SQL Self-Study Guide

Informix Red Brick Decision Server

Version 6.1
October 2000
Part No. 000-8059

Published by Informix® Press

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

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

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

Documentation Team: Twila Booth, Karen Byers, Diana Chase, Kathy Eckardt, Laura Kremers, Stephanie Krieger, Bob Rumsby

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

Chapter 1 Aroma, a Database for Decision Support

Chapter 2 Basic Queries

Chapter 3 Data Analysis

Chapter 4 Comparison Queries

Chapter 5 Joins and Unions

Chapter 6 Macros, Views, and Temporary Tables

Introduction

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 provides an example-based review of SQL and introduction to the RISQL extensions, the macro facility, and **Aroma**, the sample database.

Types of Users

This guide is written for the following users:

- Database users
- Database administrators
- Database server administrators
- Database-application programmers
- Database architects
- Database designers
- Database developers
- Backup operators
- Performance engineers

This guide assumes that you have the following background:

- A working knowledge of your computer, your operating system, and the utilities that your operating system provides
- Some experience working with relational databases or exposure to database concepts
- Some experience with 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 6.1, as your database server.

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

For information about how to create and populate the demonstration database, see the *Administrator's Guide*. For a description of the database and its contents, see Chapter 1, "Aroma, a Database for Decision Support" and Appendix A, "The Complete Aroma Database" of this 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.

Documentation Conventions

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

The following conventions are discussed:

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

Typographical Conventions

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

Convention	Meaning
KEYWORD	All primary elements in a programming language statement (keywords) appear in uppercase letters in a serif font.
italics *italics* *italics*	Within text, new terms and emphasized words appear in italics. Within syntax and code examples, variable values that you are to specify appear in italics.
boldface ***boldface***	Names of program entities (such as classes, events, and tables), environment variables, file and pathnames, and interface elements (such as icons, menu items, and buttons) appear in boldface.
`monospace` *monospace*	Information that the product displays and information that you enter appear in a monospace typeface.

(1 of 2)

Convention	Meaning
KEYSTROKE	Keys that you are to press appear in uppercase letters in a sans serif font.
♦	This symbol indicates the end of one or more product- or platform-specific paragraphs.
→	This symbol indicates a menu item. For example, "Choose **Tools→Options**" means choose the **Options** item from the **Tools** menu.

(2 of 2)

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

Syntax Notation

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

Command Element	Example	Convention
Values and parameters	*table_name*	Items that you replace with an appropriate name, value, or expression are in *italic* type style.
Optional items	[ ]	Optional items are enclosed by square brackets. Do not type the brackets.
Choices	ONE | TWO	Choices are separated by vertical lines; choose one if desired.
Required choices	{ONE|TWO}	Required choices are enclosed in braces; choose one. Do not type the braces.

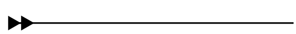
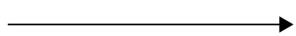
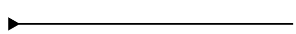
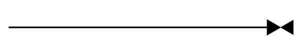
(1 of 2)

Command Element	Example	Convention
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.

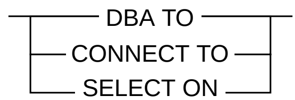
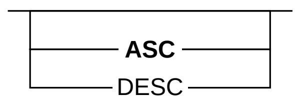
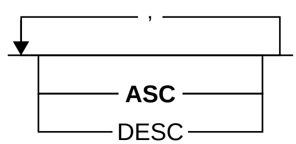
(2 of 2)

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.

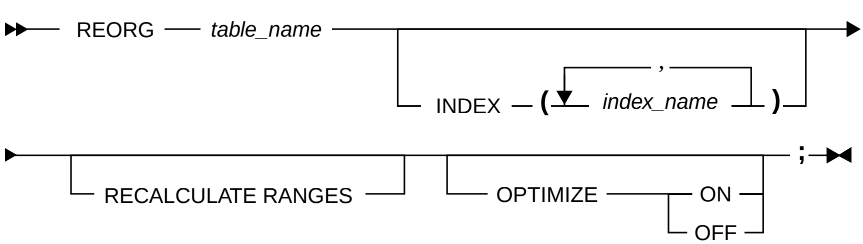
Component	Meaning
	Statement begins.
	Statement syntax continues on next line. Syntax elements other than complete statements end with this symbol.
	Statement continues from previous line. Syntax elements other than complete statements begin with this symbol.
	Statement ends.
	Required item in statement.
	Optional item.

(1 of 2)

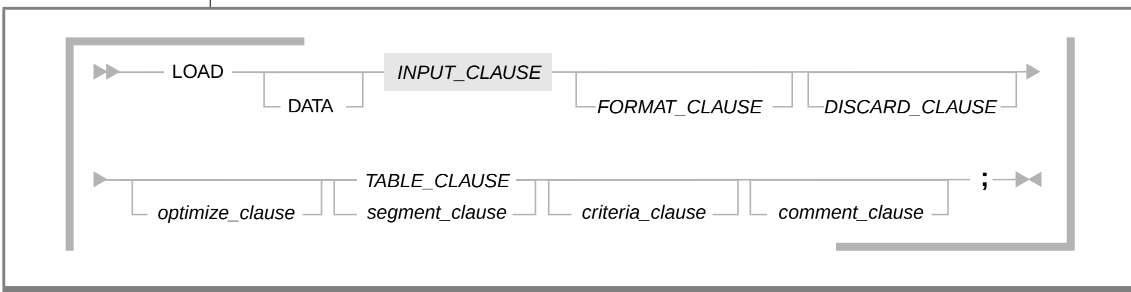
Component	Meaning
	Required item with choice. One and only one item must be present.
	Optional item with choice. If a default value exists, it is printed in **bold**.
	Optional items. Several items are allowed; a comma must precede each repetition.

(2 of 2)

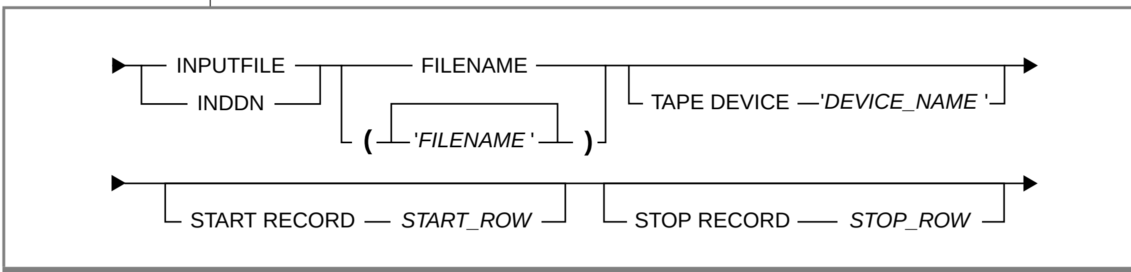
The preceding syntax elements are combined to form a diagram as follows.



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*.

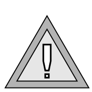
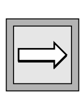
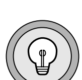
The following syntax diagram uses variables to illustrate the general form of a simple SELECT statement.

```
▶▶── 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.

Comment Icon Conventions

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

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

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. Central Time, Monday through Friday)

Internet access http://www.informix.com/techinfo

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

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

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

New Cases

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

- Red Brick Decision Server version
- 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 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 (TMU) 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 documentation set for Red Brick Decision Server includes the following documents.

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

(1 of 2)

Document	Description
The release notes	Contains information pertinent to the current release that was unavailable when the documents were printed.
RISQL Entry Tool and RISQL Reporter User's Guide	Is a complete guide to the RISQL Entry Tool, a command-line tool used to enter SQL statements, and the RISQL Reporter, an enhanced version of the RISQL Entry Tool with report-formatting capabilities.
SQL Reference Guide	Is a complete language reference for the Informix Red Brick SQL implementation and RISQL extensions for Red Brick Decision Server databases.
This guide	Provides an example-based review of SQL and introduction to the RISQL extensions, the macro facility, and **Aroma**, the sample database.
Table Management Utility Reference Guide	Describes the Table Management Utility, including all activities related to loading and maintaining data. Also includes information about data replication and the **rb_cm** copy management utility.

(2 of 2)

Additional references you might find helpful include:

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

Additional Documentation

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

- Online manuals
- Printed manuals

Online 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

Send electronic mail to us at the following address:

doc@informix.com

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

We appreciate your suggestions.

Chapter

1

Aroma, a Database for Decision Support

In This Chapter

This guide shows how to express commonly asked business questions as database queries by using the Structured Query Language (SQL) and the RISQL extensions to SQL developed by Informix. This guide also illustrates how query writing can be simplified with RISQL macros when queries or parts of queries are issued repetitively.

All the examples in this document and in most of the Informix documentation are based on **Aroma**, a sample database that contains sales data for coffee and tea products sold in stores across the United States. Each example consists of three parts:

- A business question, expressed in everyday language
- One or more corresponding SELECT statements, expressed in SQL
- A table of results returned from the database

Aroma is typically installed when the Red Brick Decision Server software is installed. To run the sample queries yourself, ask your system administrator how to access the **Aroma** database at your site.

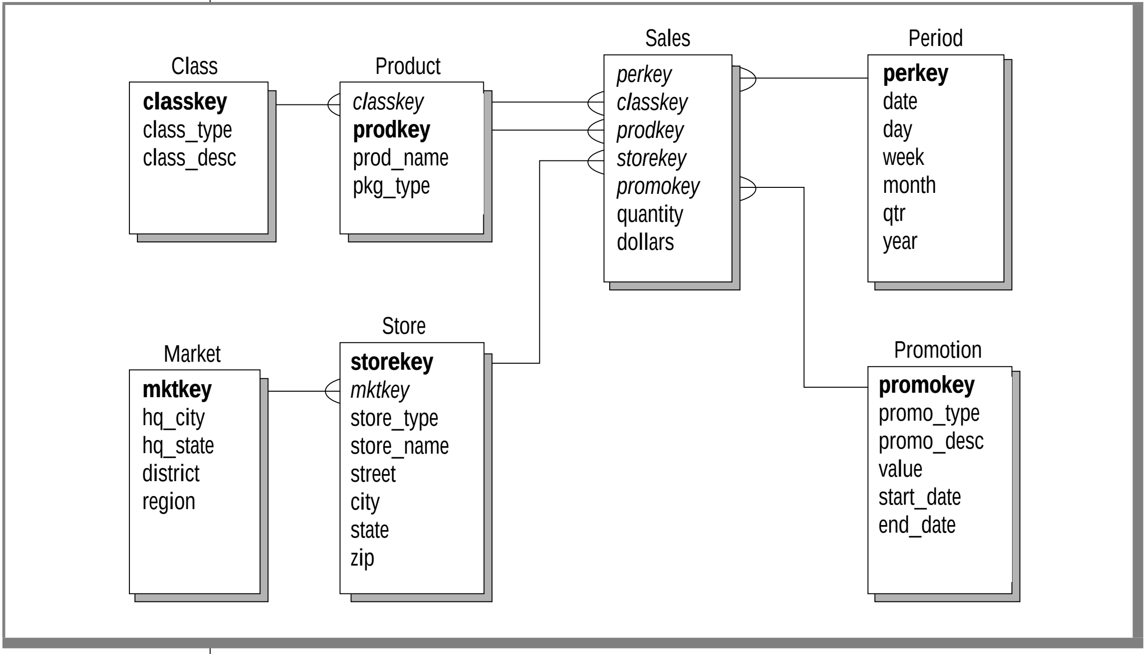
This chapter presents the tables of the basic **Aroma** database and briefly describes the primary-key to foreign-key relationships that link the data in these tables.

This chapter also presents a few of the questions that the **Aroma** database or any Red Brick Decision Server database can answer quickly and efficiently.

Aroma Database Retail Schema

Most of the examples in this guide are based on data from the basic **Aroma** database, which tracks daily retail sales in stores owned by the Aroma Coffee and Tea Company. The retail schema consists of four main *dimension tables*—**Period**, **Product**, **Store**, and **Promotion**—and a **Sales** *fact table*, as well as two *outboard tables*, **Class** and **Market**.

The following figure illustrates this basic schema.



The crow's feet in this diagram indicate a one-to-many relationship between the two tables. For example, each distinct value in the **Perkey** column of the **Period** table can occur only once in that table but many times in the Sales table. Column names in bold are primary-key columns. Column names in italic are foreign-key columns. Column names in bold italic are primary *and* foreign-key columns.

The remainder of this chapter presents sample data from each table so you can see how these primary-key to foreign-key relationships work.

Basic Aroma Schema

In a decision-support database, the tables and columns are named with familiar business terms, making the schema easy to understand and use. A well-designed schema provides the following benefits to application developers and end users:

- Business questions are easy to express as SQL queries.
- Queries run fast and return consistent answers.

The retail **Aroma** schema meets both of these criteria. The **Sales** table contains the everyday measurements of the business—the facts—and the **Store**, **Period**, **Product**, and **Promotion** tables contain the dimensions, or characteristics, of the business. The **Class** and **Market** tables contain information that adds another level of detail to the product and store information.

Most of the examples in this guide use the simple star schema formed by these seven basic tables, but the **Aroma** database also contains a purchasing schema with a more complex design. For details, refer to Appendix A, "The Complete Aroma Database."

Important: *The **Aroma** database does not contain any predefined aggregate tables. For information about using the Informix Vista query rewrite system to accelerate the performance of aggregate queries, refer to the "Informix Vista User's Guide."*

Period, Product, and Class Dimensions

Period Table

The following table displays the first few rows of the **Period** table. The primary-key column is the **Perkey** column:

Perkey	Date	Day	Week	Month	Qtr	YEAR
1	1998-01-01	TH	1	JAN	Q1_98	1998
2	1998-01-02	FR	1	JAN	Q1_98	1998
3	1998-01-03	SA	1	JAN	Q1_98	1998
4	1998-01-04	SU	2	JAN	Q1_98	1998
5	1998-01-05	MO	2	JAN	Q1_98	1998
6	1998-01-06	TU	2	JAN	Q1_98	1998
...						

Product and Class Tables

The following table displays the first few rows of the **Product** table. The primary key is a combination of the **Classkey** and **Prodkey** values:

Classkey	Prodkey	Prod_Name	Pkg_Type
1	0	Veracruzano	No pkg
1	1	Xalapa Lapa	No pkg
1	10	Colombiano	No pkg
1	11	Espresso XO	No pkg

(1 of 2)

Classkey	Prodkey	Prod_Name	Pkg_Type
1	12	La Antigua	No pkg
1	20	Lotta Latte	No pkg
...			

(2 of 2)

If a dimension table contains foreign-key columns that reference other dimension tables, the referenced tables are called *outboard* or *outrigger* tables. For example, the **Classkey** column of the **Product** table is a foreign-key reference to the **Class** table.

The following table displays the first few rows of the **Class** table.

Classkey	Class_Type	Class_Desc
1	Bulk_beans	Bulk coffee products
2	Bulk_tea	Bulk tea products
3	Bulk_spice	Bulk spices
4	Pkg_coffee	Individually packaged coffee products
5	Pkg_tea	Individually packaged tea products
6	Pkg_spice	Individually packaged spice products

Store, Market, and Promotion Dimensions

Dimension tables contain descriptions that data analysts use as they query the database. For example, the **Store** table contains store names and addresses; the **Product** table contains product and packaging information; and the **Period** table contains month, quarter, and year values. Every table contains a primary key that consists of one or more columns; each row in a table is uniquely identified by its primary-key value or values.

Store and Market Tables

The following table displays the first few rows of the **Store** table (some columns were truncated to fit on the page). The primary-key column is **Storekey**; **Mktkey** is a foreign-key reference to the **Market** table.

Storekey	Mktkey	Store_Type	Store_Name	STREET	CITY	STATE	ZIP
1	14	Small	Roasters, Los Gatos	1234 University Ave	Los Gatos	CA	95032
2	14	Large	San Jose Roasting	5678 Bascom Ave	San Jose	CA	95156
3	14	Medium	Cupertino Coffee	987 DeAnza Blvd	Cupertino	CA	97865
4	3	Medium	Moulin Rouge	898 Main Street	New Orleans	LA	70125
5	10	Small	Moon Pennies	98675 University	Detroit	MI	48209
6	9	Small	The Coffee Club	9865 Lakeshore Bl	Chicago	IL	06060
...							

The following table displays the first few rows of the **Market** table.

Mktkey	Hq_city	Hq_state	District	Region
1	Atlanta	GA	Atlanta	South
2	Miami	FL	Atlanta	South
3	New Orleans	LA	New Orleans	South
4	Houston	TX	New Orleans	South
5	New York	NY	New York	North
...				

Promotion Table

The following table displays the first few rows of the **Promotion** table. The primary-key column is **Promokey**.

Promokey	Promo_Type	Promo_Desc	Value	Start_Date	End_Date
0	1	No promotion	0.00	9999-01-01	9999-01-01
1	100	Aroma catalog coupon	1.00	1998-01-01	1998-01-31
2	100	Aroma catalog coupon	1.00	1998-02-01	1998-02-28
3	100	Aroma catalog coupon	1.00	1998-03-01	1998-03-31
4	100	Aroma catalog coupon	1.00	1998-04-01	1998-04-30
5	100	Aroma catalog coupon	1.00	1998-05-01	1998-05-31
...					

Sales Table

The following table displays the first 20 rows of the **Sales** table.

Perkey	Classkey	Prodkey	Storekey	Promokey	Quantity	Dollars
2	2	0	1	116	8	34.00
2	4	12	1	116	9	60.75
2	1	11	1	116	40	270.00
2	2	30	1	116	16	36.00
2	5	22	1	116	11	30.25
2	1	30	1	116	30	187.50
2	1	10	1	116	25	143.75
2	4	10	2	0	12	87.00

(1 of 2)

Perkey	Classkey	Prodkey	Storekey	Promokey	Quantity	Dollars
2	4	11	2	0	14	115.50
2	2	22	2	0	18	58.50
2	4	0	2	0	17	136.00
2	5	0	2	0	13	74.75
2	4	30	2	0	14	101.50
2	2	10	2	0	18	63.00
2	1	22	3	0	11	99.00
2	6	46	3	0	6	36.00
2	5	12	3	0	10	40.00
2	1	11	3	0	36	279.00
2	5	1	3	0	11	132.00
2	5	10	3	0	12	48.00
...						

(2 of 2)

The primary-key column is a combination of values from five columns:

```
perkey, classkey, prodkey, storekey, promokey
```

About the Sales Facts

The **Sales** table is a fact table; the data it contains is easily accessible through the business attributes defined in the tables it references, and it stores large amounts of statistical information about those attributes. The **Sales** table is the largest table in the **Aroma** database and its data is split into two database storage areas (known as *segments*). For information about segments, refer to the *Administrator's Guide*.

Access to business facts must be easy and quick. Red Brick Decision Server provides such access by addressing fact table rows through business dimensions familiar to the query writer. For example, to retrieve sales of La Antigua coffee at the San Jose Roasting Company on January 31, 1999, you simply specify those three dimensions (1/31/99, product name, store name), and the database server quickly retrieves your request.

Multipart Primary Key

The **Sales** table contains a multipart primary key. Each of its five columns is a foreign-key reference to the primary key of another table:

```
perkey, classkey, prodkey, storekey, promokey
```

This primary key links the **Sales** data to the **Period**, **Product**, **Store**, and **Promotion** dimensions. Through such links, figures regarding the sale of a specific product on a particular day in a given city, expressed in terms of dollars and quantities, can be quickly and easily retrieved from the database.

Commonly Asked Questions

Some of the more commonly asked business questions follow.

Easy

- What were the weekly sales of Lotta Latte brand coffee in San Jose during last year?
- What were the average monthly sales of all coffee products in the West during each month of last year?

Moderately Difficult

- How do the sales of Lotta Latte in San Jose compare with its sales in Los Angeles and New York?
- How has the monthly market share of Lotta Latte changed during the last two years in all markets?
- Which suppliers charge the most for bulk tea products?
- What was the most successful promotion last December in California?

Very Difficult Without RISQL Extensions

- What were the running totals for Lotta Latte sales during each month of last year?
- What were the ratios of monthly sales to total sales (expressed as percentages) for Lotta Latte during the same period?
- Which ten cities had the worst coffee sales in 1998 with regard to dollar sales and quantities sold?
- Which Aroma stores fall into the top 25 percent in terms of sales revenue for the first quarter of 1999? Which stores fall into the middle 50 percent, and the bottom 25 percent?

Typical Data Warehousing Queries

Many kinds of commonly asked business questions can be readily expressed as SQL queries. For example, anyone familiar with SQL can write a query that returns the quarterly sales of a given product in a given year.

However, many other commonly asked questions cannot be expressed so easily. Questions that require comparisons often challenge both the query writers and SQL itself. For example, a question requesting a comparison of weekly, monthly, quarterly, and yearly values is one of the simplest questions posed during a sales analysis, but expressing this question as a query represents a formidable challenge to the query writer, the query language, and the database server.

Business questions that request sequential processing are very difficult to express as SQL queries. To derive a simple running total, for example, data analysts typically run several queries with a client tool, then paste the results together using another tool. This approach is awkward because it requires a sophisticated user, floods the network with data, and takes place on a client that is typically much slower than a database server.

The RISQL extensions to SQL provide a better solution because they are easy to use, reduce network traffic, and perform sequential calculations that execute quickly on the server.

Summary

This chapter briefly described the retail schema of the **Aroma** database and suggested some typical business questions that a Red Brick Decision Server database can answer.

A decision-support database is designed to be queried: It has a few easy-to-understand tables, provides exceptional query performance, and guarantees data integrity. To this end, the primary tables in a Red Brick Decision Server database are:

- Few in number
- Designed using the analyst's vocabulary
- Reflective of the natural dimensions of the business

The remainder of this self-study guide consists of detailed examples that show how to write commonly asked business questions. Most of these examples are based on the **Aroma** retail schema. Some additional tables are used sporadically in the more advanced examples. These tables are described in Appendix A, "The Complete Aroma Database."

Chapter

Basic Queries

2

In This Chapter

Through a series of simple examples, this chapter illustrates how to retrieve data from a Red Brick Decision Server database with standard SQL SELECT statements.

This chapter describes how to:

- Retrieve specific columns and rows from a relational database table
- Perform logical operations on retrieved data
- Use wildcard characters in search conditions
- Retrieve data from more than one table
- Order data and calculate subtotals on numeric columns
- Perform aggregate calculations with set functions
- Group data
- Perform arithmetic operations on retrieved data
- Remove rows from the result set if specified columns contain NULLs, zeroes, or spaces

Using the SELECT Statement to Retrieve Data

Question

What regions, districts, and markets are defined in the **Aroma** database?

Example Query

```
select * from market;
```

Result

Mktkey	HQ_City	HQ_State	District	Region
1	Atlanta	GA	Atlanta	South
2	Miami	FL	Atlanta	South
3	New Orleans	LA	New Orleans	South
4	Houston	TX	New Orleans	South
5	New York	NY	New York	North
6	Philadelphia	PA	New York	North
7	Boston	MA	Boston	North
8	Hartford	CT	Boston	North
9	Chicago	IL	Chicago	Central
10	Detroit	MI	Chicago	Central
11	Minneapolis	MN	Minneapolis	Central
12	Milwaukee	WI	Minneapolis	Central
14	San Jose	CA	San Francisco	West
15	San Francisco	CA	San Francisco	West

(1 of 2)

Mktkey	HQ_City	HQ_State	District	Region
16	Oakland	CA	San Francisco	West
17	Los Angeles	CA	Los Angeles	West
19	Phoenix	AZ	Los Angeles	West

(2 of 2)

Retrieving Data: SELECT Statement

You use SELECT statements to retrieve columns and rows of data from database tables, to perform arithmetic operations on the data, and to group, order, or group and order the data. In most cases, a SELECT statement contains a simple query expression that begins with the SELECT keyword and is followed by one or more clauses and subclauses. (For detailed information about more complex query expressions, refer to the *SQL Reference Guide*.)

The most basic SELECT statement contains two keywords, SELECT and FROM:

```
SELECT select_list
FROM table_list;
```

select_list Column names or SQL expressions are separated by commas. An asterisk (*) can also be used.

table_list Table names are separated by commas. Referenced tables must contain the column names in *select_list*.

SELECT and FROM (and all other words shown in uppercase in subsequent references to syntax in this guide) are reserved SQL keywords. These words must be used exactly as defined by the SQL standard. SQL is not case sensitive, so keywords can be written in uppercase or lowercase.

About the Query

The example query retrieves the entire contents of the **Market** table. The asterisk symbol (*) is the SQL abbreviation for “all column names that occur in *table_list*.” All column names in the **Market** table could be listed instead.

Red Brick Decision Server also supports *explicit tables*, whereby this query could be stated simply as:

```
table market;
```

Usage Notes

Names in a select list must be defined in tables listed in the FROM clause; exceptions to this rule are discussed later in this chapter. Columns are returned from the database in the order listed. If you use an asterisk, columns are returned as stored in the database table.

The semicolon (;) at the end of each example in this guide is not part of SQL syntax; it is an end-of-statement marker required by the RISQL Entry Tool and the RISQL Reporter. Depending on the interactive SQL tool you use to enter queries, you might not need to specify such a marker.

Using SELECT List to Retrieve Specific Columns

Question

Which districts and regions are defined in the **Aroma** database?

Example Query

```
select district, region
from market;
```

Result

District	Region
Atlanta	South
Atlanta	South
New Orleans	South
New Orleans	South
New York	North
New York	North
Boston	North
Boston	North
Chicago	Central
Chicago	Central
Minneapolis	Central
Minneapolis	Central
San Francisco	West
San Francisco	West
San Francisco	West
Los Angeles	West
Los Angeles	West

Retrieving Specific Columns

By naming the columns in the select list of a SELECT statement, you can retrieve a specific set of columns from any table. Columns are returned in the order specified in the select list.

About the Query

The example query requests a list of districts and their corresponding regions from the **Market** table.

Usage Notes

Although column names in the select list must be defined in the tables referenced in the FROM clause, other expressions can also occur in the select list. Several examples of such expressions are discussed later in this guide.

When the select list does not include all the columns in a table, a query might return duplicate rows, as in the previous example query. You can eliminate the duplicates by using the DISTINCT keyword. For example, the following query returns only the names of distinct districts and regions in the **Market** table:

```
select distinct district, region
from market;
```

District	Region
Atlanta	South
Boston	North
Chicago	Central
Los Angeles	West
Minneapolis	Central

(1 of 2)

District	Region
New Orleans	South
New York	North
San Francisco	West

(2 of 2)

Using the WHERE Clause to Retrieve Specific Rows

Question

What products are sold without packaging?

Example Query

```
select prod_name, pkg_type
from product
where pkg_type = 'No pkg';
```

Result

Prod_Name	Pkg_Type
Veracruzano	No pkg
Xalapa Lapa	No pkg
Colombiano	No pkg
Expresso XO	No pkg
La Antigua	No pkg
Lotta Latte	No pkg
Cafe Au Lait	No pkg

(1 of 2)

Prod_Name	Pkg_Type
NA Lite	No pkg
Aroma Roma	No pkg
Demitasse Ms	No pkg
Darjeeling Number 1	No pkg
Darjeeling Special	No pkg
Assam Grade A	No pkg
Assam Gold Blend	No pkg
Earl Grey	No pkg
English Breakfast	No pkg
Irish Breakfast	No pkg
Special Tips	No pkg
Gold Tips	No pkg
Breakfast Blend	No pkg
Ruby's Allspice	No pkg
Coffee Mug	No pkg
Travel Mug	No pkg
Aroma t-shirt	No pkg
Aroma baseball cap	No pkg

(2 of 2)

Retrieving Specific Rows: WHERE Clause

By including a set of *logical conditions* in a query, you can retrieve a specific set of rows from a table. Logical conditions are declared in the WHERE clause. If a row satisfies the conditions, the query returns the row; if not, the row is discarded. Logical conditions are also called *search conditions*, *predicates*, *constraints*, or *qualifications*.

The WHERE Clause

```
SELECT select_list
FROM table_list
[WHERE search_condition];
```

search_condition This condition evaluates to true or false.

The square brackets ([ ]) indicate that the WHERE clause is optional.

About the Query

The example query retrieves and displays the names of products that are not prepacked or packaged. Red Brick Decision Server evaluates the following condition for each row of the **Product** table and returns only those rows that satisfy the condition:

```
pkg_type = 'No pkg'
```

Usage Notes

A *character literal* is a character string enclosed within single quotes. To represent a single quote in a character literal, use two single quotes (''). For example:

```
'Scarlet O''Hara'
```

Character literals must be expressed as stored in the database—in either uppercase or lowercase. For example, the decision server evaluates the following condition:

```
class_type = 'Bulk_beans'
```

The condition is false when the referenced column contains the following string:

```
'BULK_beans'
```

Set functions are not allowed in the WHERE clause. For more information about set functions, refer to page 2-31.

Using the AND, NOT, and OR Connectives to Specify Compound Conditions

Question

What cities and districts are located in the southern or western regions?

Example Query

```
select hq_city, district, region
from market
where region = 'South' or region = 'West';
```

Result

HQ_City	District	Region
Atlanta	Atlanta	South
Miami	Atlanta	South
New Orleans	New Orleans	South
Houston	New Orleans	South
San Jose	San Francisco	West
San Francisco	San Francisco	West
Oakland	San Francisco	West
Los Angeles	Los Angeles	West
Phoenix	Los Angeles	West

Specifying Compound Conditions: AND, NOT, OR

Most queries written for decision-support analysis contain *compound conditions*. Compound conditions are simple conditions joined by *logical connectives*. SQL contains the following logical connectives.

Connective	Name	Order of Precedence
( )	Parentheses (force order of evaluation)	1
AND	And	3
NOT	Negation	2
OR	Or	4

The server evaluates compound conditions as follows: all the NOT operators first, all the AND connectives second, and all the OR connectives last. This evaluation order is commonly known as the *order of precedence*.

You can control the order of evaluation by grouping compound conditions with parentheses. In a nest of parentheses, Red Brick Decision Server evaluates the innermost set of parentheses first, the next innermost set next, and so on. Whenever the logic of a compound condition is not obvious, make it obvious with parentheses.

About the Query

The example query retrieves all cities and districts in the southern or western regions. All rows that have *South* or *West* in their **Region** column satisfy the compound condition and are returned in the result table.

Usage Notes

When in doubt about the order of evaluation, force the order by grouping conditions with parentheses.

Using the AND Connective to Specify Complex Search Conditions

Question

Which large or small Aroma stores are located in Los Angeles or San Jose?

Example Query

```
select store_type, store_name, city
from store
where (store_type = 'Large' or store_type = 'Small')
   and (city = 'Los Angeles' or city = 'San Jose');
```

Result

Store_Type	Store_Name	City
Large	San Jose Roasting Company	San Jose
Large	Beaches Brew	Los Angeles
Small	Instant Coffee	San Jose

Specifying Complex Search Conditions

Search conditions, especially those written for decision-support analysis, can become complex. Though constructed from simple conditions that use the logical connectives AND, OR, and NOT, complex conditions might be difficult to understand. Fortunately, SQL is free-form, so the logical structure of these conditions can be shown by using tab characters, blanks, and newline characters to define white space and logical relationships.

About the Query

The example query retrieves and displays the names of Aroma stores that are both large or small and located in Los Angeles or San Jose.

The parentheses in this query are essential because the AND connective has a higher precedence than the OR connective. If you remove the parentheses, the query returns a different result table.

Store_Type	Store_Name	City
Large	San Jose Roasting Company	San Jose
Large	Beaches Brew	Los Angeles
Small	Instant Coffee	San Jose
Large	Miami Espresso	Miami
Large	Olympic Coffee Company	Atlanta

Usage Notes

A query retrieves and displays any data that is not explicitly excluded by its search condition, and a query with only a few general conditions can return an enormous number of rows.

Whenever you doubt how the server might evaluate a compound condition, explicitly group the conditions with parentheses to force the order of evaluation.

Using the Greater-Than (>) and Less-Than or Equal-To (<=) Operators

Question

Which cities and districts are identified by **Mktkey** values that are greater than 4 and less than or equal to 12?

Example Query

```
select mktkey, hq_city, hq_state, district
from market
where mktkey > 4
   and mktkey <= 12;
```

Result

Mktkey	HQ_City	HQ_State	District
5	New York	NY	New York
6	Philadelphia	PA	New York
7	Boston	MA	Boston
8	Hartford	CT	Boston
9	Chicago	IL	Chicago
10	Detroit	MI	Chicago
11	Minneapolis	MN	Minneapolis
12	Milwaukee	WI	Minneapolis

Using Comparison Operators

Conditions evaluate to true or false and can be expressed with comparison operators or comparison predicates. Comparison predicates are described on the next two pages.

SQL contains the following comparison operators.

Operator	Name
=	equal
<	less than
>	greater than
<>	not equal
>=	greater than or equal
<=	less than or equal

About the Query

The example query retrieves and displays all cities and districts whose **Mktkey** is greater than 4 but less than or equal to 12.

The **Mktkey** column contains integer values, which are comparable to other numeric values. If you compare an integer to a character, however, the server returns an error message:

```
select mktkey, hq_city, hq_state, district
from market
where mktkey > '4';

** ERROR ** (19) Operands of comparison must have comparable
datatypes.
```

Usage Notes

Conditions must compare values of comparable data types. If you attempt to compare unlike data types, the server returns either an error message or an incorrect result. Comparison operators can be used to compare one character string with another, as the following legal condition illustrates:

```
(city > 'L')
```

For more information about comparable data types, refer to the *SQL Reference Guide*.

Using the IN Comparison Predicate

Question

What cities are in the Chicago, New York, and New Orleans districts?

Example Query

```
select hq_city, hq_state, district
from market
where district in
   ('Chicago', 'New York', 'New Orleans');
```

Result

HQ_City	HQ_State	District
New Orleans	LA	New Orleans
Houston	TX	New Orleans
New York	NY	New York

(1 of 2)

HQ_City	HQ_State	District
Philadelphia	PA	New York
Chicago	IL	Chicago
Detroit	MI	Chicago

(2 of 2)

Using Comparison Predicates

A simple condition can be expressed with the following SQL comparison predicates:

Predicate
BETWEEN *expression1* AND *expression2*
LIKE *pattern*
IN (*list*)
IS NULL
IS NOT NULL
ALL
SOME or ANY
EXISTS

Examples of the ALL, SOME or ANY, and EXISTS predicates are presented in Chapter 4, “Comparison Queries.”

For syntax descriptions and examples of all these predicates, as well as detailed definitions of simple and complex *expressions*, refer to the *SQL Reference Guide*.

About the Query

The example query lists all the cities in the Chicago, New York, and New Orleans districts. It could also be written with the *equals* comparison operator (=) and a set of OR conditions:

```
where district = 'Chicago'
   or district = 'New York'
   or district = 'New Orleans'
```

Usage Notes

Strive to write logical sets of conditions that are simple, easy to understand, and easy to maintain. Always clarify the logical structure of your compound conditions with ample white space, define logical blocks by indentation, and force evaluation precedence with parentheses.

Using the Percent Sign (%) Wildcard

Question

Which cities are in districts that begin with the letters *Min*?

Example Query

```
select district, hq_city
from market
where district like 'Min%';
```

Result

District	HQ_City
Minneapolis	Minneapolis
Minneapolis	Milwaukee

Using Wildcard Characters

Previous queries have expressed conditions that match complete character strings. With the LIKE predicate and the two wildcard characters (percent sign (%) and underscore (_)), you can also express conditions that match a portion of a character string (a substring).

The percent (%) wildcard matches any character string. For example:

- `like 'TOT%'` is true for any string that begins with `'TOT'`.
- `like '%ZERO%'` is true for any string that contains '`ZERO`'.
- `like '%FRESH'` is true for any string that ends with '`FRESH`' and does not contain trailing blanks. Trailing blanks in character data are deemed significant when LIKE constraints are applied.

The percent sign (%) can also be used to search for a null character string—zero (0) characters.

The underscore wildcard (_) matches any one character in a fixed position. For example:

- `like '_EE_'` is true for any four-letter string whose two middle characters are '`EE`'.
- `like '%LE_N%'` is true for any string that contains the pattern '`LE_N`'. The strings '`CLEAN`', '`KLEEN`', and '`VERY KLEEN`' all match this pattern.

About the Query

The example query retrieves the names of all districts that begin with the characters *Min* and lists the cities in these districts. The wildcard percent sign (%) allows for any character combination (including blank spaces) after the *n* in *Min*, but characters that precede the *n* must match the character pattern exactly as stored.

Usage Notes

A LIKE condition is true when its pattern matches a substring in a column. If the pattern contains no wildcard characters, the pattern must match the column entry exactly. For example, the following condition is true only when the column entry contains the character string APRIL and nothing else:

```
month like 'APRIL'
```

In other words, this condition is equivalent to:

```
month = 'APRIL'
```

The LIKE predicate can be used only on columns that contain character strings.

Using Simple Joins

Question

What were the daily sales totals for Easter products sold on weekends on a type 900 promotion in 1999 and which stores registered those sales?

Example Query 1

```
select prod_name, store_name, day, dollars
from promotion, product, period, store, sales
where promotion.promokey = sales.promokey
   and product.prodkey = sales.prodkey
   and product.classkey = sales.classkey
   and period.perkey = sales.perkey
   and store.storekey = sales.storekey
   and prod_name like 'Easter%'
   and day in ('SA', 'SU')
   and promo_type = 900
   and year = 1999;
```

Example Query 2

```
select prod_name, store_name, day, dollars
from promotion natural join sales
   natural join product
   natural join period
   natural join store
where prod_name like 'Easter%'
   and day in ('SA', 'SU')
   and promo_type = 900
   and year = 1999;
```

Two Queries, Same Result

Prod_Name	Store_Name	Day	Dollars
Easter Sampler Basket	Olympic Coffee Company	SA	150.00

Joining Dimensions and Facts

So far, the queries in this chapter have retrieved data from a single table; however, most queries *join* information from different tables. Typically, dimension tables are joined to fact tables to constrain the facts in interesting ways. For example, you can join the **Sales** fact table to its **Store** and **Product** dimensions to get sales figures per product per store, or to its **Period** and **Product** dimensions to get sales figures per product per week.

About the Queries

This business question requires a join of five tables in the **Aroma** retail schema: the **Sales** fact table and its **Product**, **Period**, **Store**, and **Promotion** dimensions.

To join tables in a query, you must give the database server explicit instructions on how to perform the join. In Example Query 1, the joins are specified in the WHERE clause with five simple conditions that join the **Sales** table to its dimensions over its five primary key columns. The **Product** table has a two-part primary key, so it is joined to the **Sales** table over two columns: **Prodkey** and **Classkey**.

Because all of these conditions involve *identically named* joining columns, the question can alternatively be posed in the style of Example Query 2, using *natural joins* in the FROM clause. This approach to joining tables works well with the **Aroma** database because its main tables form a simple star schema and all the foreign-key columns use the same names as the primary keys they reference.

Natural joins operate on *all pairs* of identically named columns shared by the tables; therefore, in Example Query 2 the **Sales** and **Product** tables are joined over both the **Classkey** column and the **Prodkey** column.

There are two other ways to join these tables in the FROM clause and get the same result; for details, refer to Chapter 5, “Joins and Unions,” which identifies the different types of join queries you can write and presents more examples.

Usage Notes

Any two tables can be joined over columns with comparable data types; joins are not dependent on the primary-key to foreign-key relationships used in this example.

Using the ORDER BY Clause

Question

What were the sales figures for Assam Gold Blend and Earl Grey at the Instant Coffee store during November 1999? Order the figures for each product from highest to lowest.

Example Query

```
select prod_name, store_name, dollars
from store natural join sales
    natural join product
    natural join period
where (prod_name like 'Assam Gold%'
   or prod_name like 'Earl%')
   and store_name like 'Instant%'
   and month = 'NOV'
   and year = 1999
order by prod_name, dollars desc;
```

Result

Prod_Name	Store_Name	Dollars
Assam Gold Blend	Instant Coffee	96.00
Assam Gold Blend	Instant Coffee	78.00
Assam Gold Blend	Instant Coffee	66.00
Assam Gold Blend	Instant Coffee	58.50
Assam Gold Blend	Instant Coffee	58.50
Assam Gold Blend	Instant Coffee	39.00
Assam Gold Blend	Instant Coffee	39.00
Assam Gold Blend	Instant Coffee	32.50
Earl Grey	Instant Coffee	48.00
Earl Grey	Instant Coffee	45.50
Earl Grey	Instant Coffee	42.00
Earl Grey	Instant Coffee	32.00
Earl Grey	Instant Coffee	24.00
Earl Grey	Instant Coffee	20.00

Ordering the Result Table: ORDER BY Clause

You can use the ORDER BY clause to sort the result table of a query by the values in one or more specified columns. The default sort order is ascending (ASC); the DESC keyword changes the sort order to descending for the specified column, as follows:

```
order by prod_name, 3 desc
```

To order results by an expression in the select list (for example, a set function), specify a column alias for the expression and then name the alias in the ORDER BY clause. For more information about column aliases, refer to page 2-33.

Syntax of the ORDER BY clause

```
SELECT select_list
FROM table_list
[WHERE search_condition]
[ORDER BY order_list] ;
```

order_list A list of columns by which data is ordered. Columns in the *order_list* need not occur in the *select_list* but must exist in tables referenced in the FROM clause.

About the Query

The example query retrieves Assam Gold Blend and Earl Grey sales figures at the Instant Coffee store during November 1999. The query sorts the results by product and total daily sales.

Usage Notes

The ORDER BY clause must follow the other clauses in the SELECT statement (except the SUPPRESS BY clause) and include a list of columns to be ordered. A column can be referenced by its name, column alias, or position (ordinal number) in the select list. For example, the ORDER BY clause on the facing page could be written as follows:

```
order by prod_name, 3 desc
```

By specifying columns in *order_list* that are not in the *select_list*, you can order data by columns that are not displayed in the result table.

Calculating Subtotals

Question

What were the daily sales and monthly subtotals for Assam Gold Blend, Darjeeling Special, and Earl Grey teas at the Instant Coffee store during November 1999? What is the monthly subtotal for all three products?

Example Query

```
select prod_name, store_name, dollars
from store natural join sales
   natural join product
   natural join period
where prod_name in ('Assam Gold Blend', 'Earl Grey',
   'Darjeeling Special')
   and store_name like 'Instant%'
   and month = 'NOV'
   and year = 1999
order by prod_name, dollars desc
   break by prod_name summing 3;
```

Result

Prod_Name	Store_Name	Dollars
Assam Gold Blend	Instant Coffee	96.00
Assam Gold Blend	Instant Coffee	78.00
Assam Gold Blend	Instant Coffee	66.00
Assam Gold Blend	Instant Coffee	58.50
Assam Gold Blend	Instant Coffee	58.50

(1 of 2)

Result

Prod_Name	Store_Name	Dollars
Assam Gold Blend	Instant Coffee	39.00
Assam Gold Blend	Instant Coffee	39.00
Assam Gold Blend	Instant Coffee	32.50
Assam Gold Blend	NULL	467.50
Darjeeling Special	Instant Coffee	207.00
Darjeeling Special	Instant Coffee	168.00
Darjeeling Special	Instant Coffee	149.50
Darjeeling Special	Instant Coffee	144.00
Darjeeling Special	Instant Coffee	138.00
Darjeeling Special	Instant Coffee	132.00
Darjeeling Special	Instant Coffee	96.00
Darjeeling Special	Instant Coffee	69.00
Darjeeling Special	Instant Coffee	60.00
Darjeeling Special	Instant Coffee	60.00
Darjeeling Special	Instant Coffee	48.00
Darjeeling Special	NULL	1271.50
Earl Grey	Instant Coffee	48.00
Earl Grey	Instant Coffee	45.50
Earl Grey	Instant Coffee	42.00
Earl Grey	Instant Coffee	32.00
Earl Grey	Instant Coffee	24.00
Earl Grey	Instant Coffee	20.00
Earl Grey	NULL	211.50
NULL	NULL	1950.50

(2 of 2)

Calculating Subtotals: BREAK BY Clause

When a query contains an ORDER BY clause, you can use a BREAK BY clause to add control breaks to the result set and calculate subtotals on numeric columns. The BREAK BY clause also computes a grand total of the subtotals and displays this value in the final row of the report. This clause is a RISQL extension to the ANSI SQL-92 standard.

Syntax of the BREAK BY Clause

```
SELECT select_list
FROM table_list
[WHERE search_condition]
[ORDER BY order_list]
   [BREAK BY order_reference SUMMING select_reference_list];
```

order_reference	A column used in the *order_list*.
select_reference_list	A numeric expression used in the *select_list*.

About the Query

The example query lists the daily totals for three tea products at the Instant Coffee store in November 1999. A subtotal of the sales figures is calculated for each product, and the grand total for all three products is displayed at the end of the report. The *order_reference* is **Prod_Name** and the *select_reference_list* consists of a single column reference (3, which refers to the **Dollars** column).

Usage Notes

As well as performing simple aggregate calculations on ordered sets of rows, the BREAK BY clause makes the contents of a long result set easier to read and absorb.

If the query contains a RISQL display function, the ORDER BY clause can contain another RISQL extension, the RESET BY subclause. For more details, refer to page 3-11.

A query that includes a BREAK BY clause cannot be used as a query expression in an INSERT INTO...SELECT statement.

Using the SUM, AVG, MAX, MIN, COUNT Set Functions

Question

What were the total Lotta Latte sales figures in Los Angeles for 1999? What were the average, maximum, and minimum daily sales figures for that year, and how many daily totals were counted to produce these aggregate values?

Example Query

```
select sum(dollars), avg(dollars), max(dollars), min(dollars),
   count(*)
from store natural join sales
   natural join period
   natural join product
where prod_name like 'Lotta Latte%'
   and year = 1999
   and city like 'Los Ang%';
```

Result

Sum	Avg	Max	Min	Count
13706.50	171.33125000	376.00	39.00	80

Using Set Functions

Set functions operate on groups of values. For example, SUM(dollars) calculates the total dollars returned in a result table, and AVG(dollars) returns the average. The SQL set functions listed in the following table can occur one or more times in the select list.

Function	Description
SUM(*expression*)	Calculates the sum of all the values in *expression*.
SUM(DISTINCT *expression*)	Calculates the sum of distinct values in *expression*.
AVG(*expression*)	Calculates the average of all the values in *expression*.
AVG(DISTINCT *expression*)	Calculates the average of distinct values in *expression*.
MAX(*expression*)	Determines the maximum value in *expression*.
MIN(*expression*)	Determines the minimum value in *expression*.
COUNT(*)	Counts the number of rows returned.
COUNT(*expression*)	Counts the number of non-null values in *expression*.
COUNT(DISTINCT*expression*)	Counts the number of distinct non-null values in *expression*.

You can replace *expression* with any column name or numeric expression. Each function, except COUNT(*), ignores NULL values when calculating the returned aggregate value.

About the Query

The example query retrieves sales figures for Lotta Latte in Los Angeles during 1999. The result set also includes the average, maximum, and minimum sales during the year, and the number of daily totals on which those calculations are based.

Usage Notes

If the result set will contain individual values as well as aggregate values, the query must contain a GROUP BY clause. For more information about the GROUP BY clause, refer to page 2-35.

Set functions can be used as arguments to RISQL display functions; however, display functions cannot be used as arguments to set functions. (Display functions are discussed in Chapter 3, "Data Analysis."

Using Column Aliases

Question

What were the annual Lotta Latte sales figures in Los Angeles stores during 1999? What were the average, maximum, and minimum sales figures during this time period, and how many daily totals were counted in these aggregate values? Specify headings for the aggregate result columns.

Example Query

