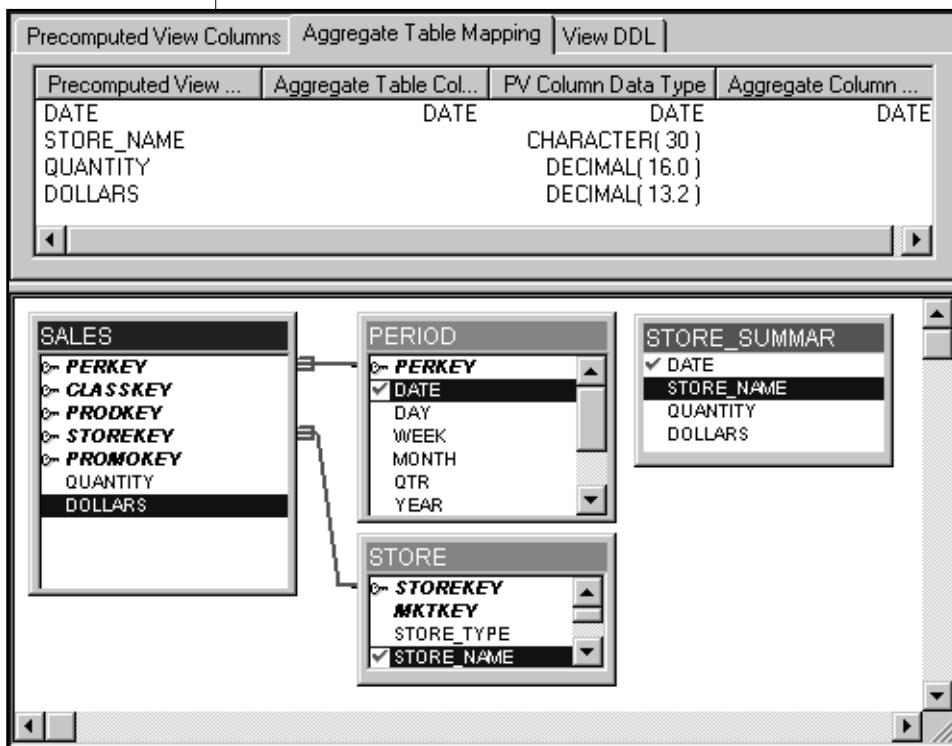


Figure 6-16
The Mapping
Process

18. To map an aggregate column to a view column, drag its name from the aggregate table in the lower pane to the appropriate column in the upper pane. For example, drag **Date** from the Store_Summaries table to **Date** in the upper pane. The Administrator then lists Date as mapped to the Aggregate table.

19. Perform the same action for the remaining three columns.

After all the aggregate columns have been mapped to precomputed view columns, the Administrator can create the precomputed view. Before you generate the precomputed view, you might want to look at the generated scripts. There are three scripts:

- **Create.** The DDL for the precomputed view.
- **Insert.** A statement that retrieves aggregate data from the Sales table and inserts it into the aggregate table.
- **SET.** A statement that marks the precomputed view valid.

Verify that each statement is complete before you create the view.

20. To view the generated scripts, right-click **VIEW DDL**, as [Figure 6-17](#) shows.

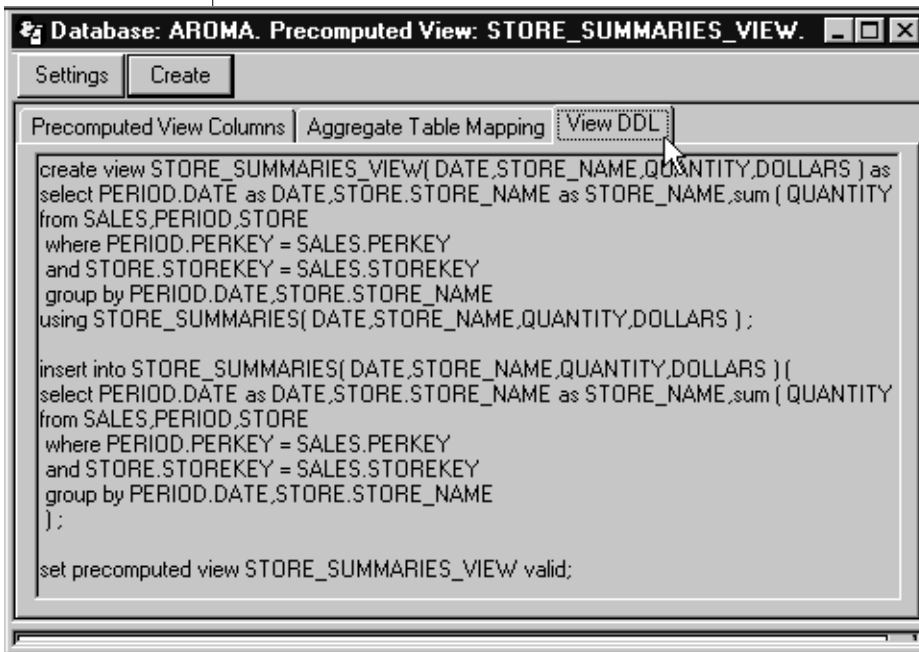


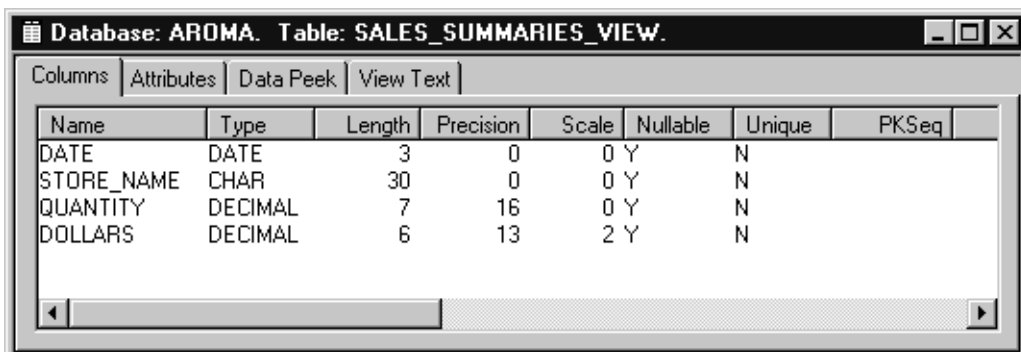
Figure 6-17
The Generated
Scripts

21. Click **Create** to execute these statements.

The status of these operations is returned in a popup pane.

22. To verify that the view is correct, return to the Source window, right-click **Sales_Summaries_View** and choose **Inspect**. The Administrator returns a secondary window that completely describes the view, as [Figure 6-18](#) shows.

Figure 6-18
Description of Precomputed View

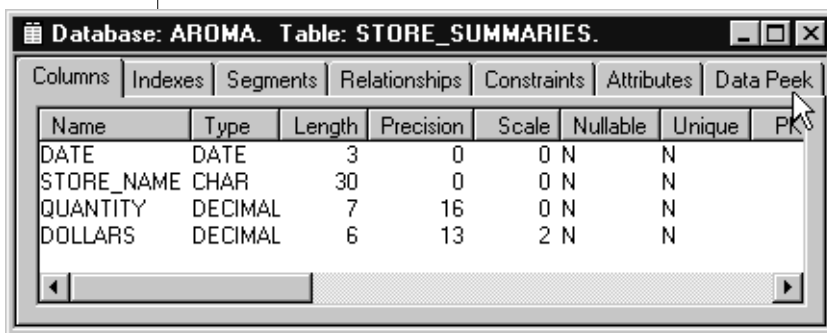


The screenshot shows a window titled 'Database: AROMA. Table: SALES_SUMMARIES_VIEW.' with tabs for 'Columns', 'Attributes', 'Data Peek', and 'View Text'. The 'Columns' tab is active, displaying a table with the following data:

Name	Type	Length	Precision	Scale	Nullable	Unique	PKSeq
DATE	DATE	3	0	0	Y	N	
STORE_NAME	CHAR	30	0	0	Y	N	
QUANTITY	DECIMAL	7	16	0	Y	N	
DOLLARS	DECIMAL	6	13	2	Y	N	

23. To verify that the aggregate table was loaded, return to the Source window, expand **Tables**, right-click **Sales_Summaries** and choose **Inspect**. The Administrator returns a secondary window that completely describes the table, as [Figure 6-19](#) shows.

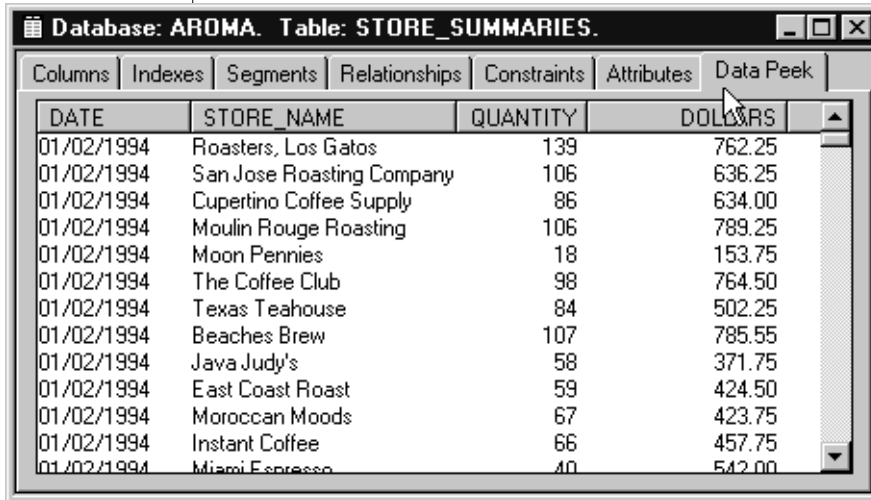
Figure 6-19
Description of Aggregate Table



The screenshot shows a window titled 'Database: AROMA. Table: STORE_SUMMARIES.' with tabs for 'Columns', 'Indexes', 'Segments', 'Relationships', 'Constraints', 'Attributes', and 'Data Peek'. The 'Columns' tab is active, displaying a table with the following data:

Name	Type	Length	Precision	Scale	Nullable	Unique	PKSeq
DATE	DATE	3	0	0	N	N	
STORE_NAME	CHAR	30	0	0	N	N	
QUANTITY	DECIMAL	7	16	0	N	N	
DOLLARS	DECIMAL	6	13	2	N	N	

24. Click **Data Peek** to verify that the aggregate table was created and loaded with the correct data. The Administrator returns a sample of the data, as [Figure 6-20](#) shows.



DATE	STORE_NAME	QUANTITY	DOLLARS
01/02/1994	Roasters, Los Gatos	139	762.25
01/02/1994	San Jose Roasting Company	106	636.25
01/02/1994	Cupertino Coffee Supply	86	634.00
01/02/1994	Moulin Rouge Roasting	106	789.25
01/02/1994	Moon Pennies	18	153.75
01/02/1994	The Coffee Club	98	764.50
01/02/1994	Texas Teahouse	84	502.25
01/02/1994	Beaches Brew	107	785.55
01/02/1994	Java Judy's	58	371.75
01/02/1994	East Coast Roast	59	424.50
01/02/1994	Moroccan Moods	67	423.75
01/02/1994	Instant Coffee	66	457.75
01/02/1994	Miami Espresso	40	542.00

Figure 6-20
*A Peek at the
Aggregate Data*

You could also peek at the data from the precomputed view but this would not verify that the aggregate table was loaded. In a case where the aggregate table was empty, the query defined by the view would retrieve the data from the Sales table.

Now that you have verified that the precomputed view and its aggregate table are correct, you can grant Select privilege on the view.

25. To grant Select privilege on the view to users, return to the Source window. Expand **Users** and choose **Grant** from the popup menu.

The Administrator returns the Grant User Wizard, as [Figure 6-21](#) shows. You can use this wizard to grant Select privilege on the new view to all or a few users.