```
select sum(dollars) as dol_sales, avg(dollars) as avg_sales,
   max(dollars) as max_dol, min(dollars) as min_dol,
   count(*) as num_items
from store natural join sales
   natural join period
   natural join product
where prod_name like 'Lotta Latte%'
   and year = 1999
   and city like 'Los Ang%';
```

Result

Dol_Sales	Avg_Sales	Max_Dol	Min_Dol	Num_Items
13706.50	171.33125000	376.00	39.00	80

Using Column Aliases: AS

By default, the SELECT command returns values calculated by set functions but does not label the returned values with headings. You can specify a label, or *column alias*, for any column with the keyword AS followed by a character string. This alias can then be referenced in later clauses in the query.

For example, the following AS clause assigns the alias **Dol_Sales** to the **Dollars** column:

```
dollars as dol_sales
```

Column aliases are most useful when used to reference expressions from the select list in later clauses, as shown on page 2-35.

About the Query

The example query returns the same result set as the previous query in this chapter; however, in this case, column aliases are assigned to create headings for the aggregated results.

Usage Notes

To improve the readability of your result tables, assign column aliases to all set functions that occur in your query select list.

An alias is a database identifier; it must begin with a letter and have a maximum length of 128 characters. Zero or more letters, digits, or underscores can follow the initial letter. A keyword cannot serve as a database identifier. For more details, refer to the *SQL Reference Guide*.

A column alias can occur anywhere in a SELECT statement to designate the column to which it refers (for example, in a WHERE, ORDER BY, GROUP BY, or HAVING clause).

Important: *If the value contained in the column referenced by the column alias is the result of a set function, it cannot occur in the WHERE clause; however, it can occur in the HAVING clause.*

Using the GROUP BY Clause to Group Rows

Question

What were the annual totals for sales of coffee mugs in 1998 in each district? What were the average, maximum, and minimum sales during this time period? List the results by district.

Example Query

```
select district as district_city, sum(dollars) as dol_sales,
   avg(dollars) as avg_sales, max(dollars) as max_sales,
   min(dollars) as min_sales
from market natural join store
    natural join sales
    natural join period
    natural join product
where prod_name like '%Mug%'
and year = 1998
group by district_city
order by dol_sales desc;
```

Result

District_City	Dol_Sales	Avg_Sales	Max_Sales	Min_Sales
Atlanta	1378.30	35.34102564	98.55	4.00
Los Angeles	711.60	30.93913043	98.55	9.95
San Francisco	410.45	25.65312500	54.75	5.00

Grouping Rows: GROUP BY Clause

Set functions operate on all rows of a result table or on groups of rows defined by a GROUP BY clause. For example, you can group the sales for each market and calculate the respective sum, maximum, and minimum values.

Syntax of the GROUP BY Clause

```
SELECT select_list
FROM table_list
[WHERE search_condition]
[GROUP BY group_list]
[ORDER BY order_list]
   [BREAK BY order_reference SUMMING
select_reference_list];
```

group_list A list of column names (either in *select_list* or in tables listed in the FROM clause) or column aliases in *select_list*. All nonaggregated columns in *select_list* must appear in *group_list*.

About the Query

The example query retrieves annual sales totals for coffee mugs in 1998 (they are sold in three districts only), ordering the figures from highest to lowest. Conceptually speaking, the server processes this query as follows:

1. Retrieves all rows of data from tables specified in the FROM clause, joins the rows from separate tables, and generates an intermediate result table.
2. Retains all rows from the intermediate result table that satisfy the search condition specified in the WHERE clause.
3. Divides the result table into groups specified in the GROUP BY clause.
4. Processes all set functions on specified groups for the entire result table.
5. Orders results according to the ORDER BY clause.
6. Returns only those columns specified in the select list.

Usage Notes

You can accelerate the performance of aggregate queries, queries that contain set functions or GROUP BY clauses, with the Informix Vista query rewrite system. For details, refer to the *Informix Vista User's Guide*.

An ORDER BY clause references items in the select list by column name, column alias, or position. However, if the item in the order list is a set function, it must be referenced by its alias (**Dol_Sales**) or position number because it has no column name. For more information about column aliases, refer to page 2-33.

Using the GROUP BY Clause to Produce Multiple Groups

Question

What were the total sales in each city during 1998 and 1999? List city names by year within their region and district.

Example Query

```
select year, region, district, city, sum(dollars) as sales
from market natural join store
   natural join sales
   natural join product
   natural join period
where year in (1998, 1999)
group by year, region, district, city
order by year, region, district, city;
```

Result

Year	Region	District	City	Sales
1998	Central	Chicago	Chicago	133462.75
1998	Central	Chicago	Detroit	135023.50
1998	Central	Minneapolis	Milwaukee	172321.50
1998	North	Boston	Boston	184647.50
1998	North	Boston	Hartford	69196.25
1998	North	New York	New York	181735.00
1998	North	New York	Philadelphia	172395.75
1998	South	Atlanta	Atlanta	230346.45
1998	South	Atlanta	Miami	220519.75
1998	South	New Orleans	Houston	183853.75
1998	South	New Orleans	New Orleans	193052.25
1998	West	Los Angeles	Los Angeles	219397.20
1998	West	Los Angeles	Phoenix	192605.25
1998	West	San Francisco	Cupertino	180088.75
1998	West	San Francisco	Los Gatos	176992.75
1998	West	San Francisco	San Jose	395330.25
1999	Central	Chicago	Chicago	131263.00
1999	Central	Chicago	Detroit	136903.25
1999	Central	Minneapolis	Milwaukee	173844.25
1999	Central	Minneapolis	Minneapolis	132125.75
1999	North	Boston	Boston	189761.00
1999	North	Boston	Hartford	135879.50

(1 of 2)

Year	Region	District	City	Sales
1999	North	New York	New York	171749.75
1999	North	New York	Philadelphia	171759.50
1999	South	Atlanta	Atlanta	229615.05
1999	South	Atlanta	Miami	234458.90
1999	South	New Orleans	Houston	186394.25
1999	South	New Orleans	New Orleans	190441.75
1999	West	Los Angeles	Los Angeles	228433.00
1999	West	Los Angeles	Phoenix	197044.50
1999	West	San Francisco	Cupertino	196439.75
1999	West	San Francisco	Los Gatos	175048.75
1999	West	San Francisco	San Jose	398829.10

(2 of 2)

Nesting Grouped Results: GROUP BY Clause

When several column names occur in a GROUP BY clause, the result table is divided into groups within groups. For example, if you specify column names for *year*, *region*, and *district* in the GROUP BY clause, the returned figures are divided by year, each year is divided by region, and each region is divided by district.

Syntax of the GROUP BY Clause

```
SELECT select_list
FROM table_list
[WHERE search_condition]
[GROUP BY group_list]
[ORDER BY order_list]
   [BREAK BY order_reference SUMMING
select_reference_list];
```

group_list A list of column names (in either the *select_list* or the tables in the *table_list*) or column aliases in the *select_list*. Columns that do not participate in a set function (nonaggregate columns) in the *select_list* must appear in the *group_list*.

About the Query

The example query retrieves annual sales of all products for each city during 1998 and 1999. The sales figures are both grouped and ordered by year, region, district, and city.

Important: *The cities referred to in this query are the city locations of each store, as defined in the* ***Store*** *table, not the cities defined as* ***hq_cities*** *in the* ***Market*** *table.*

Usage Notes

If the select list includes an aggregation function but the query has no GROUP BY clause, all column references must be aggregation functions.

Using the Division Operator (/)

Question

What was the average price per sale of each product during 1998? Calculate the average as the total sales dollars divided by the total sales quantity.

Example Query

```
select prod_name, sum(dollars) as total_sales,
   sum(quantity) as total_qty,
   string(sum(dollars)/sum(quantity), 7, 2) as price
from product natural join sales
   natural join period
where year = 1998
group by prod_name
order by price;
```

Result

Prod_Name	Total_Sales	Total_Qty	Price
Gold Tips	38913.75	11563	3.36
Special Tips	38596.00	11390	3.38
Earl Grey	41137.00	11364	3.61
Assam Grade A	39205.00	10767	3.64
Breakfast Blend	42295.50	10880	3.88
English Breakfast	44381.00	10737	4.13
Irish Breakfast	48759.00	11094	4.39
Coffee Mug	1054.00	213	4.94
Darjeeling Number 1	62283.25	11539	5.39
Ruby's Allspice	133188.50	23444	5.68

(1 of 2)

Prod_Name	Total_Sales	Total_Qty	Price
Assam Gold Blend	71419.00	11636	6.13
Colombiano	188474.50	27548	6.84
Aroma Roma	203544.00	28344	7.18
La Antigua	197069.50	26826	7.34
Veracruzano	201230.00	26469	7.60
Expresso XO	224020.00	28558	7.84
Aroma baseball cap	15395.35	1953	7.88
Lotta Latte	217994.50	26994	8.07
Cafe Au Lait	213510.00	26340	8.10
Aroma Sounds Cassette	5206.00	620	8.39
Xalapa Lapa	251590.00	29293	8.58
NA Lite	231845.00	25884	8.95
Demitasse Ms	282385.25	28743	9.82
Aroma t-shirt	20278.50	1870	10.84
Travel Mug	1446.35	133	10.87
Darjeeling Special	127207.00	10931	11.63
Spice Sampler	6060.00	505	12.00
Aroma Sounds CD	7125.00	550	12.95
French Press, 2-Cup	3329.80	224	14.86
Spice Jar	4229.00	235	17.99
French Press, 4-Cup	3323.65	167	19.90
Tea Sampler	13695.00	550	24.90
...			

(2 of 2)

Using the Arithmetic Operators: ( ), +, –, *, /

You can perform arithmetic operations within a select list or within a search condition. A complete set of arithmetic operators is listed in the following table. The order of evaluation precedence is from highest to lowest (top to bottom) and, within a given level, left to right, in the table:

Operator	Name
( )	Forces order of evaluation
+, –	Positive and negative
*, /	Multiplication and division
+, –	Addition and subtraction

If you have any doubt about the order of evaluation for a given expression, group the expression with parentheses. For example, the server evaluates (4 + 3 * 2) as 10 but evaluates the grouped expression ((4 + 3) * 2) as 14.

Usage Notes

This query would normally return long-numeric values for the **Price** column. The STRING scalar function is used to remove all but two of the decimal places from each **Price** value:

```
string(sum(dollars)/sum(quantity), 7, 2) as price
```

For more information about the STRING function and other scalar functions, refer to the *SQL Reference Guide*.

Using the HAVING Clause to Exclude Groups

Question

Which products had total sales of less than $25,000 during 1999? How many individual sales were made?

Example Query

```
select prod_name, sum(dollars) as total_sales,
   sum(quantity) as total_units
from product natural join sales
   natural join period
where year = 1999
group by prod_name
having total_sales < 25000
order by total_sales desc;
```

Result

Prod_Name	Total_Sales	Total_Units
Aroma t-shirt	21397.65	1967
Espresso Machine Royale	18119.80	324
Espresso Machine Italiano	17679.15	177
Coffee Sampler	16634.00	557
Tea Sampler	14907.00	597
Aroma baseball cap	13437.20	1696
Aroma Sheffield Steel Teapot	8082.00	270
Spice Sampler	7788.00	649
Aroma Sounds CD	5937.00	459
Aroma Sounds Cassette	5323.00	630

(1 of 2)

Prod_Name	Total_Sales	Total_Units
French Press, 4-Cup	4570.50	230
Spice Jar	4073.00	227
French Press, 2-Cup	3042.75	205
Travel Mug	1581.75	145
Easter Sampler Basket	1500.00	50
Coffee Mug	1258.00	256
Christmas Sampler	1230.00	41

(2 of 2)

Conditions on Groups: HAVING Clause

Although dividing data into groups reduces the amount of information returned, queries often still return more information than you need. You can use a HAVING clause to exclude groups that fail to satisfy a specified condition, such as sums of dollars that are less than or higher than a given number.

This query calculates the total sales revenue for each product in 1999, then retains only those products whose totals fall below $25,000.

Syntax of the HAVING Clause

```
SELECT select_list
FROM table_list
[WHERE search_condition]
[GROUP BY group_list]
[HAVING condition]
[ORDER BY order_list]
   [BREAK BY order_reference SUMMING
select_reference_list];
```

condition An SQL condition that can include set functions.

The HAVING clause differs from the WHERE clause in the following ways.

WHERE Clause	HAVING Clause
Works on rows of data prior to grouping.	Works on the result set after grouping.
Conditions cannot be expressed with set functions (for example, SUM or AVG), but column aliases for nonaggregate expressions can be used.	Conditions can be expressed with any set function or column alias.

Usage Notes

Any set function can be referenced in a condition in the HAVING clause. A query with a HAVING clause must contain a GROUP BY clause unless the select list contains only set functions. For example:

```
select min(prodkey), max(classkey)
from product
having min(prodkey) = 0;
```

Removing Rows That Contain NULLs, Zeroes, and Spaces

Question

What is the average discount applied to orders received from each Aroma supplier?

Example Query 1

```
select name as supplier,
   dec(sum(discount)/count(order_no),7,2) as avg_deal
from supplier natural join orders
   natural join deal
group by supplier
order by avg_deal desc;
```

Result

Supplier	Avg_Deal
Espresso Express	500.00
Western Emporium	66.66
Aroma West Mfg.	50.00
CB Imports	47.50
Leaves of London	40.00
Tea Makers, Inc.	20.00
Colo Coffee	0.00
Aroma East Mfg.	0.00
Crashing By Design	0.00

Example Query 2

```
select name as supplier,
   dec(sum(discount)/count(order_no),7,2) as avg_deal
from supplier natural join orders
   natural join deal
group by supplier
order by avg_deal desc
suppress by 2;
```

Result

Supplier	Avg_Deal
Espresso Express	500.00
Western Emporium	66.66
Aroma West Mfg.	50.00

(1 of 2)

Supplier	Avg_Deal
CB Imports	47.50
Leaves of London	40.00
Tea Makers, Inc.	20.00

(2 of 2)

Removing Blank Rows: SUPPRESS BY Clause

If one or more columns in the data retrieved by a query contain NULLs, spaces, or zeroes, you can use a SUPPRESS BY clause to remove those rows from the final result set. This clause is a RISQL extension to the ANSI SQL-92 standard.

Syntax of the SUPPRESS BY Clause

```
SELECT select_list
FROM table_list
[WHERE search_condition]
[GROUP BY group_list]
[HAVING condition]
[ORDER BY order_list]
   [BREAK BY order_reference SUMMING select_reference_list]
[SUPPRESS BY column_list];
```

column_list A list of column names or aliases from the *select_list* or a list of positional numbers that specify those columns.

About the Queries

The first example query retrieves a complete list of Aroma suppliers, whether or not they have given discounts on orders. Consequently, the result set lists three suppliers whose average deal amounts to zero dollars.

The second example removes those three suppliers from the result set by suppressing rows that contain 0.00 in column 2 (**Avg_Deal**).

Usage Notes

The DEC scalar function truncates the long-numeric values for the **Avg_Deal** column. Unlike the STRING function, which is described on page 2-42, the DEC function converts the average values to more precise decimal values (not character strings).

The SUPPRESS BY clause is applied before any RISQL display functions in the query are computed. Consequently, you cannot suppress rows by referencing a column that contains a display function. For examples of queries that include display functions, refer to Chapter 3, "Data Analysis."

Summary

The SELECT Statement

```
SELECT select_list
FROM table_list
[WHERE search_condition]
[GROUP BY group_list]
[HAVING search_condition]
[ORDER BY order_list]
   [BREAK BY order_reference SUMMING select_reference_list]
[SUPPRESS BY column_list];
```

Logical Connectives

( )	parentheses (force order of evaluation)
NOT	negation
AND	and
OR	or

Comparison Operators

=	equal
<	less than
>	greater than
<>	not equal
>=	greater than or equal
<=	less than or equal

Comparison Predicates

BETWEEN	*expression1* AND *expression2*
LIKE	pattern
IN	(*list*)
IS	NULL
IS	NOT NULL

This chapter discussed how to express many commonly asked business questions as SELECT statements and how to retrieve, group, and order data selected from one or more relational tables. This chapter also showed how to perform aggregate calculations such as sums, averages, minimums, and maximums, how to calculate subtotals with the BREAK BY clause, and how to use the SUPPRESS BY clause to remove rows that contain zeroes, NULLs, or space characters.

Most questions discussed in this chapter are easily expressed as standard SELECT statements and challenge neither the user nor SQL. The remaining chapters of this guide address more difficult questions, questions that require sequential processing, comparisons of aggregated values, more complex join specifications, or lengthy SELECT statements.

The next chapter shows how you can use RISQL extensions to answer business questions that require sequential processing.

Chapter

Data Analysis

3

In This Chapter

This chapter describes how to write queries that require some kind of data analysis. Many of the queries contain sequential calculations, or calculations that operate on an ordered set of rows—queries frequently encountered during business analysis. For example:

- What is the cumulative total (or running sum) by month?
- What is the moving average by week?
- How do monthly sales figures rank with one another?
- What is the ratio of current month sales to annual sales?

Standard SQL cannot perform these types of calculations, so analysts typically retrieve all the data required for the calculation and then perform sequential calculations with a client tool. This process can be laborious and time consuming. Informix provides *RISQL display functions* as a solution to this problem. With these functions, sequential calculations are performed quickly and easily on the server and only the results you want are returned to your data-analysis application.

This chapter also shows how to use scalar functions to calculate and extract date information from DATETIME columns.

RISQL Display Functions

Function Name and Syntax	Purpose
CUME(*expression*)	Calculate a cumulative sum (running total).
MOVINGAVG(*expression*, *n*)	Calculate an average of the previous *n*-rows.
MOVINGSUM(*expression*, *n*)	Calculate a sum of the previous *n* rows.
NTILE(*expression*, *n*)	Determine an *n*-level rank of values.
RANK(*expression*)	Determine a numeric rank of values.
RATIOTOREPORT(*expression*)	Calculate a ratio of portion to total.
TERTILE(*expression*)	Determine a three-level rank of values (high, middle, and low).

Using RISQL Display Functions

RISQL display functions operate on sets of rows and perform sequential calculations. For example, the function CUME(dollars) returns a cumulative total of dollars for each row of a result table.

Within a SELECT statement, RISQL display functions can be used:

- In the select list
- In an expression
- As arguments of scalar functions
- As a condition in a WHEN clause
- In a subquery

RISQL display functions cannot be used:

- As arguments of set functions
- In the search condition of a WHERE clause

Although these functions are not defined by the ANSI SQL-92 standard, they are valuable because they are efficient, fast, easy to use, and they simplify the expression of commonly asked business questions.

Usage Notes

Most of the RISQL display functions are order-dependent; that is, they operate on an ordered result set. Therefore, queries containing these functions typically contain an ORDER BY clause. RISQL display functions are calculated after the processing of the ORDER BY is complete.

Many of the queries in this chapter rely on *aggregated* sales totals. Because the **Sales** table stores only daily totals, it would be useful to create aggregate tables to answer these queries. For more information about how to accelerate the performance of aggregate queries, refer to the *Informix Vista User's Guide* for instructions on creating and using aggregate tables.

Using the CUME() Function

Question

What were the daily sales figures for Aroma Roma coffee during January, 2000? What were the cumulative subtotals for dollars and quantities during the month?

Example Query

```
select date, sum(dollars) as total_dollars,
   cume(sum(dollars)) as run_dollars,
   sum(quantity) as total_qty,
   cume(sum(quantity)) as run_qty
from period natural join sales
   natural join product
where year = 2000
   and month = 'JAN'
   and prod_name = 'Aroma Roma'
group by date
order by date;
```

Result

Date	Total_Dollars	Run_Dollars	Total_Qty	Run_Qty
2000-01-02	855.50	855.50	118	118
2000-01-03	536.50	1392.00	74	192
2000-01-04	181.25	1573.25	25	217
2000-01-05	362.50	1935.75	50	267
2000-01-06	667.00	2602.75	92	359
2000-01-07	659.75	3262.50	91	450
2000-01-08	309.50	3572.00	54	504
2000-01-09	195.75	3767.75	27	531
2000-01-10	420.50	4188.25	58	589
2000-01-11	547.50	4735.75	78	667
2000-01-12	536.50	5272.25	74	741
2000-01-13	638.00	5910.25	88	829
2000-01-14	1057.50	6967.75	150	979
2000-01-15	884.50	7852.25	122	1101
2000-01-16	761.25	8613.50	105	1206
2000-01-17	455.50	9069.00	66	1272
2000-01-18	768.50	9837.50	106	1378
2000-01-19	746.75	10584.25	103	1481
2000-01-20	261.00	10845.25	36	1517
2000-01-21	630.75	11476.00	87	1604
2000-01-22	813.75	12289.75	115	1719
...				

Cumulative Totals: CUME

The RISQL CUME function calculates and displays running totals. Although a standard SQL SELECT statement can calculate and display sales figures for a given period of time, it cannot calculate a cumulative total on a result set.

CUME Function

To calculate a cumulative total, place a CUME function in the select list for each numeric column to be summed:

```
CUME(expression)
```

expression A column name or a numeric expression.

About the Query

The example query calculates the daily sales figures and quantities for Aroma Roma coffee during January of 2000. The query also calculates and displays cumulative totals for those values.

An ORDER BY clause specifies `mon` to ensure that the result is returned in chronological order.

Usage Notes

Never make assumptions about the order of a result table. If the months are not in chronological order, the running totals you receive might be incorrect. Running totals are sequentially computed subtotals. To obtain the correct running totals, always include an ORDER BY clause in queries that contain RISQL display functions.

The CUME function maintains a running total for a numeric expression that might, but does not have to, contain a column reference. For example:

```
select cume(1) as row_num, order_no, price from orders;
```

Row_Num	Order_No	Price
1	3600	1200.46
2	3601	1535.94
3	3602	780.00
...		

When a column reference occurs in the expression, it must be a numeric column.

When a query contains a GROUP BY clause, each expression in the select list must either reference one of the columns that occurs in the GROUP BY clause or be an SQL set function or RISQL display function.

An ORDER BY clause is included to ensure that the CUME function operates on an ordered set of rows. Columns used in ORDER BY clauses must exist in the GROUP BY clause; therefore, this result set is both grouped and ordered by the **Date** column.

Using CUME with RESET BY

Question

What were the cumulative daily Aroma Roma sales figures during each week of January, 2000?

Example Query