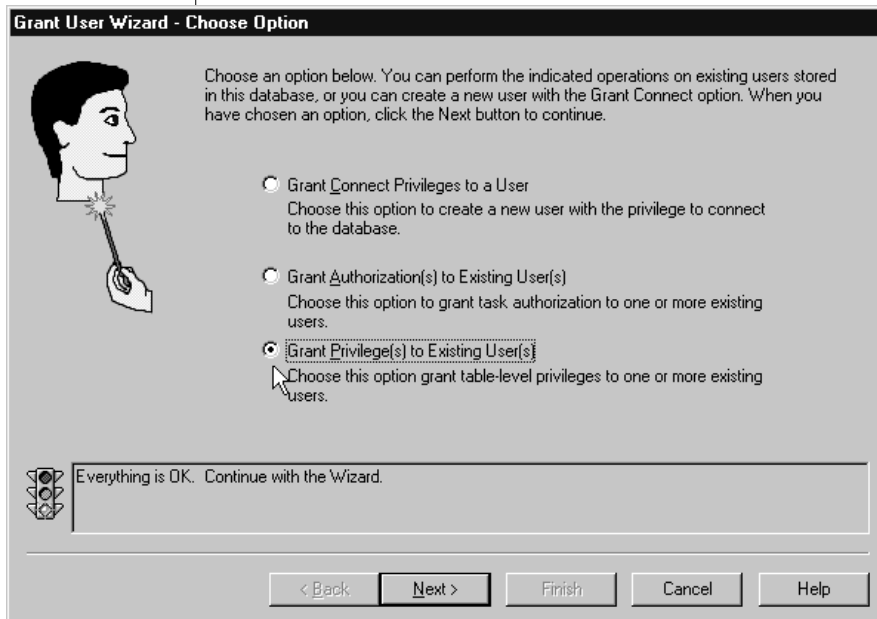


Figure 6-21
*Grant Select
Privilege to Users*

Creating and Verifying Hierarchies

You can use the Administrator to inspect, create, modify, and validate hierarchies. The following example shows how to create a hierarchy used by “Case 2. Making Use of Explicit Hierarchies” on [page 4-10](#).

The Store to District Hierarchy

Case Study 2 defined a Geography hierarchy on columns in the Store and Market tables.

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

You can readily create this hierarchy using the Administrator.

To Create the Geography Hierarchy

1. Right-click **Hierarchies** and select **Create** from the popup menu. The Administrator returns a Hierarchy Settings text box. Enter the name of the hierarchy in the text box and then click **OK**.
2. The Administrator returns a box that lists the names of all tables in the database. Select Store and Market and then click **Add**.

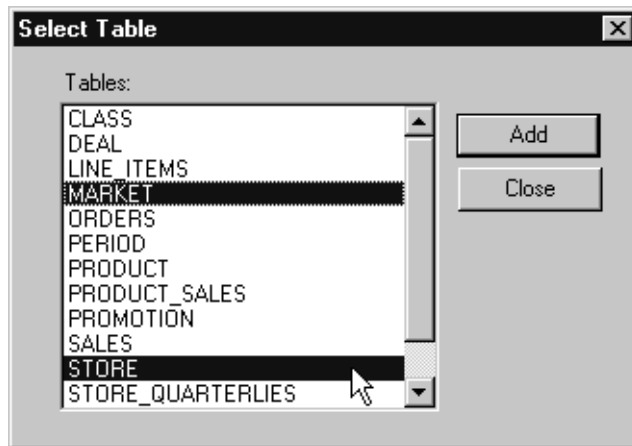


Figure 6-22
Select Table List Box

The Administrator returns a window for the Geography hierarchy and describes the selected tables in the lower pane.

3. To define a hierarchy, you drag column names from the lower pane and drop them over the hierarchy name in the upper pane.

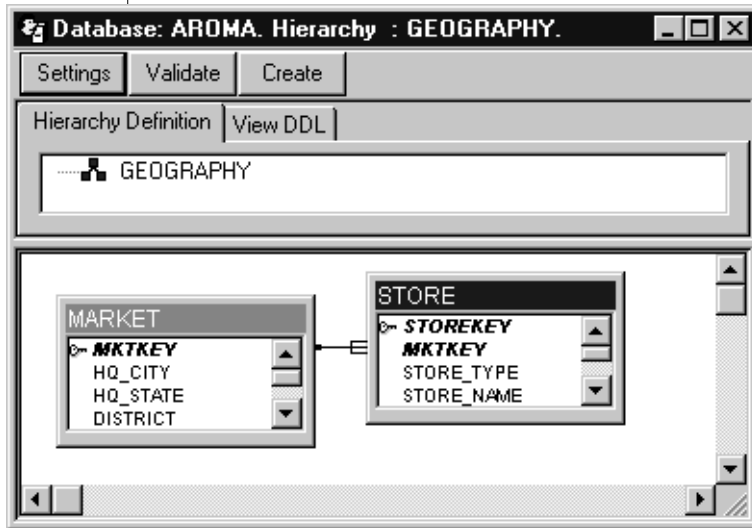


Figure 6-23
Geography
Hierarchy

- **Define TO column.** Drag **District** from the lower pane to **Geography** in the upper pane. The Administrator confirms this operation by placing Market(District) under Store_District.
- **Define FROM column.** Drag **City** from the lower pane to **Market(District)** in the upper pane. The Administrator confirms this operation by placing Store(City) under Market(District).
- **Define FROM column.** Drag **Store_Name** from the lower pane to **Store(City)** in the upper pane. The Administrator confirms this operation by placing the store name under the city name.

The Administrator confirms each drag operation in the upper pane.

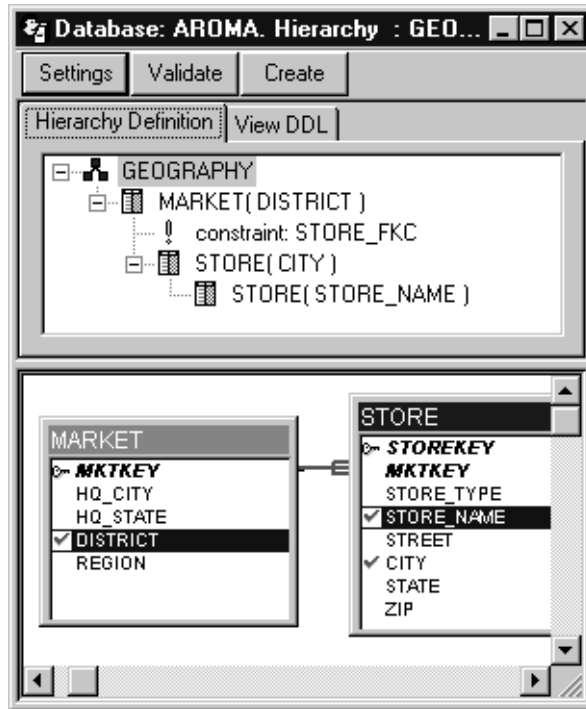


Figure 6-24
Graph of a Hierarchy

4. Click **View DDL** to confirm that the hierarchy has been correctly defined. The Administrator displays the generated DDL.

```
create hierarchy GEOGRAPHY(
  from STORE( CITY ) to MARKET( DISTRICT )
  on STORE_FKC,
  from STORE( STORE_NAME ) to STORE( CITY ) )
```

5. Click **Validate** to verify that the functional dependencies defined by the Store_District hierarchy also obtain in the physical data.

If the hierarchy obtains in the physical data, Vista returns the confirmation in a message box; otherwise, Vista returns specific information that describes why the hierarchy is invalid.



Warning: The validation operation can take a long time if the hierarchy is complex and the dimensions large.

6. Click **Create** to build the hierarchy. The Administrator responds with a popup menu that asks whether to validate the hierarchy. You have already done that, so click **No**.

The Administrator diagrams the hierarchy in the lower pane on the Market and Store table using red lines, as [page 6-26](#) shows.

The script defines the hierarchy from a fine to a coarser granularity while the Administrator diagrams it from coarser to finer.

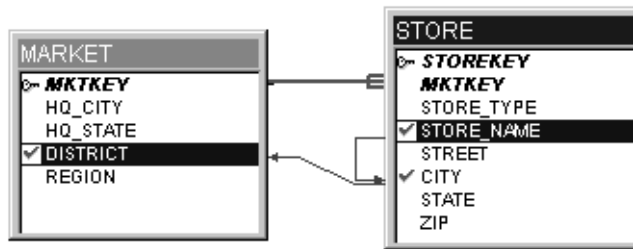


Figure 6-25
Hierarchy Diagram

Using the Advisor

You can use the Administrator tool to analyze log data that resides in the Advisor log file. Specifically, you can use the Administrator to:

- Create an Advisor log, and start and stop Advisor logging.
- Analyze precomputed view utilization based on the Advisor log file and display the results in a tabular report or one or more graphs.
- Generate candidates for precomputed views based on a historical query profile and the actual detail data (or a representative sample of the detail data).

This section shows you how to analyze precomputed views using the Administrator; for information on how to create an Advisor log and turn logging on and off, see [“Managing Vista” on page 6-6](#).

Precomputed View Utilization

When the DBA turns on Advisor logging, the Advisor captures and stores statistics on the utilization of precomputed views in its log file.

To Analyze Precomputed View Utilization

1. Left-click **Tools** on the Administrator toolbar and then select **Vista Advisor** from the drop-down list.

The Administrator returns the Vista Advisor Wizard. The initial page provides two choices: analyze utilization or analyze candidates.

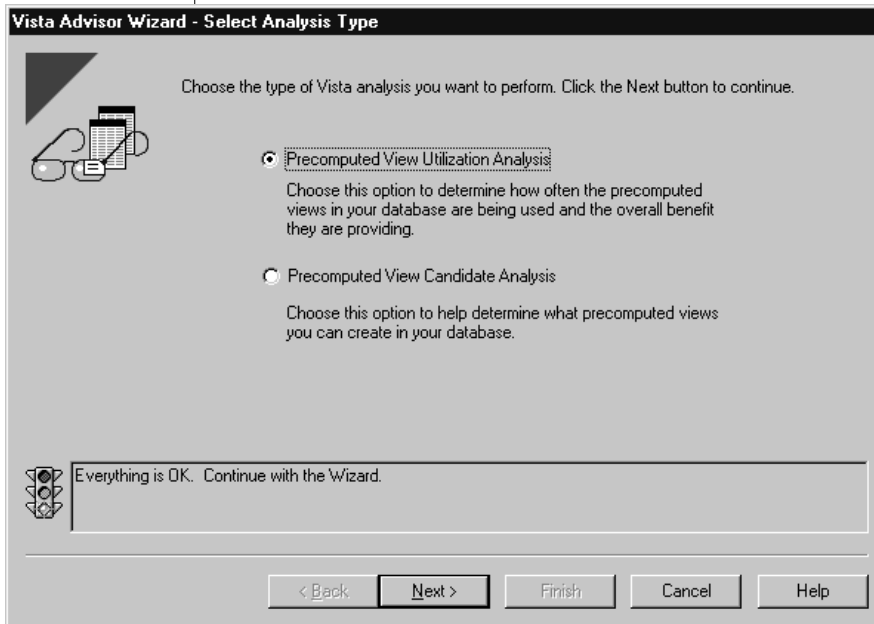


Figure 6-26
Vista Advisor
Wizard

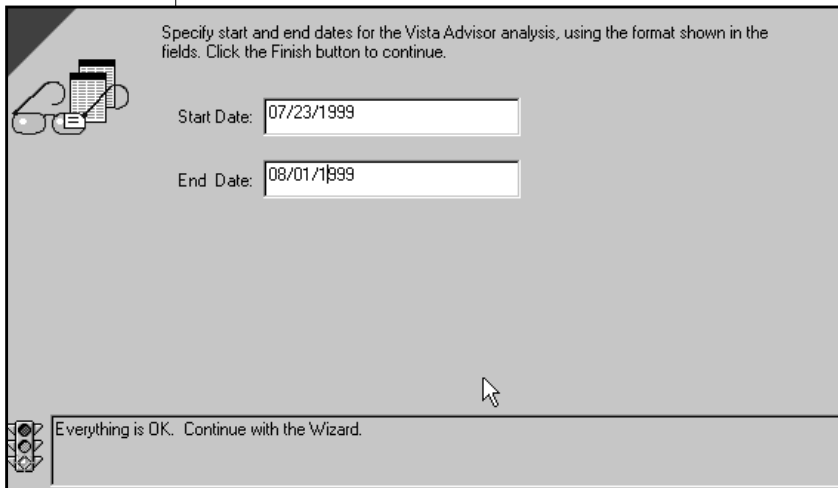
2. Check the **Precomputed View Utilization Analysis** button and then click **Next**.