```
select week, date, sum(dollars) as total_dollars,
   cume(sum(dollars)) as run_dollars,
   sum(quantity) as total_qty,
   cume(sum(quantity)) as run_qty
from period natural join sales
   natural join product
where year = 2000
   and month = 'JAN'
   and prod_name = 'Aroma Roma'
group by week, date
order by week, date
   reset by week;
```

Result

Week	Date	Total_Dollars	Run_Dollars	Total_Qty	Run_Qty
2	2000-01-02	855.50	855.50	118	118
2	2000-01-03	536.50	1392.00	74	192
2	2000-01-04	181.25	1573.25	25	217
2	2000-01-05	362.50	1935.75	50	267
2	2000-01-06	667.00	2602.75	92	359
2	2000-01-07	659.75	3262.50	91	450
2	2000-01-08	309.50	3572.00	54	504
3	2000-01-09	195.75	195.75	27	27

(1 of 2)

Week	Date	Total_Dollars	Run_Dollars	Total_Qty	Run_Qty
3	2000-01-10	420.50	616.25	58	85
3	2000-01-11	547.50	1163.75	78	163
3	2000-01-12	536.50	1700.25	74	237
3	2000-01-13	638.00	2338.25	88	325
3	2000-01-14	1057.50	3395.75	150	475
3	2000-01-15	884.50	4280.25	122	597
4	2000-01-16	761.25	761.25	105	105
4	2000-01-17	455.50	1216.75	66	171
4	2000-01-18	768.50	1985.25	106	277
4	2000-01-19	746.75	2732.00	103	380
4	2000-01-20	261.00	2993.00	36	416
4	2000-01-21	630.75	3623.75	87	503
4	2000-01-22	813.75	4437.50	115	618
...					

(2 of 2)

Resetting Cumulative Totals: RESET BY Subclause

You can reset the calculation of cumulative subtotals for multiple columns by including a RESET BY subclause in the ORDER BY clause. This subclause resets running totals to zero whenever the values in the specified columns change. Client tools often refer to such changes as *control breaks* or *programmed breaks*.

RESET BY Subclause

The RESET BY subclause must occur within an ORDER BY clause:

```
SELECT select_list
FROM table_list
[WHERE search_condition]
[GROUP BY group_list]
[HAVING search_condition]
[ORDER BY order_list
   [RESET BY reset_list]]
   [BREAK BY order_reference SUMMING select_reference_list]
[SUPPRESS BY column_list];
```

reset_list One or more columns listed in the select list.

About the Query

The example query calculates running totals for sales of Aroma Roma coffee during January, 2000. The RESET BY subclause resets the running total to zero when the **Week** value changes. To do this, the query must be ordered and grouped by the **Week** column; therefore, the **Week** column occurs in all three clauses: GROUP BY, ORDER BY, and RESET BY.

The blank line in the result set was produced by using the RISQL Reporter-command SET COLUMN *column_name* SKIP LINE.

Usage Notes

Columns referenced in the RESET BY subclause must occur in the select list and the ORDER BY clause. Positional references to columns in the select list can be used in the ORDER BY and RESET BY clauses but not in the GROUP BY clause.

For more information about the ORDER BY clause, refer to page 2-24.

Using the MOVINGAVG() Function

Question

What was the three-week moving average of product sales at San Jose and Miami stores during the third quarter of 1999?

Example Query

```
select city, week, sum(dollars) as sales,
   string(movingavg(sum(dollars), 3), 7, 2) as mov_avg,
   cume(sum(dollars)) as run_sales
from store natural join sales natural join period
where qtr = 'Q3_99' and city in ('San Jose', 'Miami')
group by city, week
order by city, week
   reset by city;
```

Result

City	Week	Sales	Mov_avg	Run_sales
Miami	27	1838.55	NULL	1838.55
Miami	28	4482.15	NULL	6320.70
Miami	29	4616.70	3645.80	10937.40
Miami	30	4570.35	4556.40	15507.75
Miami	31	4681.95	4623.00	20189.70
Miami	32	3004.50	4085.60	23194.20
Miami	33	3915.9	3867.45	27110.10
Miami	34	4119.35	3679.91	31229.45
Miami	35	2558.90	3531.38	33788.35

(1 of 2)

City	Week	Sales	Mov_avg	Run_sales
Miami	36	4556.25	3744.83	38344.60
Miami	37	5648.50	4254.55	43993.10
Miami	38	5500.25	5235.00	49493.35
Miami	39	4891.40	5346.71	54384.75
Miami	40	3693.80	4695.15	58078.55
San Jose	27	3177.55	NULL	3177.55
San Jose	28	5825.80	NULL	9003.35
San Jose	29	8474.80	5826.05	17478.15
San Jose	30	7976.60	7425.73	25454.75
San Jose	31	7328.65	7926.68	32783.40
San Jose	32	6809.75	7371.66	39593.15
San Jose	33	7116.35	7084.91	46709.50
San Jose	34	6512.35	6812.81	53221.85
San Jose	35	6911.50	6846.73	60133.35
San Jose	36	5996.10	6473.31	66129.45
San Jose	37	10000.60	7636.06	76130.05
San Jose	38	7274.70	7757.13	83404.75
San Jose	39	9021.15	8765.48	92425.90
San Jose	40	5045.20	7113.68	97471.10

(2 of 2)

Calculating Moving Averages: MOVINGAVG

Sales figures fluctuate over time; when they fluctuate radically, they obscure underlying, long-range trends. Moving averages are used to smooth the effects of these fluctuations. For example, a three-week moving average divides the sum of the last three consecutive weekly aggregations by three.

MOVINGAVG Function

To calculate a moving average, place a MOVINGAVG function in the select list for each numeric column to be averaged. The function refers to a column or numeric expression to be averaged and an integer representing the number of rows to average:

```
MOVINGAVG(n_expression, n)
```

n_expression	The name of a column that contains numeric data or a numeric expression.
n	An integer that represents a smoothing factor.

About the Query

The example query calculates a three-week average for sales at San Jose and Miami stores during the third quarter of 1999. A control break is triggered by a RESET BY subclause on the **City** column. Three weeks must pass before a three-week moving average can be calculated; consequently, the first two rows following each control break contain NULLs.

The result table must be fully ordered, and the moving average must be reset when the city changes. If the rows are not in chronological order, the moving average function returns incorrect results. Therefore, the ORDER BY clause includes both the **City** column and the **Week** column.

The STRING scalar function is used to truncate the long-numeric values returned for the **Mov_Avg** column. For details, refer to the *SQL Reference Guide*.

As in the previous example, the blank line in the result set was produced by using the RISQL Reporter command SET COLUMN *column_name* SKIP LINE.

Usage Notes

ORDER BY clauses are recommended for all queries that contain order-dependent RISQL display functions.

Using the MOVINGSUM Function

Question

What was the seven-day moving sum of quantities of Demitasse Ms coffee sold during March, 2000?

Example Query

```
select date, sum(quantity) as day_qty,
   string(movingsum(sum(quantity), 7),7,2) as mov_sum
from store natural join sales natural join period
   natural join product
where year = 2000 and month = 'MAR' and prod_name = 'Demitasse Ms'
group by date
order by date;
```

Result

Date	Day_Qty	Mov_Sum
2000-03-01	65	NULL
2000-03-02	19	NULL
2000-03-03	92	NULL
2000-03-04	91	NULL
2000-03-05	106	NULL
2000-03-06	92	NULL
2000-03-07	102	567.00

(1 of 2)

Result

Date	Day_Qty	Mov_Sum
2000-03-08	21	523.00
2000-03-09	74	578.00
2000-03-10	81	567.00
2000-03-11	77	553.00
2000-03-12	127	574.00
2000-03-13	169	651.00
2000-03-14	31	580.00
2000-03-15	56	615.00
2000-03-16	40	581.00
2000-03-17	84	584.00
2000-03-18	34	541.00
2000-03-19	128	542.00
2000-03-20	97	470.00
2000-03-21	50	489.00
2000-03-22	147	580.00
2000-03-23	104	644.00
2000-03-24	48	608.00
2000-03-25	93	667.00
2000-03-26	130	669.00
2000-03-27	95	667.00
2000-03-28	122	739.00
...		

(2 of 2)

Calculating Moving Sums: MOVINGSUM

A moving sum function, like a moving average, is used to smooth the effects of fluctuations. For example, a seven-day moving sum is calculated by summing seven consecutive days.

MOVINGSUM Function

To calculate a moving sum, place a MOVINGSUM function in the select list for each numeric column to be summed. The function requires a column name or numeric expression (*n_expression*) indicating the column to be summed and an integer (*n*) representing the number of rows to sum (a smoothing factor):

```
MOVINGSUM(n_expression, n)
```

About the Query

The example query calculates a seven-day moving sum of quantities of Demitasse Ms coffee sold during March, 2000. The first six rows have NULL entries because seven days must pass before the moving sum can be calculated.

The STRING scalar function is used to truncate the long-numeric values returned for the **Mov_Sum** column. For more information about this function, refer to the *SQL Reference Guide*.

Usage Notes

If the MOVINGSUM function is applied to results that are not in chronological order, the function will return incorrect results; therefore, an ORDER BY clause is recommended. As in the previous example, the **Date** column must be used in both the ORDER BY clause and the GROUP BY clause.

Using the RANK Function

Question

What were the March 1999 rankings of stores in the Western region, in terms of total dollar sales?

Example Query

```
select store_name, district, sum(dollars) as total_sales,
   rank(sum(dollars)) as sales_rank
from market natural join store
   natural join sales
   natural join period
where year = 1999
   and month = 'MAR'
   and region = 'West'
group by store_name, district;
```

Result

Store_Name	District	Total_Sales	Sales_Rank
Cupertino Coffee Supply	San Francisco	18801.50	1
San Jose Roasting Company	San Francisco	18346.90	2
Beaches Brew	Los Angeles	18282.05	3
Java Judy's	Los Angeles	17826.25	4
Instant Coffee	San Francisco	15650.50	5
Roasters, Los Gatos	San Francisco	12694.50	6

Ranking Data: RANK

You can rank any set of values with the RANK function, which assigns 1 to the largest value in a group, 2 to the next largest, and so forth. Magnitude, not order, determines the rank of a value.

RANK Function

To rank a set of values, specify:

```
RANK(expression)
```

In the select list, *expression* is any data type. If *expression* is NULL, RANK returns NULL. For more information about numeric expressions, refer to the *SQL Reference Guide*.

About the Query

The example query ranks stores in the Western region in terms of dollar sales for March 1999. The daily totals from the **Sales** table that meet the search conditions in the WHERE clause are summed, then ranked.

Usage Notes

The GROUP BY clause is required in this example. In a SELECT statement without a GROUP BY clause, where an aggregation function is included in the select list, all column references must be aggregation functions.

For nonnumeric data types, the ranking is based on the collation sequence defined in the Red Brick Decision Server locale specification.

RANK is not an order-dependent display function; by default, queries that contain a RANK function and no ORDER BY clause sort the result set by the ranking values (highest to lowest).

To rank a set of values from bottom to top, reverse the sign of the ranked column with the *unary negation operator*:

```
RANK(-expression)
```

For example:

```
rank(-dollars) as sales_rank
```

Using the RANK, WHEN Function

Question

In the first quarter of 2000 at the Olympic Coffee Company, what were the top 10 days for sales of Breakfast Blend tea? What were the corresponding ranks by quantity?

Example Query

```
select date, day, dollars as day_sales,
   rank(dollars) as sales_rank,
   quantity as day_qty,
   rank(quantity) as qty_rank
from product natural join sales
   natural join period
   natural join store
where qtr = 'Q1_00'
   and prod_name like 'Break%'
   and store_name like 'Olympic%'
when sales_rank <= 10
order by date;
```

Result

Date	Day	Day_Sales	Sales_Rank	Day_Qty	Qty_Rank
2000-01-21	FR	30.00	9	8	9
2000-02-01	TU	56.25	3	15	2
2000-02-08	TU	30.00	9	8	9
2000-02-22	TU	71.25	1	19	1
2000-02-23	WE	41.25	7	11	6
2000-03-03	FR	59.50	2	14	4
2000-03-11	SA	55.25	5	13	5

(1 of 2)

Date	Day	Day_Sales	Sales_Rank	Day_Qty	Qty_Rank
2000-03-16	TH	56.25	3	15	2
2000-03-22	WE	38.25	8	9	8
2000-03-23	TH	42.50	6	10	7

(2 of 2)

Ranking the Top Ten: RANK, WHEN

You can rank any set of values with the RANK function and then specify that only those of interest are displayed. For example, you can rank the sales of all products but display only the top ten, the bottom ten, or any other combination that you can express in a search condition.

WHEN Clause

To restrict the rows returned in a result table after ranked values are calculated (or other display functions or set functions), include a WHEN clause in the query:

```
SELECT select_list
FROM table_name
[WHERE search_condition]
[GROUP BY group_list]
[HAVING search_condition]
[WHEN condition]
[ORDER BY order_list]
   [RESET BY reset_list]
   [BREAK BY order_reference SUMMING select_reference_list]
[SUPPRESS BY column_list];
```

Compound conditions constructed with the AND, OR, and NOT logical connectives are allowed in the WHEN clause. For more information about conditions, refer to the *SQL Reference Guide*.

About the Query

The example query ranks daily sales of Breakfast Blend tea at a single store in a single quarter but returns figures only for the top 10 days. The query also returns the corresponding quantity rankings. The ORDER BY clause sorts the result set in chronological order (by the values in the **Date** column).

When values to be ranked are equal, they are assigned the same ranking value. For example, two rows receive a sales rank of 3 in this case.

Using the NTILE Function

Question

Which products rank in the top 25 percent and bottom 25 percent based on annual sales totals for 1999?

Example Query

```
select prod_name, sum(dollars) as total_sales,
   ntile(total_sales, 4) as sales_rank
from sales natural join product
   natural join period
where year = 1999
group by prod_name
when sales_rank in (1, 4);
```

Result

Prod_name	Total_sales	Sales_rank
Demitasse Ms	304727.00	1
Xalapa Lapa	263353.00	1
NA Lite	262162.00	1
Lotta Latte	251713.00	1
Cafe Au Lait	251086.50	1
Expresso XO	229201.25	1
Veracruzano	227769.50	1
La Antigua	223528.25	1
Aroma Roma	218574.75	1
Colombiano	218009.50	1

(1 of 2)

Prod_name	Total_sales	Sales_rank
Aroma Sounds CD	5937.00	4
Aroma Sounds Cassette	5323.00	4
French Press, 4-Cup	4570.50	4
Spice Jar	4073.00	4
French Press, 2-Cup	3042.75	4
Travel Mug	1581.75	4
Easter Sampler Basket	1500.00	4
Coffee Mug	1258.00	4
Christmas Sampler	1230.00	4

(2 of 2)

Ranking Values in Groups: NTILE

You can rank each value in a group of numeric values as 1 (highest) through any specified number (lowest) by using the NTILE function. This function assigns the appropriate rank to a value depending on its magnitude relative to other values in a group.

Syntax

To rank a set of values into 100 equal groups, include the NTILE function in the select list and specify the numeric expression or column to be ranked, followed by the number 100:

```
NTILE(expression, 100)
```

If *expression* is NULL, NTILE returns NULL. For more information about numeric expressions, refer to the *SQL Reference Guide*.

About the Query

The example query ranks products as 1, 2, 3, or 4, based on annual sales totals for 1999. The WHEN clause removes the middle 50 percent (2 and 3) from the result set.

Usage Notes

In those cases where equal values span a boundary, they are distributed between adjacent groups.

If the set of values is not divisible by the specified number, the NTILE function puts leftover rows in the higher-level group.

Important: *By using the NTILE function inside a CASE expression, you can redistribute ranked values into unequal groups and replace the default NTILE numeric values with more meaningful labels. For an example, refer to* *"Using the NTILE Function with a CASE Expression."*

Using the NTILE Function with a CASE Expression

Question

What products fell into the top 20 percent, middle 60 percent, and bottom 20 percent of sales totals for the second week of 1998, at stores in the Western region?

Example Query

```
select prod_name, sum(quantity) as quantity,
   sum(dollars) as sales,
   case ntile(sum(dollars), 5)
      when 1 then 'TOP_20'
      when 2 then 'MID_60'
      when 3 then 'MID_60'
      when 4 then 'MID_60'
      when 5 then 'LOW_20' end as grp
from market natural join store natural join sales
   natural join period natural join product
where year = 1998 and week = 2 and region = 'West'
group by prod_name;
```

Result

Prod_Name	Qty	Sales	Grp
Expresso XO	368	2887.00	TOP_20
Aroma Roma	246	1783.50	TOP_20
Colombiano	257	1757.75	TOP_20
Darjeeling Special	143	1655.00	TOP_20
Lotta Latte	198	1621.00	TOP_20
La Antigua	213	1589.25	TOP_20
Demitasse Ms	151	1503.75	MID_60
Xalapa Lapa	163	1395.50	MID_60
Ruby’s Allspice	183	1018.50	MID_60
Veracruzano	120	925.50	MID_60
NA Lite	100	900.00	MID_60
Cafe Au Lait	106	869.50	MID_60
Assam Gold Blend	104	636.50	MID_60
English Breakfast	137	561.50	MID_60

(1 of 2)

Prod_Name	Qty	Sales	Grp
Aroma t-shirt	44	481.80	MID_60
Coffee Sampler	16	480.00	MID_60
Assam Grade A	114	380.00	MID_60
Darjeeling Number 1	69	378.25	MID_60
Irish Breakfast	81	345.75	MID_60
Breakfast Blend	82	322.00	MID_60
Gold Tips	91	320.75	MID_60
Earl Grey	80	305.00	MID_60
Special Tips	74	253.50	MID_60
Aroma Sheffield Steel Teapot	7	210.00	LOW_20
Espresso Machine Italiano	1	99.95	LOW_20
Aroma baseball cap	11	87.45	LOW_20
Spice Sampler	4	48.00	LOW_20
Travel Mug	1	10.95	LOW_20

(2 of 2)

Ranking Values in Unequal Groups: CASE and NTILE

The NTILE function can combine powerfully with a CASE expression to rank and redistribute a set of values. For example, NTILE might be used to rank values into five equal groups; then a CASE expression could be used to redistribute those values into unequal groups, representing a curve.

CASE Syntax

A CASE expression is a conditional scalar expression that can be used in the select list to substitute a specified column value for another value:

```
CASE expression WHEN result THEN result1 ELSE result2
   END AS col_alias
```

expression Any valid expression.

result A value to which the expression is expected to evaluate.

result1 A value that substitutes for *result*.

result2 A default value that is used, if specified, when *expression* does not evaluate to *result*.

Typically, multiple WHEN...THEN conditions are used to find and replace several different values.

Important: *A CASE expression can take one of two forms: simple or searched. This example uses the simple form. For more details, refer to the "SQL Reference Guide." For an example of the searched form, see "Using CASE Expressions" on page 4-6.*

About the Query

The example query uses the NTILE function inside a CASE expression to spread the tiled values (fifths) into three unequal groups: the top 20 percent is represented in the final result set as `TOP_20`, the middle 60 percent as `MID_60`, and the bottom 20 percent as `LOW_20`. When the following expression evaluates to 1, that value is substituted with the character string TOP_20:

```
ntile(sum(dollars), 5)
```

When the expression evaluates to 2, 3, or 4, those values are replaced with `MID_60`, and when it evaluates to 5, it is replaced with `LOW_20`.

Usage Notes

By adding a WHEN clause to the end of this query, you could eliminate a specified portion of the result set. For example:

```
when grp = 'MID_60'
```

Using the TERTILE Function

Question

Which cities ranked high, middle, and low in the West and South in 1999, in terms of unit sales of Earl Grey tea?

Example Query

```
select city, sum(quantity) as qty_1999,
   tertile(sum(quantity)) as q_rk
from market natural join store
   natural join sales
   natural join product
   natural join period
where year = 1999
   and prod_name like 'Earl Grey%'
   and region in ('West', 'South')
group by city;
```

Result

City	Qty_1999	Q_RK
San Jose	1469	H
Los Angeles	911	H
Phoenix	814	H
Los Gatos	805	M
Miami	782	M
Cupertino	778	M
Houston	768	L
New Orleans	684	L
Atlanta	614	L

Ranking Values as High, Middle, or Low: TERTILE

You can rank each value in a group of numeric values as High, Middle, or Low with the TERTILE function. This function assigns the letter *H*, *M*, or *L* to a value depending on its magnitude relative to other values in a group.

Syntax

To rank a group into thirds, include the TERTILE function in the select list and specify the numeric expression or column to be ranked:

```
TERTILE(expression)
```

expression A column name or a numeric expression.

If *expression* is NULL, TERTILE returns NULL. For more information about numeric expressions, refer to the *SQL Reference Guide*.

About the Query

The example query ranks cities according to quantities of Earl Grey tea sold during 1999. The result table is divided into city groups and the **Quantity** column is summed for each city for the year.

Usage Notes

Values do not always fall nicely into three sets. When a set of values is not divisible by three, the TERTILE function puts any leftover rows in the higher-level group. In those cases where equal values span a boundary, they are distributed between adjacent groups.

Even though the column referenced by the TERTILE function must be numeric, the result of this function is always a character column.

For more information about the TERTILE function and data types, refer to the *SQL Reference Guide*.

Using the RATIOTOREPORT Function

Question

What was the ratio of monthly sales to total sales of Xalapa Lapa coffee in San Jose and Los Angeles stores during the third quarter of 1999?

Example Query

```
select city, month, sum(dollars) as total_sales,
   ratiotoreport(sum(dollars))*100 as pct_of_sales
from store natural join sales
   natural join product
   natural join period
where prod_name like 'Xalapa%'
   and qtr = 'Q3_99'
   and city in ('San Jose', 'Los Angeles')
group by city, month;
```

Result

City	Month	Total_Sales	Pct_of_Sales
San Jose	JUL	2499.50	26.99
Los Angeles	JUL	1627.00	17.57
San Jose	AUG	1004.00	10.84
Los Angeles	AUG	995.00	10.74
San Jose	SEP	1802.00	19.46
Los Angeles	SEP	1334.00	14.40

Calculating Ratios as Percentages: RATIOTOREPORT*100

The RATIOTOREPORT function calculates the ratio of a numeric row value to the total value of that column in the result set. For example, if a given column lists sales figures for different products, each value in that column can be expressed as a ratio of the total sales for all the products listed.

Syntax

To calculate a ratio of a column value to the sum of all the values in the column, include the RATIOTOREPORT function in the select list and specify a numeric expression or the name of a column that contains numeric values:

```
RATIOTOREPORT(expression)
```

If *expression* is NULL, RATIOTOREPORT returns NULL. For more information about numeric expressions, refer to the *SQL Reference Guide*.

To calculate ratios as percentages, simply use the following notation after the expression:

```
*100
```

About the Query

The example query displays the ratio of monthly sales of Xalapa Lapa coffee in San Jose and Los Angeles stores during the third quarter of 1999 to the total sales of that product in those stores during the same period.

The following expression returns the results of the RATIOTOREPORT function as percentages:

```
(sum(dollars))*100
```

The values in the **Pct_of_Sales** column add up to exactly 100.

Usage Notes

The RATIOTOREPORT function can be reset for groups of values with the RESET BY subclause of the ORDER BY clause. For information about RESET BY, refer to page 3-11.

Using the DATEADD Function

Question

Calculate a date 90 days prior to and 90 days after a given date.

Example Query