The next window provides two choices: run a new analysis or open an existing analysis. Currently, an analysis does not exist. After you complete a new analysis, you can save it for later analysis.

3. Check the **Run New Analysis** button and then click **Next**.

The next window lists the detail tables that are referenced by precomputed views. You can limit the Advisor analysis to one detail table or expand it to include several.

4. Double-click **Sales** from the **Detail Tables** box and then click **Next**.
The next page contains two boxes: one for a start date and the other for an end date. These dates define the time period for the analysis.
5. Enter a start date in the upper box and an end date in the lower box and then click **Finish**.



The screenshot shows a window titled 'Specify start and end dates for the Vista Advisor analysis, using the format shown in the fields. Click the Finish button to continue.' The window has a light gray background. In the top-left corner, there is an icon of a magnifying glass over a document. Below the title bar, there are two text input fields. The first field is labeled 'Start Date:' and contains the text '07/23/1999'. The second field is labeled 'End Date:' and contains the text '08/01/1999'. At the bottom of the window, there is a status bar with a traffic light icon on the left and the text 'Everything is OK. Continue with the Wizard.' on the right. A mouse cursor is visible over the status bar.

Figure 6-27
Specify Time Period

Note that a start and end date defined on the same day define an empty interval. See [“Inserting the Results of an Advisor Query into a Table” on page 5-10.](#)

You have now specified the following information:

- Utilization analysis
- New analysis (contrasted with the review of a previous one)
- Analysis based on the Sales detail table
- Analysis period

The Administrator starts the analysis and returns a status box.

6. Analyze the results of the Utilization Analysis report returned by the Administrator, as [Figure 6-28](#) shows.

Detail Table	Aggregate Table	Valid	Size	Reduction Factor	Ber
SALES	STORE_SUMMARIES	Y	13871	5	
SALES	STORE_QUARTERLIES	Y	156	448	

Figure 6-28
Utilization Analysis

From this window you can display a detailed description of the view or graphs of selected columns.

7. Double-left-click a precomputed view name in the window to display a detailed description of the view, as [Figure 6-29](#) shows.

Name	Type	Length	Precision	Scale	Nullable	Unique
DATE	DATE	3	0	0	Y	N
STORE_NAME	CHAR	30	0	0	Y	N
QUANTITY	DECIMAL	7	16	0	Y	N
DOLLARS	DECIMAL	6	13	2	Y	N

Figure 6-29
Precomputed View Description

You can dig deeper with a few clicks:

- Click **Data Peek** for a sample of data from the view.
- Click **Attributes** for the view attributes.
- Click **View Text** for the DDL used to create the view.

You can also use the Administrator to generate graphs that provide a different look into the analysis.

8. Click the down-arrow for **Graph** on the tool bar of the Utilization Analysis window (see [Figure 6-28](#)). Select **Reference Count** from the list box.

The Administrator returns a pie chart similar to the one in [Figure 6-30](#) (The Administrator version is in color and has a much nicer format).

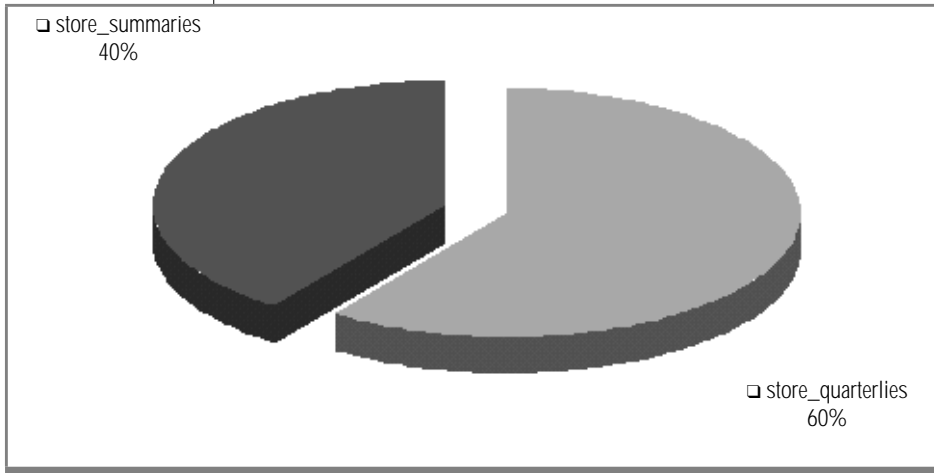


Figure 6-30
*Pie Chart of
Reference Counts*

The Administrator can graph the following information:

- Reference Count pie chart
- Size pie chart
- Benefit pie chart
- Reduction Factor pie chart
- Non-Exact Match Count pie chart
- Size versus Benefit graph

All the charts are in color and fully annotated.

9. Click **Save** on the tool bar to save the utilization analysis. The Administrator returns a standard Windows dialog box.

Locate a directory where you want to save the analysis, enter a filename in the text box, and then click **Save**. At a later time you can review this analysis using the Administrator.

Candidate View Generation

The Advisor can help the DBA improve query performance by suggesting candidates for new precomputed views. These candidates are based on actual patterns of query usage during a specified time interval. You can run this kind of analysis easily using the Administrator, and the procedure is remarkably similar to the previous one.

To Generate Precomputed View Candidates

1. Left-click **Tool** on the Administrator tool bar and then select **Vista Advisor** from the drop-down list.

The Administrator returns the initial page of the Vista Advisor Wizard. The initial page provides two choices: precomputed view utilization or candidate analysis.

2. Check the **Precomputed View Candidate Analysis** button and then click **Next**.

The next page provides two choices: run a new analysis or open an existing one. After you complete a new analysis, the Administrator can save it for further analysis at a later time.

3. Check the **Run New Analysis** button and then click **Next**.

The next page lists detail tables that are referenced by precomputed views. You can limit the Advisor analysis to a single detail table or include several. You can also base the analysis on a sample of data. See [“Generating Candidates from Sampled Data” on page 6-35](#).

4. Double-click **Sales** from the **Detail Tables** box and then click **Next**.

The next page contains two boxes: one for a start date and the other for an end date. These dates define the time period for the analysis.

5. Enter a start date in the upper box and an end date in the lower box and then click **Finish**.

You have now specified the following information:

- Candidate analysis
- New analysis (contrasted with the review of a previous one)
- Analysis based on the Sales detail table
- Analysis period

The Administrator starts the analysis and returns a status box that says the analysis is under way. This could take a long time.

6. Study the analysis summarized in the Precomputed View Candidate Analysis window, as [Figure 6-31](#) shows.

Figure 6-31
Candidate Analysis

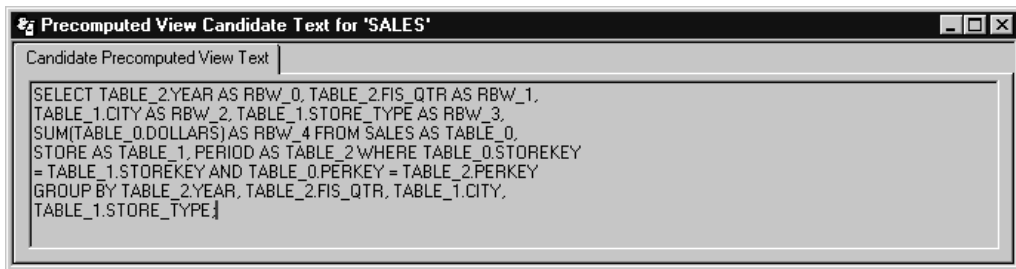
Precomputed View	Detail Table	Aggregate Table	Agg ...	Reference ...	Size	Reduction...	Benefit
STORE_SUMMARIES_VIEW	SALES	STORE_SUMMARIES	0	0	13871	5	0
CANDIDATE1	SALES		5	42	13871	5	0
STORE_QUARTERLIES_VIEW	SALES	STORE_QUARTERLIES	0	0	156	448	0
CANDIDATE2	SALES		167	94	468	149	1876420
CANDIDATE3	SALES		92	46	156	448	14352
CANDIDATE4	SALES		45	22	174	401	301334
CANDIDATE5	SALES		7	37	125	559	508602
CANDIDATE6	SALES		0	0	13871	5	13512870

This window lists the name of a precomputed view defined on a detail table followed by a list of new views (candidates) that could improve query performance. Each candidate also has a set of statistics to help you evaluate its usefulness.

From this window, you can display a detailed description of a precomputed view, display the DDL for a candidate view, or graph the statistics in the tabular report.

7. Double-click the name of an existing precomputed view to display a detailed description of the view and take a peek at its data.
8. Click the down-arrow for **Graph** on the tool bar of the Candidate Analysis window and select a graph type from the drop-down list box. This is the same procedure used to graph the utilization data. For more details, see [Figure 6-28 on page 6-29](#).
9. Double-left-click **CANDIDATE4** to display the DDL for this candidate.

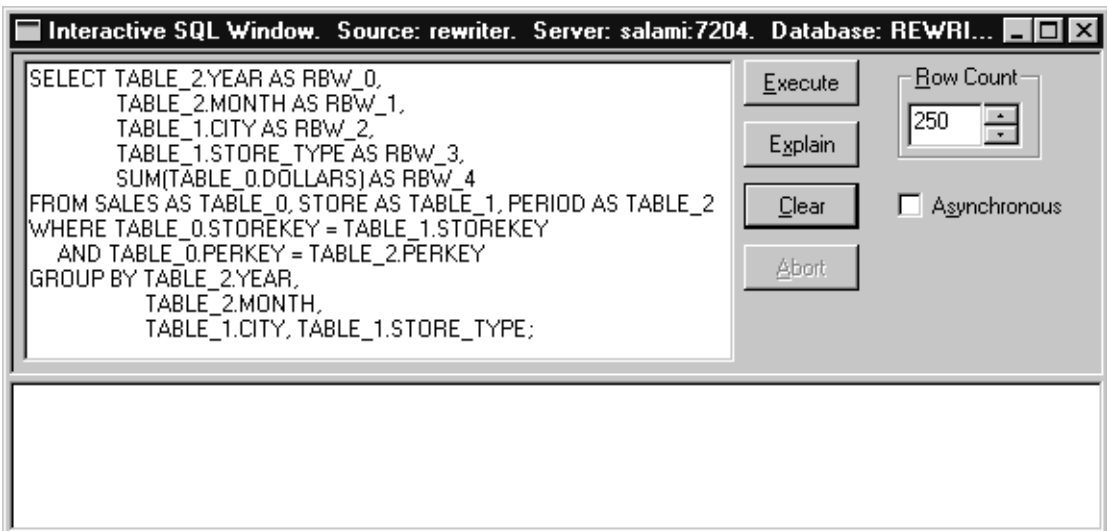
Figure 6-32
Candidate View Text



10. You can create a precomputed view using this text in a few steps.
 - a. Create an aggregate table.
 - b. Insert rows into the aggregate table. (You can use the query in [Figure 6-32](#) as the SELECT clause of an INSERT statement.)
 - c. Create a precomputed view for the aggregate table. (You can use the query in [Figure 6-32](#) for the AS clause of the view.)
 - d. Mark the view valid.

To complete these steps using the Administrator, left-click the **ISQL** icon on the Administrator toolbar. The Administrator returns its Interactive SQL Window, as [Figure 6-33](#) shows.

Figure 6-33
Interactive SQL Window



From this window, you can write and execute scripts that create the aggregate table, insert data into the aggregate table, and create the precomputed view.