```
select dateadd(day, -90, date) as due_date,
   date as cur_date,
   dateadd(day, 90, date) as past_due
from period
where year = 2000
   and month = 'JAN';
```

Result

Due_Date	Cur_Date	Past_Due
1999-10-03	2000-01-01	2000-03-31
1999-10-04	2000-01-02	2000-04-01
1999-10-05	2000-01-03	2000-04-02
1999-10-06	2000-01-04	2000-04-03
1999-10-07	2000-01-05	2000-04-04
1999-10-08	2000-01-06	2000-04-05
1999-10-09	2000-01-07	2000-04-06
1999-10-10	2000-01-08	2000-04-07
1999-10-11	2000-01-09	2000-04-08
1999-10-12	2000-01-10	2000-04-09
1999-10-13	2000-01-11	2000-04-10
1999-10-14	2000-01-12	2000-04-11
1999-10-15	2000-01-13	2000-04-12
1999-10-16	2000-01-14	2000-04-13
1999-10-17	2000-01-15	2000-04-14
1999-10-18	2000-01-16	2000-04-15
1999-10-19	2000-01-17	2000-04-16
1999-10-20	2000-01-18	2000-04-17
1999-10-21	2000-01-19	2000-04-18
1999-10-22	2000-01-20	2000-04-19
1999-10-23	2000-01-21	2000-04-20
1999-10-24	2000-01-22	2000-04-21

(1 of 2)

Due_Date	Cur_Date	Past_Due
1999-10-25	2000-01-23	2000-04-22
1999-10-26	2000-01-24	2000-04-23
1999-10-27	2000-01-25	2000-04-24
1999-10-28	2000-01-26	2000-04-25
1999-10-29	2000-01-27	2000-04-26
1999-10-30	2000-01-28	2000-04-27
1999-10-31	2000-01-29	2000-04-28
1999-11-01	2000-01-30	2000-04-29
1999-11-02	2000-01-31	2000-04-30

(2 of 2)

Incrementing or Decrementing Dates: DATEADD

The DATEADD function returns a datetime value calculated from three arguments:

- Datepart that specifies an increment measure such as *day*, *month*, or *year*
- Positive or negative increment value
- Value to be incremented or decremented (column name or datetime expression)

Function	Returns
DATEADD(*day*, 90, '07-01-99')	1999-09-29
DATEADD(*month*, 3, '07-01-99')	1999-10-01
DATEADD(*year*, 1, '07-01-99’)	2000-07-01

About the Query

The example query calculates a date 90 days before and 90 days after a given date. The DATEADD function returns the value in the ANSI SQL-92 datetime format.

You can also reformat the DATETIME value as a month name using the DATENAME function. The following query uses the DATENAME function in its WHERE clause:

```
select datename(month, dateadd(day, -90, date)) as prior,
   datename(month, date) as cur,
   datename(month, dateadd(day, 90, date)) as next
from period
where datename(yy, date) = '2000'
   and month = 'JAN';
```

Prior	Cur	Next
October	January	March
October	January	April
October	January	April
October	January	April
October	January	April
October	January	April
...		

Because there are 29 days in February of 2000, January 1 plus 90 days is March 31; therefore, the first row in the Next column is March. For more information about DATETIME functions, refer to the *SQL Reference Guide*.

Using the DATEDIFF Function

Question

How long did the storewide Christmas special promotion run in 1999?

Example Query

```
select promo_desc, year,
   datediff(day, end_date, start_date)+1 as days_on_promo
from promotion p, period d
where p.start_date = d.date
   and promo_desc like 'Christmas%'
   and year = 1999;
```

Result

Promo_Desc	Year	Days_on_Promo
Christmas special	1999	31

Calculating Elapsed Days: DATEDIFF

The DATEDIFF function returns a datetime value calculated from three arguments:

- Datepart that specifies the increment measure such as *day*, *month*, or *year*
- Two datetime expressions, which must be DATE, TIME, or TIMESTAMP data types

Function	Returns
DATEDIFF(*day*, '07-01-00','01-01-00')	182
DATEDIFF(*month*, '07-01-00','01-01-00')	6
DATEDIFF(*quarter*, '07-01-00','01-01-00')	2

About the Query

The example query calculates the number of days elapsed between the start and finish of a storewide promotion. To return the result, the following DATEDIFF function operates on the datetime values in the **Promotion** table:

```
datediff(day, end_date, start_date)+1
```

The `+1` is required because the *difference* between the `End_Date` and `Start_Date` values is equal to 30 days, whereas the duration of the promotion includes both the `Start_Date` and the `End_Date` (31 days).

Usage Notes

This query is also an example of a join between two tables that do not have a primary-key to foreign-key relationship; the joining columns simply have comparable datetime data types:

```
where p.start_date = d.date
```

Any two tables, including system tables, can be joined over comparable columns.

The purpose of the join in this case is to return the `Year` value from the **Period** table; alternatively, this value could be extracted from the datetime columns in the **Promotion** table.

Using the EXTRACT Function

Question

What were the day and month names and numbers for the first six weeks of 1998 as extracted from the datetime values in the **Period** table?

Example Query

```
select datename(weekday, date) as day_name,
   extract(weekday from date) as day_num,
   extract(day from date) as day,
   extract(dayofyear from date) as day_yr,
   datename(month, date) as mo_name,
   extract(month from date) as mo_num
from period
where extract(year from date) = 1998
   and extract(week from date) < 7;
```

Result

Day_Name	Day_Num	Day	Day_Yr	Mo_Name	Mo_Num
Thursday	5	1	1	January	1
Friday	6	2	2	January	1
Saturday	7	3	3	January	1
Sunday	1	4	4	January	1
Monday	2	5	5	January	1

(1 of 2)

Day_Name	Day_Num	Day	Day_Yr	Mo_Name	Mo_Num
Tuesday	3	6	6	January	1
Wednesday	4	7	7	January	1
Thursday	5	8	8	January	1
Friday	6	9	9	January	1
Saturday	7	10	10	January	1
Sunday	1	11	11	January	1
Monday	2	12	12	January	1
Tuesday	3	13	13	January	1
Wednesday	4	14	14	January	1
Thursday	5	15	15	January	1
Friday	6	16	16	January	1
Saturday	7	17	17	January	1
Sunday	1	18	18	January	1
Monday	2	19	19	January	1
Tuesday	3	20	20	January	1
Wednesday	4	21	21	January	1
Thursday	5	22	22	January	1
Friday	6	23	23	January	1
Saturday	7	24	24	January	1
Sunday	1	25	25	January	1
Monday	2	26	26	January	1
Tuesday	3	27	27	January	1

(2 of 2)

Displaying Dateparts as Integers: EXTRACT

The EXTRACT function returns an integer value representing a part of a DATETIME value. The function requires two arguments:

- Datepart, which specifies the increment measure such as *day*, *month*, or *year*
- Datetime expression (column name or DATETIME expression)

Function	Returns
extract(weekday from date_col)	Weekday as an integer value from the set (1, 2, …, 7)
extract(day from date_col)	Day of month as an integer from the set (1, 2, …, 31)

About the Query

The example query uses the DATENAME and EXTRACT scalar functions to return day and month names and day and month numbers for the first six weeks of 1998.

Except for the first week of the year, weeks typically begin on Sunday or Monday, depending on the territory specified in the Red Brick Decision Server locale. For information about locale specifications, refer to the *Administrator's Guide* and the *Installation and Configuration Guide*.

Summary

RISQL Display Functions

Function	Result
CUME(*expression*)	Cumulative sum
MOVINGAVG(*expression*, *n*)	Average of the previous *n* rows
MOVINGSUM(*expression*, *n*)	Sum of the previous *n* rows
NTILE(*expression*, *n*)	*n*-level rank of values
RANK(*expression*)	Numeric rank of values
TERTILE(*expression*)	Three-level (high, medium, and low) rank of values
RATIOTOREPORT(*expression*)	Ratio of portion to total

In all of the preceding functions except RANK, the *expression* argument must be a numeric expression or a numeric column name. The expression argument for the RANK function can be of any data type.

CASE Expressions

CASE expressions in the select list are useful for substituting column values with other specified values, such as meaningful character strings to replace numeric values returned by display functions.

For another use of the CASE expression, see "Using CASE Expressions" on page 4-6.

DATETIME Functions

Function	Action
DATEADD	Adds interval to datetime value.
DATEDIFF	Subtracts the difference between two datetime values.
DATENAME	Extracts datepart component from datetime value as character string.
EXTRACT	Extracts datepart from datetime value as integer.

This chapter described how to:

- Use RISQL display functions to perform data analysis, such as calculating ranks, moving averages, and cumulative sums
- Redistribute ranked values into unequal groups and give meaningful labels to those values with an NTILE calculation inside a CASE expression
- Use DATETIME scalar functions to calculate and extract date information from DATETIME columns

Chapter

Comparison Queries

4

In This Chapter

This chapter focuses on queries that compare data. The chapter begins by illustrating the problem that confronts the query writer: how to use SQL to return a spreadsheet or "cross-tab" report rather than a standard, vertically ordered result set that is hard to read. The problem is solved by using either CASE expressions or subqueries.

The CASE solution is presented first, as a simple and concise way of comparing like groups of values. Then several examples of FROM clause and select-list subqueries are presented. These subqueries have the added value of being able to both compare data from different groups and include calculations against the compared values, such as share percentages over given time periods.

This chapter describes subqueries stated as conditions in the WHERE clause, which are useful for simpler comparison queries. The last sections also describe the ALL, EXISTS, and SOME or ANY predicates, which can be used to express conditions on subquery results.

Comparing Data with SQL

Question

How did sales of packaged coffee compare at stores in the western region in 1998?

Query

```
select store_name, prod_name, sum(dollars) as sales
   from market natural join store
      natural join sales
      natural join period
      natural join product
      natural join class
where region like 'West%'
   and year = 1998
   and class_type = 'Pkg_coffee'
group by store_name, prod_name
order by store_name, prod_name;
```

Result

Store_Name	Prod_Name	Sales
Beaches Brew	Aroma Roma	3483.50
Beaches Brew	Cafe Au Lait	3129.50
Beaches Brew	Colombiano	2298.25
Beaches Brew	Demitasse Ms	4529.25
Beaches Brew	Expresso XO	4132.75
Beaches Brew	La Antigua	4219.75
Beaches Brew	Lotta Latte	3468.00
Beaches Brew	NA Lite	4771.00

(1 of 2)

Store_Name	Prod_Name	Sales
Beaches Brew	Veracruzano	4443.00
Beaches Brew	Xalapa Lapa	4304.00
Cupertino Coffee Supply	Aroma Roma	4491.00
Cupertino Coffee Supply	Cafe Au Lait	4375.50
Cupertino Coffee Supply	Colombiano	2653.50
Cupertino Coffee Supply	Demitasse Ms	3936.50
Cupertino Coffee Supply	Expresso XO	4689.25
Cupertino Coffee Supply	La Antigua	2932.00
Cupertino Coffee Supply	Lotta Latte	5146.00
Cupertino Coffee Supply	NA Lite	4026.00
Cupertino Coffee Supply	Veracruzano	3285.00
Cupertino Coffee Supply	Xalapa Lapa	5784.00
Instant Coffee	Aroma Roma	3485.25
Instant Coffee	Cafe Au Lait	3599.50
Instant Coffee	Colombiano	3321.75
Instant Coffee	Demitasse Ms	5422.25
Instant Coffee	Expresso XO	2851.00
Instant Coffee	La Antigua	2937.25
Instant Coffee	Lotta Latte	4783.50
Instant Coffee	NA Lite	3740.00
Instant Coffee	Veracruzano	4712.00
Instant Coffee	Xalapa Lapa	3698.00
...		

(2 of 2)

A Simple Comparison Query

You can list the sales of a group of products at specific stores using a simple SELECT statement, but the format of the result table makes the values difficult to compare. For example, the preceding partial result set shows that La Antigua coffee was sold at several stores in the western region, but these figures are hard to isolate.

This kind of data is much easier to compare when it is formatted like a spreadsheet. There are two ways to produce a spreadsheet, or "cross-tab," report: by using CASE expressions or subqueries. The following examples in this chapter illustrate both methods of writing comparison queries.

About the Query

The example query returns 1998 sales figures for packaged coffee products sold at each store in the western region, but the format of the output data makes it difficult to compare the figures product by product, store by store.

Using CASE Expressions

Question

How did sales of packaged coffee compare at stores in the western region in 1998?

Example Query

```
select prod_name,
   sum(case when store_name = 'Beaches Brew'
      then dollars else 0 end) as Beaches,
   sum(case when store_name = 'Cupertino Coffee Supply'
      then dollars else 0 end) as Cupertino,
   sum(case when store_name = 'Roasters, Los Gatos'
      then dollars else 0 end) as RoastLG,
   sum(case when store_name = 'San Jose Roasting Company'
      then dollars else 0 end) as SJRoastCo,
   sum(case when store_name = 'Java Judy''s'
      then dollars else 0 end) as JavaJudy,
```

```
      sum(case when store_name = 'Instant Coffee'
         then dollars else 0 end) as Instant
   from market natural join store
      natural join sales
      natural join period
      natural join product
      natural join class
   where region like 'West%'
      and year = 1998
      and class_type = 'Pkg_coffee'
   group by prod_name
   order by prod_name;
```

Result

Prod_Name	Beaches	Cupertino	RoastLG	SJRoastCo	JavaJudy	Instant
Aroma Roma	3483.50	4491.00	4602.00	4399.25	3748.25	3485.25
Cafe Au Lait	3129.50	4375.50	4199.00	3620.00	4864.50	3599.50
Colombiano	2298.25	2653.50	4205.00	3530.75	3509.00	3321.75
Demitasse Ms	4529.25	3936.50	4347.75	5699.00	6395.25	5422.25
Expresso XO	4132.75	4689.25	4234.50	3811.00	5012.25	2851.00
La Antigua	4219.75	2932.00	3447.50	4323.00	2410.25	2937.25
Lotta Latte	3468.00	5146.00	4469.50	5103.50	4003.00	4783.50
NA Lite	4771.00	4026.00	3250.00	2736.00	4791.00	3740.00
Veracruzano	4443.00	3285.00	4467.00	3856.00	4510.00	4712.00
Xalapa Lapa	4304.00	5784.00	3906.00	3645.00	3182.00	3698.00

A Solution for Comparing Data: CASE Expressions

An efficient and concise way to display compared values in a readable spreadsheet format is to use CASE expressions in the select list. Each CASE operation evaluates a specified expression and supplies a different value depending on whether a certain condition is met.

CASE Syntax

In general, you construct a CASE comparison query by specifying the constraints for the entire domain over which results are to be produced in the WHERE clause of the *main*, or *outer*, query. Then you break the result into subsets with a CASE expression in the select list:

```
CASE WHEN search_condition THEN result1 ELSE result2
   END AS col_alias
```

search_condition	A logical condition that evaluates to true or false.
result1	A value used when *search_condition* is true.
result2	A default value when *search_condition* is false.

Important: *A CASE expression can take one of two forms: simple or searched. This example uses the searched form. For more details, refer to the "SQL Reference Guide." For an example of the simple form, see "Using the NTILE Function with a CASE Expression" on page 3-24.*

About the Query

This query poses the same business question as the previous query in this chapter. In this case, however, the CASE expression is used to produce six different columns in the result set that contain aggregate dollar values, one column for each store.

Usage Notes

In the WHEN condition for the store named *Java Judy's*, the apostrophe must be expressed as two single quotes:

```
when store_name = 'Java Judy''s'
```

Otherwise, the apostrophe will be interpreted as the closing quote for the character string, and the query will return an "incomplete string" error.

Using Subqueries in the FROM Clause

Question

How did product sales in San Jose during January 1998 compare with annual product sales in the same city during the same year?

Example Query

```
select product, jan_98_sales, total_98_sales
from
      (select p1.prod_name, sum(dollars)
      from product p1 natural join sales s1
         natural join period d1 natural join store r1
      where d1.year = 1998 and month = 'JAN'
         and r1.city like 'San J%'
      group by p1.prod_name) as sales1(product, jan_98_sales)

   natural join

      (select p2.prod_name, sum(dollars) as total_98_sales
      from product p2 natural join sales s2
         natural join period d2 natural join store r2
      where d2.year = 1998 and r2.city like 'San J%'
      group by p2.prod_name) as sales2(product, total_98_sales)

order by product;
```

Result

Product	Jan_98_Sales	Total_98_Sales
Aroma Roma	1653.00	21697.50
Aroma Sheffield Steel Teapot	120.00	1122.00
Aroma Sounds Cassette	58.50	866.00
Aroma baseball cap	7.95	2960.15
Aroma t-shirt	470.85	4470.50
Assam Gold Blend	652.00	11375.00
Assam Grade A	352.00	5429.00
Breakfast Blend	608.25	6394.75
Cafe Au Lait	1936.50	24050.50
Colombiano	2148.00	22528.50
Darjeeling Number 1	867.50	8590.00
Darjeeling Special	1355.00	17787.50
Demitasse Ms	2163.00	35523.50
Earl Grey	540.50	6608.50
English Breakfast	393.00	5365.50
Espresso Machine Italiano	899.55	4397.80
Expresso XO	2935.50	27362.00
French Press, 2-Cup	104.65	1196.00
French Press, 4-Cup	19.95	1109.20
Gold Tips	440.00	5381.50
Irish Breakfast	703.25	7455.50
...		

A More Flexible Solution: Subqueries in the FROM Clause

A subquery is any query expression enclosed in parentheses that occurs inside another query. A subquery is sometimes referred to as an *inner query* that operates within an *outer query*, or as the *child query* of a *parent query*.

About the Query

A value is often compared with a sum of a set of values. The example query compares product sales in San Jose in *January* 1998 with product sales in San Jose *throughout* 1998. This kind of query requires *mixed aggregations*; therefore, it cannot be written with CASE expressions, which must operate on values within a single group or scope. Instead, subqueries in the FROM clause are used to make the comparison.

Important: *Any query that can be expressed as a subquery in the FROM clause can also be expressed as a subquery in the select list, as shown later in this chapter. However, subqueries in the FROM clause generally run faster and are conceptually easier to write.*

Usage Notes

The example query relies on the flexibility of the *query expression* in standard SQL to join the results of two subqueries. For detailed information about query expressions, refer to the *SQL Reference Guide*.

Tables derived from the evaluation of subqueries can be joined with other table references. To this end, a subquery in the FROM clause must have a correlation name; however, the list of derived columns is optional. For example, the subqueries in this example evaluate to the following tables:

```
sales1(product, jan_98_sales)
sales2(product, total_98_sales)
```

The natural join of these tables (over the **Product** column) produces an unnamed derived table with three columns, the source of the three select-list items in the main query:

```
product, jan_98_sales, total_98_sales
```

For more examples of table joins, refer to Chapter 5, "Joins and Unions."

Performing Calculations and Comparisons

Question

What percentage of annual product sales in San Jose did the January 1998 sales figures in that city represent? What were the top ten products in terms of those percentages?

Example Query

```
select product, jan_98_sales, total_98_sales,
   dec(100 * jan_98_sales/total_98_sales,7,2) as pct_of_98,
   rank(pct_of_98) as rank_pct
from
      (select p1.prod_name, sum(dollars)
      from product p1 natural join sales s1
         natural join period d1 natural join store r1
      where d1.year = 1998 and month = 'JAN'
         and r1.city like 'San J%'
      group by p1.prod_name) as sales1(product, jan_98_sales)

   natural join

      (select p2.prod_name, sum(dollars)
      from product p2 natural join sales s2
         natural join period d2 natural join store r2
      where d2.year = 1998
         and r2.city like 'San J%'
      group by p2.prod_name) as sales2(product, total_98_sales)

when rank_pct <= 10
order by product;
```

Result

Product	Jan_98_Sales	Total_98_Sales	Pct_of_98	Rank_Pct
Aroma Sheffield Steel Teapot	120.00	1122.00	10.69	4
Aroma t-shirt	470.85	4470.50	10.53	5

(1 of 2)

Product	Jan_98_Sales	Total_98_Sales	Pct_of_98	Rank_Pct
Breakfast Blend	608.25	6394.75	9.51	9
Colombiano	2148.00	22528.50	9.53	8
Darjeeling Number 1	867.50	8590.00	10.09	7
Espresso Machine Italiano	899.55	4397.80	20.45	1
Expresso XO	2935.50	27362.00	10.72	3
Irish Breakfast	703.25	7455.50	9.43	10
La Antigua	2643.25	22244.50	11.88	2
Lotta Latte	3195.00	31200.00	10.24	6

(2 of 2)

Calculations with FROM Clause Subqueries

The result set of a comparison query can be used as the source data for various calculations. For example, a monthly total for a product can be expressed as a *share* of annual sales with a simple percentage calculation:

```
100 * monthly_sales / annual_sales
```

Simple and complex market, product, and time interval shares or percentages can be calculated with subqueries in the FROM clause.

About the Query

Based on the previous example, this query calculates the monthly sales figures for each product in San Jose as a share or percentage of annual sales for that product in the same city. Using the RANK display function (introduced in Chapter 3), the query also ranks the percentage values and discards all but the top ten products from the result set.

The figures in the **Pct_of_98** column do not add up to 100 because these figures represent percentages of one month to a year for individual product sales, not percentages of monthly sales to all annual sales.

Usage Notes

The select list of the main query consists entirely of derived column names, column aliases, and / or expressions that include those names and aliases. For example, the following select-list item uses columns named in the table that derives from the natural join of the subqueries as the operands of the multiplication (*) and division (/) calculations:

```
dec(100 * jan_98_sales/total_98_sales,7,2) as pct_of_98
```

In turn, the final select-list item uses the column alias from the previous expression as the argument of the RANK function:

```
rank(pct_of_98) as rank_pct
```

For more examples of RISQL display functions and the use of the WHEN clause, refer to Chapter 3, "Data Analysis."

Queries that calculate various percentages and performance metrics might require numerous lines of repetitive instructions. For information on how to abbreviate and generalize long SQL statements with RISQL macros, refer to Chapter 6, "Macros, Views, and Temporary Tables."

Using Subqueries in the Select List

Question

During which days of December 1999 were Lotta Latte sales figures at the San Jose Roasting Company lower than the average daily sales figure for the same product at the same store during December 1998?

Display the daily average for 1998 as a separate column.

Example Query

```
select prod_name, store_name, date, dollars as sales_99,
   (select dec(avg(dollars),7,2)
   from store natural join sales
      natural join product
      natural join period
   where year = 1998
      and month = 'DEC'
      and store_name = 'San Jose Roasting Company'
      and prod_name like 'Lotta%') as avg_98
from store natural join sales
      natural join product
      natural join period
where prod_name like 'Lotta%'
   and store_name = 'San Jose Roasting Company'
   and year = 1999
   and month = 'DEC'
   and dollars <
      (select avg(dollars)
      from store natural join sales
         natural join product
         natural join period
      where year = 1998
         and month = 'DEC'
         and store_name = 'San Jose Roasting Company'
         and prod_name like 'Lotta%');
```

Result

Prod_Name	Store_Name	Date	Sales_99	Avg_98
Lotta Latte	San Jose Roasting Company	1999-12-09	153.00	154.72
Lotta Latte	San Jose Roasting Company	1999-12-28	144.50	154.72

Comparisons with Select-List Subqueries

A subquery can occur in the select list of a main query only if it returns one row or no rows. This kind of subquery, a *scalar subquery,* is useful for spreadsheet-style comparisons in which a series of values returned by the main query is compared to a single value returned by the subquery.

About the Query

This example subquery returns the daily Lotta Latte sales figures at the San Jose Roasting Company in 1999 in cases where those figures were lower than the average daily sales figure at the same store during 1998. The **Avg_98** column contains a single, repeated value that represents the 1998 average; this same value would appear in that column regardless of the number of rows in the result set.

The same subquery occurs twice in the main query:

- Once as a column definition in the select list
- Once as an operand of the less-than operator (<) in a WHERE clause condition

This query is processed in the following order:

1. The second subquery, which defines the search condition in the WHERE clause of the main query, is executed.
2. The value derived from the second subquery is inserted into the WHERE clause of the main query.
3. The select-list subquery is executed.
4. The main query is executed.

Usage Notes

The DEC scalar function is used on the **Avg_98** column of the result set to truncate the average sales figures:

```
dec(avg(dollars),7,2)
```

Using Correlated Subqueries

Question

How did individual product sales in San Jose during January 1998 compare with annual sales in the same city during the same year?

Example Query

```
select p1.prod_name, sum(s1.dollars) as jan_98_sales,

   (select sum(s2.dollars)
   from store r2 natural join sales s2
      natural join product p2 natural join period d2
      where p1.prod_name = p2.prod_name
         and d1.year = d2.year
         and r1.city = r2.city) as total_98_sales

from store r1 natural join sales s1
   natural join product p1
   natural join period d1
where year = 1998 and month = 'JAN'
   and city like 'San J%'
group by p1.prod_name, d1.year, r1.city
order by p1.prod_name;
```

Result

Prod_Name	Jan_98_Sales	Total_98_Sales
Aroma Roma	1653.00	21697.50
Aroma Sheffield Steel Teapot	120.00	1122.00
Aroma Sounds Cassette	58.50	866.00
Aroma baseball cap	7.95	2960.15
Aroma t-shirt	470.85	4470.50
Assam Gold Blend	652.00	11375.00

(1 of 2)

Result

Prod_Name	Jan_98_Sales	Total_98_Sales
Assam Grade A	352.00	5429.00
Breakfast Blend	608.25	6394.75
Cafe Au Lait	1936.50	24050.50
Colombiano	2148.00	22528.50
Darjeeling Number 1	867.50	8590.00
Darjeeling Special	1355.00	17787.50
Demitasse Ms	2163.00	35523.50
Earl Grey	540.50	6608.50
English Breakfast	393.00	5365.50
Espresso Machine Italiano	899.55	4397.80
Expresso XO	2935.50	27362.00
French Press, 2-Cup	104.65	1196.00
French Press, 4-Cup	19.95	1109.20
Gold Tips	440.00	5381.50
Irish Breakfast	703.25	7455.50
La Antigua	2643.25	22244.50
Lotta Latte	3195.00	31200.00
NA Lite	1319.00	27457.00
...		

(2 of 2)

Correlated Subqueries in the Select List

Although select-list subqueries must return a single value or no value, they can be executed *more than once* in reference to results returned by the main query. In this way, such *correlated subqueries* in the select list can be used to the same effect as subqueries in the FROM clause.

A correlated subquery is closely related to the main query through cross-references to specific values in rows retrieved by the main query. For example, a correlated subquery might reference values in the **Month** column of the main query; therefore, the subquery returns a new value each time the value of the **Month** column changes. These cross-references are expressed with table correlation names assigned in the FROM clause.

About the Query

The example query presents the same business question as the query on page 4-9, but places the subquery in the select list instead of in the FROM clause. The query compares the sales of products in San Jose during January 1998 with annual sales of products in San Jose in the same year.

To enable the subquery to return a series of values instead of one fixed value, three *cross-references* correlate the subquery with the main query:

```
p1.prod_name = p2.prod_name
d1.year = d2.year
r1.city = r2.city
```

The correlation names p2, d2, and r2, defined in the FROM clause of the subquery, remove ambiguity. Each correlation condition references a specific product, year, and city in the row currently being processed by the main query. These cross-references are sometimes called *outer references*.

Usage Notes

When an aggregate function occurs in the select list of the main query, a GROUP BY clause is required. Column names referenced in a correlation condition of a subquery must appear in the GROUP BY clause of the main query; therefore, the following columns must be listed in the GROUP BY clause, as well as the **Prod_Name** column:

```
d1.year, r1.city
```

As database identifiers, correlation names must begin with a letter and contain no more than 128 characters. A combination of letters, digits, and underscores can follow the initial letter. (A keyword cannot serve as a database identifier.)

Using Cross-References

Question

What were the monthly sales of Lotta Latte in San Jose during the first three months of 1999 and 1998?

Example Query

```
select q.prod_name, e.month, sum(dollars) as sales_99,
   (select sum(dollars)
   from store t natural join sales s
      natural join product p
      natural join period d
   where d.month = e.month
      and d.year = e.year-1
      and p.prod_name = q.prod_name
      and t.city = u.city) as sales_98
from store u natural join product q
   natural join period e natural join sales l
where qtr = 'Q1_99'
   and prod_name like 'Lotta Latte%'
   and city like 'San J%'
group by q.prod_name, e.month, e.year, u.city;
```

Result

Prod_Name	Month	Sales_99	Sales_98
Lotta Latte	JAN	1611.00	3195.00
Lotta Latte	FEB	3162.50	4239.50
Lotta Latte	MAR	2561.50	2980.50

Cross-References with Expressions

Cross-references in subqueries are not limited to qualified column names; they can also be expressions. For example, the following expressions are valid cross-references:

```
period.year-1 (previous year)
period.quarter-1 (previous quarter)
```

These kinds of generalized cross-references simplify the design of applications written for client tools.

About the Query

This query returns the monthly Lotta Latte sales in San Jose during the first three months of both 1999 and 1998. The key to the correlation is that the intended result contains data from the same months but for different years.

The FROM clause of the main query assigns correlation names to all of the joined tables:

```
from store u natural join product q
   natural join period e natural join sales l
```

The subquery then correlates its execution with the execution of the main query based on the following conditions in the WHERE clause:

```
d.month = e.month
d.year = e.year-1
p.prod_name = q.prod_name
t.city = u.city
```

As the main query retrieves rows, the values of each column in the parent query can change, and the correlation conditions transmit this change to the subquery. The cross-reference to the previous year as *year-1* generalizes the subquery by eliminating a constant value (1998).

To change the query to report on other year periods, only the year constraint in the main query need be changed.

Usage Notes

Whenever possible, generalize correlated subqueries and minimize user interaction by using expressions as cross-references. For more information about generalizing queries, refer to Chapter 6, "Macros, Views, and Temporary Tables."

Calculating Percentages of Quarter and Year

Question

What were the monthly sales totals in the first quarter of 1998 for products sold in one-pound bags in San Jose? What were the corresponding *share of quarter and share of year percentages* for each monthly total?

Example Query

```
select pj.prod_name, dj.month, sum(dollars) as mon_sales_98,
   dec(100 * sum(dollars)/
      (select sum(si.dollars)
      from store ri natural join sales si
         natural join product pi
         natural join period di
      where di.qtr = dj.qtr
         and di.year = dj.year
         and pi.prod_name = pj.prod_name
         and pi.pkg_type = pj.pkg_type
         and ri.city = rj.city), 7, 2) as pct_qtr1,
   dec(100 * sum(dollars)/
      (select sum(si.dollars)
      from store ri natural join sales si
         natural join product pi
         natural join period di
      where di.year = dj.year
         and pi.prod_name = pj.prod_name
         and pi.pkg_type = pj.pkg_type
         and ri.city = rj.city), 7, 2) as pct_yr
from store rj natural join sales sj
   natural join product pj
   natural join period dj
where rj.city = 'San Jose'
   and dj.year = 1998
   and dj.qtr = 'Q1_98'
   and pkg_type = 'One-pound bag'
group by pj.prod_name, dj.month, dj.qtr, dj.year, pj.pkg_type,
   rj.city
order by pj.prod_name, pct_qtr1 desc;
```

Result

Prod_Name	Month	Mon_Sales_98	Pct_Qtr1	Pct_Yr
Aroma Roma	FEB	688.75	39.91	8.73
Aroma Roma	JAN	594.50	34.45	7.54
Aroma Roma	MAR	442.25	25.63	5.60
Cafe Au Lait	MAR	742.00	40.61	10.27
Cafe Au Lait	JAN	600.50	32.86	8.31
Cafe Au Lait	FEB	484.50	26.51	6.71
...				

Calculations with Select-List Subqueries

Monthly percentages for quarters, years, or other time periods can be calculated with a select-list subquery. The main query retrieves the monthly sales figures and two subqueries retrieve the quarterly and yearly sales figures. The monthly percentages require simple calculations: ratios of month-to-quarter sales and month-to-year sales.

About the Query

This example query calculates month-to-quarter and month-to-year sales percentages for selected coffee products sold in San Jose during the first quarter of 1998. After calculating the percentages, the query orders the result table by product and quarterly percentage in descending order.

Usage Notes

Like the previous example, this select-list subquery requires explicit cross-references to correlate the execution of the subquery with the retrieval of new rows by the main query.

In most cases, this kind of comparison query runs faster and is easier to conceptualize as a series of subqueries expressed in the FROM clause. Nonetheless, if the correlated method is your preferred way of expressing the query and the query performs well, there is no need to rewrite it. Both approaches offer the same functions and produce the same results.

Using Subqueries in the WHERE Clause

Question

During which days in June 1999 were Lotta Latte sales figures at stores in the Chicago district lower than the average daily sales figures for the same product in the same district during June 1998?

Example Query

```
select prod_name, district, date, dollars as sales_99
from market natural join store
   natural join sales
   natural join product
   natural join period
where prod_name like 'Lotta%'
   and district like 'Chic%'
   and year = 1999
   and month = 'JUN'
   and dollars <
      (select avg(dollars)
         from market natural join store
            natural join sales
            natural join product
            natural join period
         where prod_name like 'Lotta%'
            and district like 'Chic%'
            and year = 1998
            and month = 'JUN');
```

Result

Prod_Name	District	Date	Sales_99
Lotta Latte	Chicago	1999-06-08	76.50
Lotta Latte	Chicago	1999-06-11	59.50
Lotta Latte	Chicago	1999-06-17	42.50
Lotta Latte	Chicago	1999-06-18	76.50
Lotta Latte	Chicago	1999-06-30	110.50

Comparisons with WHERE Clause Subqueries

So far, this chapter has focused on the equivalent functionality but different syntax involved in placing subqueries in the select list or the FROM clause. Subqueries can also be used as search conditions or predicates in the WHERE clause as a means of pushing complex constraints through to the early stages of the main query's execution. For example, although you cannot use a set function as part of a simple WHERE clause search condition, you *can* use a set function in the WHERE clause if it is embedded inside a subquery.

About the Query

This query returns the Lotta Latte sales figures at stores in the Chicago district during 1999 for days on which the sales were lower than the average daily Lotta Latte sales figure for the same city during 1998.

The subquery in this example is scalar—it produces one value. After the subquery has calculated the average dollar figure per day in Chicago in 1998, that single average value is used as a constraint on all the rows returned by the main query. Only those figures for 1999 that were lower than the 1998 average are displayed in the result set; the average figure itself cannot be displayed unless the subquery is moved into the select list or the FROM clause.

Usage Notes

The logical order of query processing dictates that WHERE clause constraints are applied by the server immediately after the tables in the FROM clause were joined and prior to any calculations with set functions (such as AVG and SUM), RISQL display functions, and so on. Therefore, you cannot use one of those functions in a simple search condition in the WHERE clause.

Using the ALL Comparison Predicate

Question

What product registered the highest daily sales total in Hartford, Connecticut, in January 2000?

Example Query

```
select prod_name, date, dollars
from store natural join sales
   natural join product
   natural join period
where year = 2000
   and city = 'Hartford'
   and month = 'JAN'
   and dollars >= all
      (select dollars
      from store natural join sales
         natural join product
         natural join period
      where year = 2000
         and city = 'Hartford'
         and month = 'JAN');
```

Result

Prod_Name	Date	Dollars
NA Lite	2000-01-24	414.00

Comparison Predicates in Subqueries

The predicates ALL, ANY, SOME, and EXISTS are useful for expressing conditions on groups of values retrieved by a subquery. A comparison predicate states a logical relationship between two values: The comparison is true, false, or unknown with respect to a given row. (The ANY and SOME predicates are synonyms.)

Predicate	Evaluates to "true" when	When no value is returned
ALL	The comparison is true for all values returned by the subquery.	Evaluates to true.
ANY, SOME	The comparison is true for at least one of the values returned by the subquery.	Evaluates to false.
EXISTS	The subquery produces at least one row.	Evaluates to false.

For more information about these predicates, refer to the *SQL Reference Guide*.

About the Query

The example query returns the name of the product that recorded the highest daily sales total in Hartford in January 2000 and the specific date when that total was recorded. The query could be rewritten to return the lowest total by replacing the greater-than or equal-to (>=) operator with a less-than or equal-to (<=) operator.

Usage Notes

An alternative (and more concise) way to write this query is to use the RANK function in the WHEN clause:

```
select prod_name, date, dollars
from sales natural join period
   natural join product
   natural join store
where year = 2000
   and month = 'JAN'
   and city = 'Hartford'
when rank(dollars) = 1;
```

However, RANK queries can yield multiple rows that tie for the rank of 1, while a subquery in the WHERE clause must return one row or no rows.

Using the EXISTS Predicate

Question

Which suppliers closed at least one order in March 2000?

Example Query

```
select distinct name as supplier_name
from supplier
where exists
         (select * from orders
         where supplier.supkey = orders.supkey
         and extract(year from close_date) = 2000
         and extract(month from close_date) = 03);
```

Result

Supplier Name
Aroma East Mfg.
Aroma West Mfg.
Crashing By Design
Espresso Express
Leaves of London
Tea Makers, Inc.
Western Emporium

EXISTS Predicate

The EXISTS predicate operates on a subquery and evaluates to true or false. If it evaluates to true, the main query produces a result set. If it evaluates to false, the main query returns no rows.

About the Query

The example query returns the names of each supplier that closed one or more orders with the Aroma Coffee Company in March 2000.

The subquery contains three conditions that test whether any such suppliers exist. The first condition is a join of the **Supplier** and **Orders** tables over the **Supkey** column. The second and third conditions are expressed with the EXTRACT function, which checks for the appropriate dateparts in the **Close_Date** column of the **Orders** table. (For a detailed example of this function, refer to "Using the EXTRACT Function" on page 3-38.)

Usage Notes

You could ask the same business question by joining the **Supplier**, **Orders**, and **Period** tables:

```
select distinct name as supplier_name
from supplier s, orders o, period p
where s.supkey = o.supkey
   and o.close_date = p.date
   and year = 2000
   and month = 'MAR';
```

This alternative query must join the **Orders** and **Period** tables over their **Close_Date** and **Date** columns, not their **Perkey** columns. This is because the **Perkey** column indicates the date when the orders were entered, which might be in an earlier month. For example, an order might be entered in the last week of February but received and closed in the first week of March.

This join of the **Orders** and **Period** tables is a good example of a join over columns that have no primary-key to foreign-key relationship. The join is possible because the **Close_Date** and **Date** columns have comparable data types.

The opposite of the EXISTS predicate is NOT EXISTS:

```
...
where not exists (select...)
```

For more information about this predicate, refer to the *SQL Reference Guide*.

Using the SOME or ANY Predicate

Question

Which suppliers have *at some point* provided orders priced at more than $10,000? What were the actual prices of orders closed in March 2000 by those suppliers?

Example Query

```
select name as supplier_name, price
from supplier natural join orders
   where extract(year from close_date) = 2000
      and extract(month from close_date) = 03
      and supplier_name = some
         (select name from supplier
            natural join orders
            where price > 10000)
order by supplier_name;
```

Result

Supplier_Name	Price
Aroma West Mfg.	4425.00
Espresso Express	30250.00
Espresso Express	25100.00
Espresso Express	26400.00
Espresso Express	22700.00
Western Emporium	10234.50

SOME or ANY Predicate

The SOME and ANY comparison predicates evaluate to true when at least one of the values returned by the subquery meets the conditions specified in it. You can use these predicates interchangeably because they are synonyms.

SOME and ANY are useful for retaining rows in the result set when they meet all the conditions specified in the inner query but not all the conditions in the outer query. For example, the outer query might request a list of suppliers that shipped orders in a specific month, regardless of their price, while the inner query might request a list of suppliers that have shipped orders that cost more than a specific amount in *any* month.

About the Query

The example query uses the SOME predicate to return a list of suppliers and order prices:

- The subquery returns a list of suppliers that have supplied at least one order that cost more than $10,000.
- The main query takes that list of suppliers and matches it with records of orders closed in March 2000, with no constraint on the price of each order.

The last row in the result set shows that an order was supplied by Aroma West Mfg. in March 2000 that cost $4,425.00. The presence of this row indicates that at some other point in time, Aroma West Mfg. supplied at least one order that cost more than $10,000.

Usage Notes

The EXTRACT function is used in the same way in this query as in the example of the EXISTS predicate on page 4-28.

For more information about these predicates, refer to the *SQL Reference Guide*.

Summary

This chapter described how to write queries that compare data and display the results in a readable format. Various approaches are illustrated:

- CASE expressions
- FROM clause subqueries
- Select-list subqueries, including correlated subqueries
- WHERE clause subqueries

The chapter ended with examples of the ALL, ANY, SOME, and EXISTS comparison predicates, which can be used as conditions on subquery results.

Some of the more complex examples showed how to include calculations in comparison queries, such as percentages that represent share of quarter or share of year.

Important: *In general, query performance is faster when comparison queries use CASE expressions rather than subqueries. If subqueries are necessary, however, the preferred method is to use the FROM clause rather than the select list.*

Chapter

Joins and Unions

5

In This Chapter

This chapter describes two ways to combine data from different tables:

- By joining the tables
- By using the UNION, EXCEPT, and INTERSECT operators

The first part of this chapter presents several examples of inner and outer joins.

The second part illustrates how to combine data from different tables by using UNION, EXCEPT, and INTERSECT operators, which take the intermediate result set from one query expression and combine it with the result set from another query expression.

Join of Two Tables

State Table

City	State
Jacksonville	FL
Miami	FL
Nashville	TN

Region Table

City	Area
Jacksonville	South
Miami	South
New Orleans	South

Example Query

```
select * from state, region;
```

Cartesian Product (join predicate not specified)

City	State	City	Area
Jacksonville	FL	Jacksonville	South
Jacksonville	FL	Miami	South
Jacksonville	FL	New Orleans	South
Miami	FL	Jacksonville	South
Miami	FL	Miami	South
Miami	FL	New Orleans	South
Nashville	TN	Jacksonville	South
Nashville	TN	Miami	South
Nashville	TN	New Orleans	South

Example Query

```
select * from state, region
where state.city = region.city;
```

Subset of Cartesian Product (join predicate specified)

State:City	State:State	Region:City	Region:Area
Jacksonville	FL	Jacksonville	South
Miami	FL	Miami	South

Inner Joins

Most queries join information from different tables. Any two tables can be joined over columns with comparable data types; joins are not dependent on primary-key to foreign-key relationships.

Cartesian Product

When two or more tables are referenced in the FROM clause of a query, the database server joins the tables. If neither the FROM clause nor the WHERE clause specifies a predicate for the join, the server computes a Cartesian product that contains $m * n$ rows, where m is the number of rows in the first table and n is the number of rows in the second table. This product is the set of all possible combinations formed by concatenating a row from the first table with a row from the second table.

Important: *If the OPTION CROSS_JOIN parameter in the **rbw.config** file is set to OFF (the default), cross-join queries are not executed.*

Subset of the Cartesian Product

If tables are explicitly joined over columns with comparable datatypes, the server computes a subset of the Cartesian product. This subset contains only those rows where the values in the joining columns match. For the duration of the query, the subset functions as a derived table that can be joined with other tables or the results of other query expressions.

About the Query

The **State** and **Region** tables both contain **City** columns, which are specified as the joining columns in the WHERE clause. Consequently, only those rows of the Cartesian product that have matching **City** keys are displayed in the result. In the example query, the result table contains only two rows whereas the full Cartesian product of these two tables contains nine rows.

The joining columns could alternatively be specified in the FROM clause, as discussed on page 5-7.

Important: *The tables used in the following three queries are not part of the **Aroma** database; Aroma tables are used in the examples later in this chapter.*

Different Ways to Join Tables

Question

How long did the Christmas special promotion run in 1998 and 1999? What were the total sales for products sold on that promotion in each year, and what was the average sales total per day in each year?

Example Query 1

```
select promo_desc, year, sum(dollars) as sales,
   datediff(day, end_date, start_date)+1 as days_on_promo,
   string(sales/days_on_promo, 7, 2) as per_day
from period natural join sales
   natural join promotion
where promo_desc like 'Christmas%'
   and year in (1998, 1999)
group by promo_desc, year, days_on_promo;
```

Example Query 2

```
select promo_desc, year, sum(dollars) as sales,
   datediff(day, end_date, start_date)+1 as days_on_promo,
   string(sales/days_on_promo, 7, 2) as per_day
from period join sales on period.perkey = sales.perkey
   join promotion on promotion.promokey = sales.promokey
where promo_desc like 'Christmas%'
   and year in (1998, 1999)
group by promo_desc, year, days_on_promo;
```

Example Query 3

```
select promo_desc, year, sum(dollars) as sales,
   datediff(day, end_date, start_date)+1 as days_on_promo,
   string(sales/days_on_promo, 7, 2) as per_day
from period join sales using(perkey)
   join promotion using(promokey)
where promo_desc like 'Christmas%'
   and year in (1998, 1999)
group by promo_desc, year, days_on_promo;
```

Three Queries, Same Result

Promo_Desc	Year	Sales	Days_on_Promo	Per_Day
Christmas special	1999	1230.00	31	39.67
Christmas special	1998	690.00	31	22.25

Joins in the FROM Clause

You can explicitly join tables in the FROM clause in three ways:

- Natural join
- Join over named columns (USING syntax)
- Join over predicate (ON syntax)

About the Query

This query joins the **Promotion**, **Period**, and **Sales** tables over columns with identical names; therefore, it can be abbreviated with the NATURAL JOIN syntax, as shown in Query 1. Queries 2 and 3 show alternative methods of specifying inner equijoins in the FROM clause. The result set is the same in all three cases; however, the ON and USING join specifications retain both joining columns in their intermediate result sets, whereas the NATURAL JOIN specification combines each pair of joining columns into one column.

Note the use of scalar functions in this query:

- The DATEDIFF function is used to calculate the duration of the Christmas promotion. This function is discussed in detail on page 3-36.
- The STRING function is used to scale the **Per_Day** column values down to a precision of two decimal places; without this function, the following expression would return long-numeric values:

```
sales/promo_days
```

Usage Notes

Natural joins operate on all pairs of columns that have identical names and should be used with caution; otherwise, tables might be inadvertently joined over columns that happen to have the same name but were not intended to participate in the join.

In the retail schema of the **Aroma** database, all the primary-key to foreign-key relationships are based on columns with the same name, so natural joins are effective for most queries that involve the **Sales** table and its dimensions.

For an example of a join over nonprimary-key and foreign-key columns, refer to "Calculating Elapsed Days: DATEDIFF" on page 3-37. For a complete discussion of join syntax, refer to the *SQL Reference Guide*.

System Table Join

Question

What are the names of the segments and physical storage units (PSUs) used to store the Aroma **Sales** table?

Example Query