11. Click **Save** on the tool bar to save the candidates analysis. The Administrator returns a standard Windows dialog box.

Locate a directory where you want to save the analysis, enter a file name in the text box, and then click **Save**. At a later time you can review this analysis using the Administrator.

Generating Candidates from Sampled Data

You can direct the Advisor to generate candidates for new views based on:

- All the data in a detail table.
- A sample of the data in a detail table. The sample is defined by a view defined on the detail data.

A sample of data is used in lieu of all the data so that a more *representative* data set can be analyzed. This restriction can also speed up the analysis.

You can use the Administrator to perform a candidate analysis on a sample of data provided you have created a view to define the sample. You can define the view using the Administrator.

The following example *assumes* that the third week of July represents the desired workload characteristics. The example illustrates how to use the Administrator to create a view that defines the sample data set and then use it to perform an Advisor candidate analysis.

To Generate Candidates from a Sample

1. Left-click the **Manage** icon on the Administrator tool bar and then select Views from the drop-down list.
The Administrator returns the Manage Views wizard. This wizard provides three options: create a view, alter a view, or drop a view.
2. Check **Create A New View** and click **Next**.
The next page provides a text box for the name of the view.
3. Enter the name of the view (**SAMPLE_WEEK**) in the text box and then click **Next**.
The next page provides a text box for a query that defines the view.

4. Enter the query that defines the view in the SQL Command text box and then click **Next**.

The following query defines the third week of July as the sample.

```
select PERKEY, CLASSKEY, PRODKEY, STOREKEY, PROMOKEY,
       QUANTITY , DOLLARS
from   sales natural join period
where  year = 1999
and    week = 29;
```

You can create this query from within the Administrator.

The next window (**Enter Result Set Column Names**) requests a description of any additional columns not defined by the query.

5. Click **Next** because there are no additional columns.

The next window provides a text box to enter a comment on the view. This comment is included with the view text in the system tables.

6. Enter the following statement in the Comment text box before you click **Next**.

This view characterizes a representative sample of data for the Vista Advisor to analyze: The third week of July 1999.

The next page contains the statement that generates the view.

7. Inspect the statement and if correct click **Finish**.

The view for the sample now exists and you can base an Advisor on this sample instead of on all the data contained in the Sales table.

8. Click **Tools** from the Administrator tool bar and select Vista Advisor.

The next page has two choices: utilization analysis or candidate analysis.

9. Choose **Precomputed View Candidate Analysis** and click **Next**.

The next page has two choices: new or existing analysis.

10. Choose **Run New Analysis** and click **Next**.


The next page requests a list of table or view names that define the detail data to be used for the analysis.

11. Double-click the view name that contains the July 1999 sample and then click **Next**.

Figure 6-34
Detail and Sample Tables

Vista Advisor Wizard - Select Objects for Analysis

Select the objects for the analysis: tables for Utilization Analysis; tables and/or sample views for Candidate Analysis. By default only detail tables are shown; check the Show All Tables box to list all the tables in the database. Click Next to continue.




Detail Tables

PERIOD PRODUCT PROMOTION STORE STORE_QUARTERLIES STORE_SUMMARIES SUPPLIER	-> <-	SALES
---	----------	-------

☒ Show All Tables

Sample Views

PEAK_USAGE_SAMPLE TYPICAL_MONTH	-> <-	SAMPLE_WEEK
------------------------------------	----------	-------------

 OK. When you are finished, click the Next button to proceed to the next page.

The remainder of the analysis proceeds as before with one exception: the Advisor uses the sample of data defined by the view instead of using all the data contained in the Sales table.

Glossary

Advisor	The logging and analysis component of Informix Vista. 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. <i>See</i> 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 Informix 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	<i>See</i> 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	<i>See</i> 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 database server from an operational system. For example, the detail table used to analyze retail sales might derive from a point-of-sale (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	<i>See rollup.</i>
functional dependency	A many-to-one relationship shared by columns of values in database tables. A functional dependency from column <i>X</i> to column <i>Y</i> is a constraint that requires two rows to have the same value for the <i>Y</i> column if they have the same value for the <i>X</i> column. <i>See also</i> hierarchy.
generated SQL	SQL as rewritten internally by the Informix 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 database administrator, using the CREATE HIERARCHY command.
implicit rollup	<i>See rollup.</i>
many-to-one relationship	<i>See functional dependency.</i>
materialized view	<i>See precomputed view.</i>
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 <i>non-exact match count</i> . <i>See also</i> exact match.

precomputed table	A table associated with a precomputed view in a CREATE VIEW...USING statement. In the context of the Informix Vista option, precomputed tables are always <i>aggregate tables</i> .
precomputed view	A view linked to a database table known as a <i>precomputed table</i> . 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 <i>prestored summary</i> 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 5-9
 ADMIN_ADVISOR_LOGGING
 parameter
 creating Advisor log files 5-5
 Advisor
 candidate views 3-14
 introduction to 1-7
 overview 5-4
 Advisor logging
 candidate views 5-6
 configuring 5-5
 correlated subqueries 5-6
 log files 5-5
 queries that are not logged 5-7
 aggregate queries, defined 2-6
 aggregate tables
 creating 3-4
 defined 2-6
 example 3-5
 existing 3-6
 family of 3-22
 loading 3-4
 loading with cascaded
 inserts 3-12
 visibility to client tools 3-30
 aggregation columns 3-8
 Aroma database Intro-4, 1-8, 3-5
 averages, rewriting queries that
 calculate 4-31
 AVG function 3-9

B

BENEFIT column
 described 5-21
 how it is calculated 5-24
 understanding 5-23 to 5-28
 what numbers mean 5-25

C

candidate views 3-14
 cascaded inserts 3-12
 cases, tracked by technical
 support Intro-12
 comment icons Intro-10
 compound expressions 3-12
 constraint names, in hierarchy
 definitions 3-16
 contact information Intro-16
 conventions
 syntax diagrams Intro-7
 syntax notation Intro-6
 correlated subqueries
 logging 5-6
 rewriting 4-30
 cost-based analysis, of
 precomputed views 3-11
 COUNT function 3-8
 CREATE HIERARCHY
 command 2-11, 3-14
 CREATE TABLE statements
 for aggregate tables 3-6
 for derived dimensions 3-23
 CREATE VIEW...USING
 command 3-7

D

data skew 5-7
demonstration database
 script to install Intro-4
dependencies, software Intro-4
derived dimensions 2-13
 as referenced tables 4-23
 creating 3-20
 examples of rewritten
 queries 4-22
 precomputed views linked
 to 3-24
dimension tables, derived 3-20
DISTINCT function 3-9, 3-24
documentation
 list of Red Brick Decision
 Server Intro-13
documentation, types of
 on-line manuals Intro-15
 printed manuals Intro-16
DROP HIERARCHY
 command 3-16

E

exact match, defined 5-14
EXPLAIN command 3-28
explicit hierarchies 3-14
 defined 2-11
 example 2-12
 examples of rewritten
 queries 4-10

F

family of aggregate tables 3-22
features of this product,
 new Intro-5
fiscal periods 4-18
foreign-key constraint names, in
 hierarchy definitions 3-16
FROM clause subqueries 4-30
functional dependencies 3-13, 4-10
 defined 2-10
 example 2-12
 validity of 3-14

G

generated SQL 3-20
 simplified with derived
 dimensions 4-26
GROUP BY clause, compound
 expressions in 3-12
grouping columns 3-7

H

HAVING clause 3-9
hierarchies
 concept of 2-10
 explicit 3-14, 4-10
 implicit 3-18, 4-27
 use with Advisor 5-10
 validity of 3-14
 verifying validity of 3-17

I

icons
 important Intro-10
 tip Intro-10
 warning Intro-10
implicit hierarchies 3-18
 defined 2-13
 examples of rewritten
 queries 4-27
important paragraphs, icon
 for Intro-10
indexes
 for rewritten queries 3-11
 on aggregate tables 4-25
Informix Customer
 Support Intro-10
INSERT command 3-6
INSERT statements
 for derived dimensions 3-23
 rewriting 3-12
invalid precomputed views 3-26
 effect on Advisor 5-6

J

join predicates, for precomputed
 views 3-9

K

keywords
 in syntax diagrams Intro-9

L

logging, Advisor, *See* Advisor
 logging

M

materialized views, *See*
 precomputed views 3-7
MAX function 3-8
metavariables
 in syntax diagrams Intro-9
MIN function 3-8

N

new features of this product Intro-5
NON_EXACT_MATCH_COUNT
 column 5-14
notation conventions Intro-5

O

ODBC Driver 3-30
on-line manuals Intro-15
OPTION ADVISOR_LOGGING
 parameter
 enabling Advisor logging 5-6
outboard tables, rewriting queries
 against 2-13

P

precomputed query
 expressions 3-7
 precomputed tables, *See* aggregate tables
 precomputed views
 cost-based analysis 3-11
 creating 3-7
 defined 2-4
 example 3-10
 join predicates 3-9
 linked to derived
 dimensions 3-24
 SET commands 3-26
 unknown columns 4-10
 validity of 3-26
 visibility to client tools 3-30
 primary key/foreign key
 relationships, functional
 dependencies in 3-13
 printed manuals Intro-16

Q

query blocks, rewritten
 separately 3-11
 query expressions 3-7
 query rewrite system
 calculating averages 4-31
 derived dimensions 4-22
 detailed examples 4-3
 exact-match rewrites 4-5
 explicit hierarchies 4-10
 implicit hierarchies 4-27
 introduction to 1-5
 key concepts 2-3
 negative tests 4-10
 optimizing rewritten queries 3-19
 queries not rewritten 3-12
 query blocks 3-11
 rewriting subqueries 4-30
 summary of use 3-31, 5-32
 turning on and off 3-28
 using 3-3

R

RBW_PRECOMPVIEW_
 CANDIDATES table
 described 5-29
 rules for querying 5-15
 RBW_PRECOMPVIEW_
 UTILIZATION table
 described 5-30
 querying 5-13
 RBW_TABLES system table 3-28,
 3-30
 RBW_TABLES_VIEW,
 creating 3-30
 RBW_VIEWS system table 3-28
 REDUCTION_FACTOR
 column 5-21
 referenced tables, derived
 dimensions as 4-23
 REFERENCE_COUNT
 column 5-22
 rewritten queries, *See* query rewrite system
 rewritten SQL 3-20
 RISQL display functions, in
 rewritten queries 4-9
 rollups
 defined 2-10
 hierarchies and 3-13

S

SET commands, for precomputed
 views 3-26
 set functions 3-8
 SET STATS INFO command 3-28,
 4-4
 shrunk dimensions, *See* derived dimensions
 SIZE column 5-21
 software dependencies Intro-4
 SQLTables function 3-30
 STARindexes, for aggregate
 tables 3-21, 4-25, 4-29
 statistics messages, for queries 4-4

subqueries, rewriting 4-30
 SUM function 3-8
 support
 technical Intro-10
 syntax diagrams
 conventions for Intro-7
 keywords in Intro-9
 metavariables in Intro-9
 syntax notation Intro-5
 system requirements
 database Intro-4
 software Intro-4
 system tables, querying view-
 related 3-28

T

technical support Intro-10
 temporary tables, use with
 Advisor 5-10
 tip icons Intro-10
 troubleshooting
 general problems Intro-12

U

UNIFORM PROBABILITY FOR
 ADVISOR command 5-26
 UNION queries
 separate query blocks for 3-11
 users, types of Intro-3
 USING clause, in CREATE VIEW
 command 3-7

V

validity
 of hierarchies 3-14
 of precomputed views 3-26
 precomputed views 2-5
 versioned database
 advantages of a frozen
 version 2-5
 validity of precomputed
 views 2-5
 Vista 1-5

view of RBW_TABLES 3-30
 view validity
 auto-invalidation by Vista 2-5
 views, *See* precomputed views

W

warning icons Intro-10
 WHEN clause
 disallowed in view
 definitions 3-9
 in rewritten queries 4-9
 WHERE clause predicates, in
 rewritten queries 4-9