```
select segname as storage, location as psu_location,
   tname as table_name
from rbw_storage join rbw_segments on
   rbw_storage.segname = rbw_segments.name
where table_name = 'SALES'
order by psu_location;
```

Result

Storage	PSU_Location	Table_Name
DEFAULT_SEGMENT_23	dfltseg23_psu1	SALES
DAILY_DATA1	sales_psu1	SALES
DAILY_DATA1	sales_psu2	SALES
DAILY_DATA2	sales_psu3	SALES
DAILY_DATA2	sales_psu4	SALES

Joining System Tables

Database administrators need to know the relationships between different database objects, such as tables and indexes or tables and segments. To facilitate access to this kind of information, Red Brick Decision Server system tables can be joined in the same way as all other database tables.

About the Query

This query joins two system tables to identify the names of both default and user-defined segments and their associated PSUs for the **Sales** table in the **Aroma** database.

Usage Notes

The following WHERE clause condition must use uppercase for SALES:

```
table_name = 'SALES'
```

Otherwise, no matching rows are found.

For detailed information about system tables, table segmentation, and so on, refer to the *Administrator's Guide*.

Self-Joins

Question

Which products in the **Product** table have the same names but different types of packaging?

Example Query

```
select a.prod_name as products,
   a.pkg_type
from product a, product b
where a.prod_name = b.prod_name
   and a.pkg_type <> b.pkg_type
order by products, a.pkg_type;
```

Result

Product	Pkg_Type
Aroma Roma	No pkg
Aroma Roma	One-pound bag
Assam Gold Blend	No pkg
Assam Gold Blend	Qtr-pound bag
Assam Grade A	No pkg
Assam Grade A	Qtr-pound bag
Breakfast Blend	No pkg
Breakfast Blend	Qtr-pound bag
Cafe Au Lait	No pkg
Cafe Au Lait	One-pound bag
Colombiano	No pkg

(1 of 2)

Product	Pkg_Type
Colombiano	One-pound bag
Darjeeling Number 1	No pkg
Darjeeling Number 1	Qtr-pound bag
Darjeeling Special	No pkg
Darjeeling Special	Qtr-pound bag
Demitasse Ms	No pkg
Demitasse Ms	One-pound bag
Earl Grey	No pkg
Earl Grey	Qtr-pound bag
English Breakfast	No pkg
English Breakfast	Qtr-pound bag
Expresso XO	No pkg
Expresso XO	One-pound bag
Gold Tips	No pkg
Gold Tips	Qtr-pound bag
Irish Breakfast	No pkg
Irish Breakfast	Qtr-pound bag
...	

(2 of 2)

Joining a Table to Itself

The tables being joined in a query do not need to be distinct; you can join any table to itself as long as you give each table reference a different name. Self-joins are useful for discovering relationships between different columns of data in the same table.

About the Query

This query joins the **Product** table to itself over the **Prod_Name** column, using the aliases *a* and *b* to distinguish the table references:

```
from product a, product b
```

The self-join compares **Product** table *a* to **Product** table *b* to find rows where the product names match but the package types differ:

```
where a.prod_name = b.prod_name
   and a.pkg_type <> b.pkg_type
```

The result set consists of a list of each pair of identically named products and their individual package types.

Outer Join of Two Tables

Example Query (left outer join)

```
select * from state left outer join region
   on state.city = region.city;
```

Result

State:City	State:State	Region:City	Region:Area
Jacksonville	FL	Jacksonville	South
Miami	FL	Miami	South
Nashville	TN	NULL	NULL

Example Query (right outer join)

```
select * from state right outer join region
  on state.city = region.city;
```

Result

State:City	State:State	Region:City	Region:Area
Jacksonville	FL	Jacksonville	South
Miami	FL	Miami	South
NULL	NULL	New Orleans	South

Example Query (full outer join)

```
select * from state full outer join region
  on state.city = region.city;
```

Result

State:City	State:State	Region:City	Region:Area
Jacksonville	FL	Jacksonville	South
Miami	FL	Miami	South
Nashville	TN	NULL	NULL
NULL	NULL	New Orleans	South

Important: *These examples use the tables introduced on* *page 5-3*.

Outer Joins

In most cases, tables are joined according to search conditions that find only the rows with matching values; this type of join is known as an *inner equijoin*. In some cases, however, decision-support analysis requires *outer joins*, which retrieve both matching and non-matching rows, or *non-equijoins*, which express, for example, a greater-than or less-than relationship.

An outer join operation returns all the rows returned by an inner join plus all the rows from one table that do not match any row from the other table. An outer join can be *left*, *right*, or *full*, depending on whether rows from the left, right, or both tables are retained. The first table listed in the FROM clause is referred to as the left table and the second as the right table. For all three types of outer join, NULLs are used to represent empty columns in rows that do not match.

Syntax

As shown in the preceding examples, an outer join between two tables can be specified in the FROM clause with the OUTER JOIN keywords followed by the ON subclause:

```
FROM table_1 LEFT|RIGHT|FULL OUTER JOIN table_2
   ON table_1.column = table_2.column
```

For details about other ways to specify outer join predicates in the FROM clause, refer to the *SQL Reference Guide*.

About the Queries

- The result of the left outer join contains every row from the **State** table and all matching rows in the **Region** table. Rows found only in the **Region** table are not displayed.
- The result of the right outer join contains every row from the **Region** table and all matching rows from the **State** table. Rows found only in the **State** table are not displayed.
- The result of the full outer join contains those rows that are unique to each table, as well as those rows that are common to both tables.

Fact-to-Fact Join

Question

What were the prices paid per line item, per full order, or per line item and full order for order numbers 3619 through 3626?

Example Query

```
select coalesce(o.order_no, l.order_no) as order_num,
   order_type, o.price as full_cost,
   l.price as line_cost
from orders o left outer join line_items l
   on o.order_no = l.order_no
   join period on o.perkey = period.perkey
where o.order_no between 3619 and 3626
order by order_num;
```

Result

Order_Num	Order_Type	Full_Cost	Line_Cost
3619	Tea	4325.25	725.25
3619	Tea	4325.25	400.00
3619	Tea	4325.25	400.00
3619	Tea	4325.25	400.00
3619	Tea	4325.25	400.00
3619	Tea	4325.25	400.00
3619	Tea	4325.25	400.00
3619	Tea	4325.25	400.00
3619	Tea	4325.25	400.00
3619	Tea	4325.25	400.00

(1 of 2)

Result

Order_Num	Order_Type	Full_Cost	Line_Cost
3620	Tea	4325.25	NULL
3621	Spice	10234.50	10234.50
3622	Spice	10234.50	10234.50
3623	Hardware	4425.00	400.00
3623	Hardware	4425.00	400.00
3623	Hardware	4425.00	500.00
3623	Hardware	4425.00	450.00
3623	Hardware	4425.00	500.00
3623	Hardware	4425.00	275.00
3623	Hardware	4425.00	650.00
3623	Hardware	4425.00	1250.00
3624	Hardware	4425.00	400.00
3624	Hardware	4425.00	500.00
3624	Hardware	4425.00	450.00
3624	Hardware	4425.00	400.00
3624	Hardware	4425.00	500.00
3624	Hardware	4425.00	275.00
3624	Hardware	4425.00	650.00
3624	Hardware	4425.00	1250.00
3625	Clothing	3995.95	2500.00
3625	Clothing	3995.95	1495.95
3626	Hardware	16500.00	NULL

(2 of 2)

Left Outer Join

Outer joins are often used to join fact tables as a means of comparing related sets of measurements that cannot be queried from a single table.

About the Query

The **Orders** and **Line_Items** tables store related facts; however, the line-item detail information for an order might be loaded into the database sometime after the order information is loaded. If an analyst wants to see both order and line-item prices, if available, or just the order prices when no line-item price is available, an outer join is required.

The query returns the prices of both full orders and line items; if the line item prices are unavailable, the full order price is still displayed and the **Line_Price** and **Line_Orders** columns contain NULLs. This effect is achieved by using a left outer join, with the **Orders** table being treated as the left table.

The COALESCE function is used to derive one column heading in the report from two columns in the select list:

```
coalesce(o.order_no, l.order_no) as order_num
```

Because either column might be NULL, the COALESCE function will produce the value of the non-NULL column. Without this function, the result set would consist of duplicate columns for the order numbers.

Usage Notes

For details on how to express outer join conditions with standard SQL, refer to the *SQL Reference Guide*.

The example query uses the tables in the **Aroma** purchasing schema, which is described in detail in Appendix A, "The Complete Aroma Database."

Fact-to-Fact Join

Question

In weeks 12 and 13 of 2000, how did revenues from sales compare with expenditures on orders?

Example Query

```
select date, extract(week from date) as wk_no, prices, sales
from
      ((select d1.date, sum(price)
      from orders natural join period d1
      where d1.year = 2000 and d1.week in (12, 13)
      group by d1.date) as t1
   full outer join
      (select d2.date, sum(dollars)
      from sales natural join period d2
      where d2.year = 2000 and d2.week in (12, 13)
      group by d2.date) as t2
   on t1.date = t2.date) as t3(order_date, prices, date, sales)
order by wk_no, date
break by wk_no summing prices, sales;
```

Result

Date	Wk_No	Prices	Sales
2000-03-12	12	NULL	9991.65
2000-03-13	12	31800.00	10162.75
2000-03-14	12	NULL	9514.55
2000-03-15	12	NULL	9074.10
2000-03-16	12	NULL	11009.55
2000-03-17	12	NULL	9177.90
2000-03-18	12	NULL	7412.65

(1 of 2)

Date	Wk_No	Prices	Sales
NULL	12	31800.00	66343.15
2000-03-19	13	NULL	8620.25
2000-03-20	13	27025.25	8417.95
2000-03-21	13	NULL	8230.05
2000-03-22	13	NULL	9870.20
2000-03-23	13	NULL	8757.50
2000-03-24	13	NULL	8394.25
2000-03-25	13	3995.95	10046.90
NULL	13	31021.20	62337.10
NULL	NULL	62821.20	128680.25

(2 of 2)

Full Outer Join with ORDER BY, BREAK BY

Full outer joins return results that include rows from the left and right tables, whether or not they contain matching values in the joining columns. In the result set, the columns for which no match was found contain NULLs.

About the Query

The **Sales** and **Line_Items** tables store different sets of facts but share two dimension tables, **Product** and **Period**. To create a report of orders and sales over a given period, you can inner-join each fact table to the **Period** table, then outer-join the results of the inner joins. One way to do this is to use subqueries in the FROM clause.

The first subquery evaluates to a table named **t1**, the second to a table named **t2**. Table **t3** is the result of the full outer join of **t1** and **t2**. Table **t3** consists of four named columns:

```
t3(order_date, prices, date, sales)
```

The select list of the main query references three of these columns, **Prices**, **Date**, and **Sales**. A fourth column in the select list, **Wk_No**, is extracted from the **Date** column with the EXTRACT scalar function:

```
extract(week from date) as wk_no
```

The ORDER BY clause and its BREAK BY subclause sort the data by week and date, then display subtotals for each week for both the **Prices** column and the **Sales** column. The last row of the result set displays grand totals.

Usage Notes

Table aliases are required in this query because the same table name cannot be repeated in the FROM clause. For example, the **Period** table is referenced as **d1** in one join specification and as **d2** in another.

Any column referenced in a BREAK BY clause must also be listed in the ORDER BY clause. For more information about these clauses, refer to the *SQL Reference Guide*.

The preceding example is similar to some of the FROM clause subqueries in Chapter 4, "Comparison Queries."

For more examples of queries that use datetime scalar functions, refer to Chapter 3, "Data Analysis."

OR Versus UNION

Question

What were the total sales in week 52 of 1999 for all Aroma stores classified as "Medium"? What were the totals during the same period for "Large" stores?

Example Query with OR Condition

```
select store_name as store, store_type as size, state,
   sum(dollars) as sales
from period t join sales s on t.perkey = s.perkey
   join store r on r.storekey = s.storekey
where (store_type = 'Medium' or store_type = 'Large')
   and year = 1999
   and week = 52
group by store, size, state
order by size, store;
```

Example UNION Query

```
select store_name as store, store_type as size, state,
   sum(dollars) as sales
from period t join sales s on t.perkey = s.perkey
   join store r on r.storekey = s.storekey
where store_type = 'Medium'
   and year = 1999
   and week = 52
group by store, size, state
union
select store_name as store, store_type as size, state,
   sum(dollars)
from period t join sales s on t.perkey = s.perkey
   join store r on r.storekey = s.storekey
where store_type = 'Large'
   and year = 1999
   and week = 52
group by store, size, state
order by size, store;
```

Two Queries, Same Result

Store	Size	State	Sales
Beaches Brew	Large	CA	2908.80
Miami Espresso	Large	FL	4582.00
Olympic Coffee Company	Large	GA	3732.50
San Jose Roasting Company	Large	CA	3933.15
Beans of Boston	Medium	MA	3772.75
Cupertino Coffee Supply	Medium	CA	2893.00
Java Judy's	Medium	AZ	3011.25
Moulin Rouge Roasting	Medium	LA	3972.00
Texas Teahouse	Medium	TX	3382.75

Combining Result Sets: UNION

You can use the UNION, EXCEPT, and INTERSECT operators to combine the output of two or more query expressions into a single set of rows and columns. The server evaluates each query expression independently, then combines the output, displaying column headings from the *first* expression. The server eliminates duplicate result rows unless you specify the ALL keyword.

UNION, INTERSECT, EXCEPT

```
query_expression UNION | INTERSECT | EXCEPT [ALL]
query_expression
[ORDER BY order_list]
[SUPPRESS BY suppress_list];
```

query_expression Any join or nonjoin query expression, as defined in the *SQL Reference Guide*.

If SUPPRESS BY and ORDER BY clauses are used, they must reference columns from the select list of the first query expression.

About the Query

The same business question can be answered by either specifying an OR condition in a single SELECT statement or combining two query expressions with a UNION operator.

Using the OR connective is easier in this simple example, but in some cases a UNION operation improves query performance. For example, suppose your query requires that you access data in two large fact tables. The outer join operation required by a single query might require more processing than using a UNION operation to combine the results of two query expressions.

The ORDER BY clause must reference the column *aliases*, not the column names, defined in the select list of the first query expression:

```
order by size, store
```

Usage Notes

UNION, INTERSECT, and EXCEPT queries must be symmetrical; that is, the number of columns and their order must be the same in the select lists on both sides of the UNION operator. Corresponding columns must have the same, or comparable, datatypes, although they may have different names.

Multiple UNION, INTERSECT, and EXCEPT operators can be used in a single statement; operations are evaluated from left to right unless you specify precedence with parentheses.

INTERSECT Operation

Question

Which bulk tea products sold on promotion in San Jose in 2000 were also sold on promotion in New Orleans in 1999? What promotions were run on those products?

Example Query

```
select prod_name as tea_name, promo_desc
from sales natural join class
   natural join product
   natural join store
   natural join period
   natural join promotion
where city = 'San Jose'
   and year = 2000
   and class_desc like 'Bulk tea%'
intersect
select prod_name, promo_desc
from sales natural join class
   natural join product
   natural join store
   natural join period
   natural join promotion
where city = 'New Orleans'
   and year = 1999
   and class_desc like 'Bulk tea%'
   and promo_desc not like 'No promo%'
order by promo_desc;
```

Result

Tea_Name	Promo_Desc
Irish Breakfast	Aroma catalog coupon
Special Tips	Aroma catalog coupon
Darjeeling Special	Store display
Darjeeling Special	Temporary price reduction
Gold Tips	Temporary price reduction

Finding Common Rows: INTERSECT

You can use the INTERSECT operator to return only those rows that are *common to* the results returned by two or more query expressions.

About the Query

The example query finds the intersection of two query expressions, one that returns a list of bulk tea products sold on promotion in San Jose in 2000 and one that returns a similar list for New Orleans in 1999. The INTERSECT operator eliminates all rows that are not found in both preliminary result sets.

Usage Notes

The results of UNION, EXCEPT, and INTERSECT operations derive column headings only from the first query expression in the query; therefore, the column alias **Tea_Name** need only be specified in the first query expression.

INTERSECT Operation Inside Subquery

Question

Of the products that were ordered in March 2000, which ones were also sold at the Coffee Connection store during that month?

What did orders of these products cost in that month?

What was the total revenue (sum of sales dollars) for those products in the entire Northern region during that month?

Example Query

```
select product, cost_of_orders, revenue_north
from (select prod_name
      from product natural join sales natural join period
         natural join store
      where year = 2000 and month = 'MAR'
         and store_name = 'Coffee Connection'
      intersect
      select prod_name
      from product natural join line_items natural join period
      where year = 2000 and month = 'MAR') as p(product)
   natural join
      (select prod_name, sum(price)
      from product natural join line_items natural join period
      where year = 2000 and month = 'MAR'
      group by prod_name) as c(product, cost_of_orders)
   natural join
      (select prod_name, sum(dollars)
      from product natural join sales natural join period
         natural join store natural join market
      where year = 2000 and month = 'MAR' and region = 'North'
      group by prod_name) as r(product, revenue_north)
order by product;
```

Result

Product	Cost_of_Orders	Revenue_North
Aroma Roma	7300.00	3190.00
Cafe Au Lait	7300.00	3975.50
Colombiano	7300.00	3907.50
Demitasse Ms	8500.00	6081.25
Expresso XO	7300.00	4218.50
La Antigua	7300.00	3510.50
Lotta Latte	7300.00	4273.00
NA Lite	7300.00	6480.00
Veracruzano	7300.00	4055.00
Xalapa Lapa	7300.00	6896.50

INTERSECT of Fact Table Data

The UNION, INTERSECT, and EXCEPT operators are useful for querying tables that contain similar or comparable sets of facts.

About the Query

This query contains three subqueries in the FROM clause. The function of the INTERSECT operator inside the first subquery is to produce a list of products that were ordered in March 2000 as well as sold at the Coffee Connection store in the same month. This is done by placing the INTERSECT operator between query expressions that join two different fact tables, **Sales** and **Line_Items**.

The second subquery produces the sum of the order prices for March 2000 for the list of products produced by the first subquery.

The third subquery produces the sum of the sales dollars for the same list of products during the same month, but for the whole Northern region.

Usage Notes

The preceding query is similar to the examples of FROM clause subqueries in Chapter 4, "Comparison Queries." The select list of the main query consists entirely of columns named in the tables derived from the subqueries.

EXCEPT Operation

Question

What were the total 1999 revenues for stores in California cities that are not defined as HQ cities in the **Market** table?

Example Query

```
select city, store_name, sum(dollars) as sales_99
from (select city
      from store
      where state = 'CA'
      except
      select hq_city
      from market
      where hq_state = 'CA')
      as except_cities(city)
   natural join store
   natural join sales
   natural join period
where year = 1999
group by city, store_name
order by sales_99 desc;
```

Result

City	Store_Name	Sales_99
Cupertino	Cupertino Coffee Supply	196439.75
Los Gatos	Roasters, Los Gatos	175048.75

EXCEPT: Finding the Exceptions in Two Result Sets

The EXCEPT operator finds the exceptions in (or the difference between) the results of two query expressions. For example, an EXCEPT operation could compare lists of products sold at two stores, eliminate all the products sold at both, and retain only those products sold exclusively at the store specified in the first query expression.

About the Query

In the example query, the function of the EXCEPT operator is to select those California cities that are defined in the **City** column of the **Store** table but not in the **Hq_City** column of the **Market** table.

This query uses a subquery in the FROM clause to produce a derived table of cities that can be joined with the **Sales**, **Store**, and **Period** tables. The table derived from the subquery is given a correlation name and one column name:

```
except_cities(city)
```

This derived table can be joined with the **Store** table using a natural join over the **City** column.

Usage Notes

To test the outcome of the EXCEPT operation, you could run the subquery in this example as a query in its own right:

```
select city
from store
where state = 'CA'
except
select hq_city
from market
where hq_state = 'CA';

CITY
Cupertino
Los Gatos
```

For more examples of subqueries, refer to Chapter 4, “Comparison Queries.”

Summary

This chapter described:

- How to join tables
- How to combine the results of two independent query expressions by using the UNION, INTERSECT, and EXCEPT operators

Joining Tables

When the FROM clause of a query lists two or more tables, the server joins the tables. The server can perform both inner and outer joins between any two tables on any two columns with comparable data types. You can use either the FROM clause or the WHERE clause to write join specifications.

UNION, INTERSECT, and EXCEPT Operators

```
query_expression
UNION | INTERSECT | EXCEPT [ALL]
query_expression
[ORDER BY order_list]
[SUPPRESS BY suppress_list];
```

Chapter

Macros, Views, and Temporary Tables

6

In This Chapter

This chapter shows how to simplify SQL statements with RISQL macros.

A *macro* is an abbreviation for a complex expression. Macros allow you to write concise, reusable SQL statements.

This chapter also presents simple examples of two other means of simplifying data retrieval, views and temporary tables.

The examples in this chapter show how to:

- Abbreviate a lengthy or frequently used expression or query by writing a macro
- Write a macro that contains other macros
- Write generalized macros that use parameters
- Create and query a view
- Create, populate, and query a temporary table

Basic Macros

Question

What were the total sales of tea products during 1999?

CREATE MACRO Statement

```
create macro tea_products as
   (pt.classkey = 2 or pt.classkey = 5);
```

Example Query

```
select prod_name,
   case pt.classkey when 2 then 'Bulk Tea'
      when 5 then 'Pkg Tea' end as class,
   sum(dollars) as sales_99
from product pt join sales sa
      on pt.classkey = sa.classkey
      and pt.prodkey = sa.prodkey
   join period pd on pd.perkey = sa.perkey
where tea_products
   and year = 1999
group by prod_name, pt.classkey
order by sales_99 desc;
```

Result

Prod_Name	Class	Sales_99
Darjeeling Special	Bulk Tea	80610.50
Darjeeling Special	Pkg Tea	51266.00
Assam Gold Blend	Bulk Tea	42329.00
Darjeeling Number 1	Bulk Tea	34592.75
Irish Breakfast	Bulk Tea	27763.75

(1 of 2)

Prod_Name	Class	Sales_99
Assam Gold Blend	Pkg Tea	27192.50
English Breakfast	Bulk Tea	25848.00
Breakfast Blend	Bulk Tea	24594.00
Darjeeling Number 1	Pkg Tea	24232.00
Earl Grey	Bulk Tea	23269.50
Special Tips	Bulk Tea	22326.00
Assam Grade A	Bulk Tea	21964.00
Gold Tips	Bulk Tea	21584.50
Irish Breakfast	Pkg Tea	20084.00
English Breakfast	Pkg Tea	18955.00
Breakfast Blend	Pkg Tea	17031.50
Gold Tips	Pkg Tea	16783.25
Special Tips	Pkg Tea	16773.25
Assam Grade A	Pkg Tea	16724.00
Earl Grey	Pkg Tea	16108.00

(2 of 2)

Using Basic Macros

A macro is an abbreviation for a complex expression. For example, you can define a short, meaningful name for a numeric code and reference the code by its macro name rather than by a string of digits. In the same way, you can define a macro for a complete set of conditions and reference those conditions in a query with the macro name. The set of conditions might be a complete SELECT statement or a specific clause in a SELECT statement, for example.

A macro name is a character string that begins with a letter and does not exceed 128 characters. The database server is not sensitive to case: *share* and *SHARE* are equivalent. RISQL keywords cannot be used as macro names.

CREATE MACRO Syntax

```
CREATE MACRO macro_name AS definition;
```

macro_name A unique name that you specify in an SQL statement to call the macro definition.

definition A complete or partial SQL statement. Only one complete SQL statement can occur in the definition.

Existing macros must be dropped before a macro of the same name can be defined:

```
DROP MACRO macro_name;
```

The CREATE MACRO and DROP MACRO statements have additional optional parameters. For details, refer to the *SQL Reference Guide*.

About the Query

The macro **tea_products** is based on the knowledge that **Classkey** values 2 and 5 always refer to bulk tea and packaged tea products, respectively. The **Classkey** values are queried from the **Product** table rather than the **Class** table to simplify the joins in the query; a CASE expression is used to convert the **Classkey** values to meaningful text values.

The query calculates the 1999 sales totals for all tea products using the macro name. When the database server interprets the query, the macro name is replaced with the character string defined in the CREATE MACRO statement.

The parentheses around the macro definition are required in the example query. They force the correct evaluation of the logical operators defined in the macro.

Embedded Macros

Question

What were the total sales of tea products during 1999?

CREATE MACRO Statements

```
create macro case_tea as
   case pt.classkey when 2 then 'Bulk Tea'
      when 5 then 'Pkg Tea'
      end as class;
create macro tea_totals as
   select prod_name, case_tea, sum(dollars) as sales_99
   from product pt join sales sa
      on pt.classkey = sa.classkey and pt.prodkey = sa.prodkey
      join period pd on pd.perkey = sa.perkey
   where tea_products
      and year = 1999
   group by prod_name, class
   order by sales_99 desc;
```

Example Query

```
tea_totals;
```

Result

Prod_Name	Class	Sales_99
Darjeeling Special	Bulk Tea	80610.50
Darjeeling Special	Pkg Tea	51266.00
Assam Gold Blend	Bulk Tea	42329.00
Darjeeling Number 1	Bulk Tea	34592.75
Irish Breakfast	Bulk Tea	27763.75

(1 of 2)

Result

Prod_Name	Class	Sales_99
Assam Gold Blend	Pkg Tea	27192.50
English Breakfast	Bulk Tea	25848.00
Breakfast Blend	Bulk Tea	24594.00
Darjeeling Number 1	Pkg Tea	24232.00
Earl Grey	Bulk Tea	23269.50
Special Tips	Bulk Tea	22326.00
Assam Grade A	Bulk Tea	21964.00
Gold Tips	Bulk Tea	21584.50
Irish Breakfast	Pkg Tea	20084.00
English Breakfast	Pkg Tea	18955.00
Breakfast Blend	Pkg Tea	17031.50
Gold Tips	Pkg Tea	16783.25
Special Tips	Pkg Tea	16773.25
Assam Grade A	Pkg Tea	16724.00
Earl Grey	Pkg Tea	16108.00

(2 of 2)

Using Embedded Macros

An *embedded macro* is a macro that occurs within the definition of another macro.

About the Query

The CREATE MACRO statements in this example define two macros: **case_tea** and **tea_totals**.

- The first macro contains a CASE expression that replaces each Classkey with a meaningful class type, the same CASE expression that was used in the FROM clause of the query on page 6-4.
- The second macro is a complete SELECT statement that contains two embedded macros: **case_tea** and **tea_products** (defined on page 6-4).

To execute the macro **tea_totals**, enter the macro name:

```
tea_totals;
```

The result set is identical to the one returned by the previous example on page 6-4.

Usage Notes

Macros can be embedded more than one level down.

A macro definition can contain the name of a defined macro, but it cannot contain another macro definition.

Macros with Parameters

Question

What were the total sales of tea products during any given year?

CREATE MACRO Statement

```
create macro tea_sales(yr) as
   select year, prod_name, case_tea,
      sum(dollars) as us_sales
   from product pt join sales sa
      on pt.classkey = sa.classkey and pt.prodkey = sa.prodkey
      join period pd on pd.perkey = sa.perkey
   where tea_products
      and year = yr
   group by year, prod_name, class
   order by us_sales desc;
```

Example Query

```
tea_sales(1998);
```

Result

Year	Prod_Name	Class	US_Sales
1998	Darjeeling Special	Bulk Tea	75582.00
1998	Darjeeling Special	Pkg Tea	51625.00
1998	Assam Gold Blend	Bulk Tea	43091.00
1998	Darjeeling Number 1	Bulk Tea	36442.00
1998	Assam Gold Blend	Pkg Tea	28328.00
1998	Irish Breakfast	Bulk Tea	27440.75
1998	English Breakfast	Bulk Tea	27071.00
1998	Darjeeling Number 1	Pkg Tea	25841.25
1998	Earl Grey	Bulk Tea	24721.00
1998	Breakfast Blend	Bulk Tea	24689.25
1998	Gold Tips	Bulk Tea	23181.25
1998	Special Tips	Bulk Tea	22712.25
1998	Assam Grade A	Bulk Tea	22418.00
1998	Irish Breakfast	Pkg Tea	21318.25
1998	Breakfast Blend	Pkg Tea	17606.25
1998	English Breakfast	Pkg Tea	17310.00
1998	Assam Grade A	Pkg Tea	16787.00
1998	Earl Grey	Pkg Tea	16416.00
1998	Special Tips	Pkg Tea	15883.75
1998	Gold Tips	Pkg Tea	15732.50

Using Macros with Parameters

A macro can be generalized with one or more parameters, which can be changed each time the macro is executed. For example, a macro can be written with a parameter for *year* so that the same macro retrieves values for any year stored in the database. Similarly, a macro that contains parameters for product markets can retrieve data for any specified market.

CREATE MACRO Statement

Define a parameterized macro with the following command:

```
CREATE MACRO macro_name([parameter [, parameter] …]) AS
definition;
```

macro_name	A unique name that refers to the macro definition.
parameter	A value that customizes a generic macro. It can be changed each time the macro is used.
definition	A complete or partial SQL statement. Only one complete SQL statement can occur in the definition.

When you call a parameterized macro in a SELECT statement, you must include a value for each parameter defined in the CREATE MACRO statement.

About the Query

The CREATE MACRO statement defines a SELECT statement that contains a parameter for year (*yr*). When the macro **tea_sales(1998)** is executed, the database server replaces each occurrence of the parameter *yr* with **1998**. This query can be executed for any year for which sales data exists in the database (1998, 1999, or 2000 for the **Aroma** database).

Multiparameter Macros

Question

What were the best-selling products in a given location in a given year?

CREATE MACRO Statement

```
create macro top_rank(yr, locn, nbr) as
   select prod_name, city, year, sum(dollars) as sales,
      rank(sum(dollars)) as ranking
   from product pt join sales sa
      on pt.prodkey = sa.prodkey and pt.classkey = sa.classkey
      join period pd on pd.perkey = sa.perkey
      join store se on se.storekey = sa.storekey
   where city = locn and year = yr
   group by prod_name, city, year
   when rank(sum(dollars)) <= nbr;
```

Example Query 1 and Result

```
top_rank(1998, 'Los Angeles', 5);
```

Prod_Name	City	Year	Sales	Ranking
Xalapa Lapa	Los Angeles	1998	14930.00	1
Demitasse Ms	Los Angeles	1998	14402.25	2
Ruby's Allspice	Los Angeles	1998	14339.00	3
Aroma Roma	Los Angeles	1998	14253.25	4
Expresso XO	Los Angeles	1998	13179.50	5

Example Query 2 and Result

```
top_rank(1999, 'San Jose', 1);
```

Prod_Name	City	Year	Sales	Ranking
Demitasse Ms	San Jose	1999	32887.75	1

Example Query 3 and Result

```
top_rank(2000, 'Hartford', 3);
```

Prod_Name	City	Year	Sales	Ranking
NA Lite	Hartford	2000	5061.00	1
Cafe Au Lait	Hartford	2000	4665.00	2
Xalapa Lapa	Hartford	2000	4610.00	3

Macros with Multiple Parameters

A macro can have multiple parameters. For example, you can define a macro with parameters for year, region, and product name.

About the Query

This CREATE MACRO statement defines a SELECT statement that contains three parameters:

```
(yr, locn, nbr)
```

They represent the year, the city (or location), and the maximum number of rank values to be returned.

Query 1 retrieves the five best-selling products in Los Angeles during 1998:

```
top_rank(1998, 'Los Ang%', 5);
```

When the macro is executed, the database server replaces each occurrence of the three parameters with 1998, Los Angeles, and 5. Queries 2 and 3 show results for other years, locations, and rankings.

Usage Notes

Time periods such as days, weeks, months, quarters, and years are good candidates for parameters. So are product names, brands, trademarks, and suppliers.

The RANK function is discussed in detail in Chapter 3, "Data Analysis."

There are two Aroma stores in San Jose, but only one in Hartford and Los Angeles; the results in Query 2 represent the sum of dollars for both San Jose stores.

The sales figures in the result set for Query 3 are significantly smaller because the **Aroma** database contains sales figures for only the first quarter of 2000 but for all four quarters of 1998 and 1999.

Comparisons

Question

How do the monthly sales of Lotta Latte in San Jose compare during the first quarters of 1999 and 2000 in terms of both dollars and quantities?

CREATE MACRO Statement

```
create macro lotta_sales(facts, yr) as (
   select sum(facts)
      from store t natural join sales s
         natural join product p
         natural join period d
      where d.month = e.month
         and d.year = e.yr
         and p.prod_name = q.prod_name
         and t.city = u.city);
```

Example Query 1 and Result

```
select q.prod_name, e.month, sum(dollars) as sales_99,
   lotta_sales(dollars, year+1) as sales_00
from store u natural join product q natural join period e
   natural join sales l
where qtr = 'Q1_99'
   and prod_name like 'Lotta Latte%'
   and city like 'San J%'
group by q.prod_name, e.month, e.year, u.city;
```

Prod_Name	Month	Sales_99	Sales_00
Lotta Latte	JAN	1611.00	3475.00
Lotta Latte	FEB	3162.50	2409.50
Lotta Latte	MAR	2561.50	2831.50

Example Query 2 and Result

```
select q.prod_name, e.month, sum(dollars) as sales_99,
   lotta_sales(dollars, year+1) as sales_00,
   lotta_sales(quantity, year) as qty_99,
   lotta_sales(quantity, year+1) as qty_00
from store u natural join product q natural join period e
   natural join sales l
where qtr = 'Q1_99'
   and prod_name like 'Lotta Latte%'
   and city like 'San J%'
group by q.prod_name, e.month, e.year, u.city;
```

Prod_Name	Month	Sales_99	Sales_00	Qty_99	Qty_00
Lotta Latte	JAN	1611.00	3475.00	197	426
Lotta Latte	FEB	3162.50	2409.50	391	298
Lotta Latte	MAR	2561.50	2831.50	314	348

Using Comparison Macros

A query is often built from a basic block of instructions that is repeated several times with minor variations. These variations are often good candidates for macro parameters. For example, a query that compares sales during the current year with sales during the previous year must contain similar blocks of instructions: One block retrieves sales for the current year, the other for the previous year. A macro that contains a parameter for *year* reduces the number of instructions you must enter manually.

About the Query

In queries 1 and 2, the main query retrieves the monthly sales of Lotta Latte in San Jose during the first quarter of 1999 and the macro (a subquery) retrieves the corresponding figures for 2000.

The following macro subquery can retrieve one of two types of facts, dollars or quantity, for a specified year:

```
lotta_sales(facts, yr)
```

The **Sales** table contains additive columns for dollars and quantity only; a production database would probably contain many more types of facts.

In Query 1, the macro references the **Dollars** column of the **Sales** table and the year 2000:

```
lotta_sales(dollars, year+1)
```

The following expression evaluates to 2000 because the WHERE clause constraint in the main query refers to the year 1999:

```
year+1
```

This macro might not seem worth the effort to design until you begin to build more complex queries with it. For example, in Query 2, the macro is referenced three times, producing three different columns in the result set.

Usage Notes

This macro is based on a correlated subquery defined in the select list; however, equivalent subqueries defined in the FROM clause often result in faster performance, as described in Chapter 4, “Comparison Queries.”

Share Comparisons

Question

What were the monthly sales of Lotta Latte in San Jose during the first three months of 2000 and 1999? What was the share (percent) for each month of the quarter in each year?

CREATE MACRO Statement

```
create macro lotta_qtr_sales(facts, yr) as
   (select sum(facts)
   from store t natural join sales s
      natural join product p
      natural join period d
   where substr(d.qtr,1,2) = substr(e.qtr,1,2)
      and d.year = e.yr
      and p.prod_name = q.prod_name
      and t.city = u.city);
```

Example Query

```
select q.prod_name, e.month, sum(dollars) as sales_99,
   dec(100*sales_99/lotta_qtr_sales(dollars, year),7,2) as
      share_qtr_99,
   lotta_sales(dollars, year+1) as sales_00,
   dec(100*sales_00/lotta_qtr_sales(dollars, year+1),7,2) as
      share_qtr_00
from store u natural join product q
   natural join period e
   natural join sales l
where qtr = 'Q1_99'
   and prod_name like 'Lotta Latte%'
   and city like 'San J%'
group by q.prod_name, e.month, e.qtr, e.year, u.city, sales_00;
```

Result

Prod_Name	Month	Sales_99	Share_Qtr_99	Sales_00	Share_Qtr_00
Lotta Latte	JAN	1611.00	21.96	3475.00	39.86
Lotta Latte	FEB	3162.50	43.11	2409.50	27.64
Lotta Latte	MAR	2561.50	34.92	2831.50	32.48

Using Share Comparison Macros

Macros can also simplify calculations. For example, if you have a macro that retrieves monthly sales for a product and another macro that calculates the sum of the sales for that product during the quarter or year, you can easily calculate the monthly share of the total sales for the quarter or year.

This share is expressed as a simple percentage calculation. For example:

```
100*(monthly_sales/quarterly_sales)
```

This kind of macro can be applied to other years as well.

About the Query

This query retrieves monthly sales of Lotta Latte in San Jose during the first quarters of 1999 and 2000 and calculates for each month the share of the quarter during these years. This query would be much longer and more difficult to understand without the three macros.

The following macro works the same in this query as in the previous example in this chapter:

```
lotta_sales(facts, yr)
```

The following macro is another subquery that calculates, in this case, quarterly sales *dollars* for the specified year:

```
lotta_qtr_sales(facts, yr)
```

Alternatively, the macro could be used to calculate quarterly sales *quantities*. The results of this macro are not displayed in the report but used as the source data for the two share calculations.

The SUBSTR function is used in the macro definition to correlate the **Qtr** column values based on their first two characters (Q1). This constraint is necessary because the **Qtr** values in the **Period** table are specific to each year (**Q1_99** versus **Q1_00**, for example). For more information about the SUBSTR function, refer to the *SQL Reference Guide*.

Usage Notes

The GROUP BY clause must include the **Sales_00** column, as well as the other nonaggregate columns that either appear in the select list or are referenced in the correlation conditions of the subqueries.

Change in Share

Question

When you compare the monthly sales of Lotta Latte in San Jose during the first quarter of 1999 and the first quarter of 2000:

- Did the sales figures for each month go up or down? By what percentage?
- Did the share-of-quarter percentage for each month go up or down? By what percentage?

Example Query

```
select q.prod_name, e.month, sum(dollars) as sales_99,
   lotta_sales(dollars, year+1) as sales_00,
   dec(100*((sales_00 - sales_99)/sales_99),7,2) as sales_chg,
   dec(100*
      ((sales_00/lotta_qtr_sales(dollars, year+1))
      -
      (sales_99/lotta_qtr_sales(dollars, year)))
      ,7,2)
      as share_chg
from store u natural join product q
   natural join period e natural join sales l
where e.year = 1999
   and e.qtr = 'Q1_99'
   and q.prod_name like 'Lotta Latte%'
   and u.city like 'San J%'
group by q.prod_name, e.month, e.qtr, e.year, u.city, sales_00;
```

Result

Prod_Name	Month	Sales_99	Sales_00	Sales_Chg	Share_Chg
Lotta Latte	JAN	1611.00	3475.00	115.70	17.90
Lotta Latte	FEB	3162.50	2409.50	-23.81	-15.47
Lotta Latte	MAR	2561.50	2831.50	10.54	-2.43

Using Macros That Calculate Change in Share

When analysts have macros at their disposal, they can think more about sales and markets and less about how to express their business questions with SQL.

For example, the change in monthly sales for a product over two years can be expressed as a percentage by using the following calculation:

```
100*((monthly_sales_00 - monthly_sales_99)/monthly_sales_99)
```

Similarly, the change in the product share for a quarter can be calculated as follows:

```
100*(monthly_sales_00/quarterly_sales_00)
-
(monthly_sales_99/quarterly_sales_99)
```

Neither percentage is difficult to calculate, but macros simplify the writing of a query that returns these percentages.

About the Query

The two previously defined macros (**lotta_sales** and **lotta_qtr_sales**) are used in the example query to calculate the percentage change in monthly sales for the Lotta Latte product, as well as the change in share-of-quarter for product sales in corresponding months from 1999 and 2000.

Views

Question

What were the sales totals and ranks by store for Assam Gold Blend tea in 1999?

CREATE VIEW Statement

```
create view tea_sales99
   as select prod_name, store_name, sum(dollars) as
      tea_dollars, rank(sum(dollars)) as tea_rank
   from sales natural join product
      natural join period
      natural join store
   where sales.classkey in (2, 5)
      and year = 1999
   group by prod_name, store_name;
```

Example Query

```
select prod_name, store_name, tea_dollars, tea_rank
from tea_sales99
where prod_name like 'Assam Gold%';
```

Result

Prod_Name	Store_Name	Tea_Dollars	Tea_Rank
Assam Gold Blend	Beans of Boston	6201.50	15
Assam Gold Blend	Beaches Brew	6080.00	16
Assam Gold Blend	Texas Teahouse	5422.50	17
Assam Gold Blend	Olympic Coffee Company	5350.50	18
Assam Gold Blend	Cupertino Coffee Supply	5277.00	19
Assam Gold Blend	Moroccan Moods	5178.50	20
Assam Gold Blend	Coffee Brewers	5151.00	21
Assam Gold Blend	Moulin Rouge Roasting	4977.00	22
Assam Gold Blend	East Coast Roast	4769.00	25
Assam Gold Blend	Miami Espresso	4506.50	28
Assam Gold Blend	Roasters, Los Gatos	4414.50	29
Assam Gold Blend	San Jose Roasting Company	4226.50	32
Assam Gold Blend	Instant Coffee	4190.50	33
Assam Gold Blend	Java Judy's	3776.50	40

Selecting from Views

Analysts might be interested in certain products or time periods only, rather than the full range of facts and dimensions stored in the database. You can create views, which are read-only tables that contain subsets of information from existing tables or views, to make access to the specific data you want to query both easier and faster.

CREATE VIEW Syntax

```
CREATE VIEW view_name AS query_expression
```

query_expression Any join or nonjoin query expression, as defined in the *SQL Reference Guide*.

To create a precomputed view, you must add another USING clause. For information about precomputed views, refer to the *Informix Vista User's Guide*.

About the Query

This view contains four columns:

- Product names (**Prod_Name**)
- Store names (**Store_Name**)
- Aggregated sales totals per store, per tea product for 1999 (**Tea_Totals**)
- Rankings based on the aggregated sales totals (**Tea_Rank**)

The query simply constrains on the **Prod_Name** column to return sales totals and ranks per store for Assam Gold Blend tea.

The following search condition ensures that only tea products are selected by the view:

```
where sales.classkey in (2, 5)
```

In the **Class** table, the **Classkey** values are meaningful and map to specific groups of products.

Usage Notes

This business question can be asked with or without creating the view; however, the view improves performance and simplifies the analyst's approach to writing queries.

Query expressions cannot contain ORDER BY clauses; therefore, it might not be practical to include an order-dependent display function in a CREATE VIEW statement. Because the RANK function is not order-dependent (unlike CUME, for example), it is used successfully in the example query. For detailed information about RISQL display functions, refer to the *SQL Reference Guide*.

INSERT INTO SELECT Statement

Question

Create a temporary table to hold daily and cumulative sales totals for clothing products. Issue a SELECT statement on the table to retrieve data only for stores in Los Angeles.

CREATE TEMPORARY TABLE Statement

```
create temporary table clothing_sales
(date date,
prod_name char(30),
city char(20),
dollars dec(7,2),
cume_tot integer);
```

INSERT Statement

```
insert into clothing_sales
   (date, prod_name, city, dollars, cume_tot)
      select date, prod_name, city, dollars, cume(dollars)
      from store s join sales l on s.storekey = l.storekey
         join period t on l.perkey = t.perkey
         join product p on l.classkey = p.classkey
            and l.prodkey = p.prodkey
         join class c on p.classkey = c.classkey
      where class_type = 'Clothing'
      order by date, city
      reset by date;


** INFORMATION ** (209) Rows inserted: 816.
```

Example Query

```
select date, prod_name, dollars, cume_tot
from clothing_sales
where city = 'Los Angeles'
   and extract(year from date) = 2000
order by date;
```

Result

Date	Prod_Name	Dollars	Cume_Tot
2000-01-08	Aroma t-shirt	197.10	308
2000-01-18	Aroma t-shirt	131.40	131
2000-01-18	Aroma baseball cap	135.15	266
2000-01-23	Aroma baseball cap	15.90	15
2000-02-01	Aroma t-shirt	175.20	175
2000-02-04	Aroma t-shirt	164.25	164
...			

Creating a Temporary Table

If you have resource or DBA authorization for a database, you can create a temporary table that contains the result set of a query. Temporary tables are useful when you want to perform repeated analysis on a result set without reprocessing the original query. For example, you can store the results of RISQL display functions in temporary tables and issue SELECT statements against them to further constrain the result data. When you use display functions to fill a temporary table, remember to order the data to ensure that the results of the display-function column are accurate.

INSERT INTO SELECT

```
INSERT INTO table_name select_statement
```

table_name A valid table name.

select_statement A complete or partial SELECT statement, as defined in the *SQL Reference Guide*.

About the Query

The example query shows how to create a temporary table named **Clothing_Sales**, insert daily and cumulative sales totals into it, and query it by issuing a standard SELECT statement.

The results of the query in the example could be retrieved with more limiting search conditions in a regular SELECT statement. However, creating a temporary table to store cumulative totals improves query performance when you are working with large fact tables.

Usage Notes

Temporary tables are removed from the database automatically when your SQL session ends. These tables are not visible to other users connected to the same database.

To create tables in a Red Brick Decision Server database, you must have resource or DBA authorization. Along with resource and DBA authorization comes INSERT privilege, which allows you to insert data into any tables you create. For a complete discussion of authorizations and privileges, refer to the *SQL Reference Guide*.

The CREATE TABLE statement for a temporary table must define columns that are of the same data type and size as columns defined in the base tables of the database. Otherwise, input data from the INSERT INTO…SELECT statement will be incompatible with columns in the temporary table.

Summary

This chapter showed how to simplify SQL statements with RISQL macros and how to create views and temporary tables with the CREATE VIEW, CREATE TEMPORARY TABLE, and INSERT INTO...SELECT statements.

CREATE MACRO Statement

```
CREATE MACRO macro_name(parameter [, parameter] … ) AS definition ;
```

A macro name is a character string that begins with a letter and does not exceed 128 characters. Macro names are not case sensitive. A RISQL keyword cannot be a macro name.

When you call a parameterized macro, you must include a value for each parameter defined in the CREATE MACRO statement.

CREATE VIEW Statement

```
CREATE VIEW view_name AS query_expression
```

CREATE TEMPORARY TABLE Statement

```
CREATE TEMPORARY TABLE table_name (column_definitions)
```

INSERT INTO SELECT Statement

```
INSERT INTO table_name select_statement
```

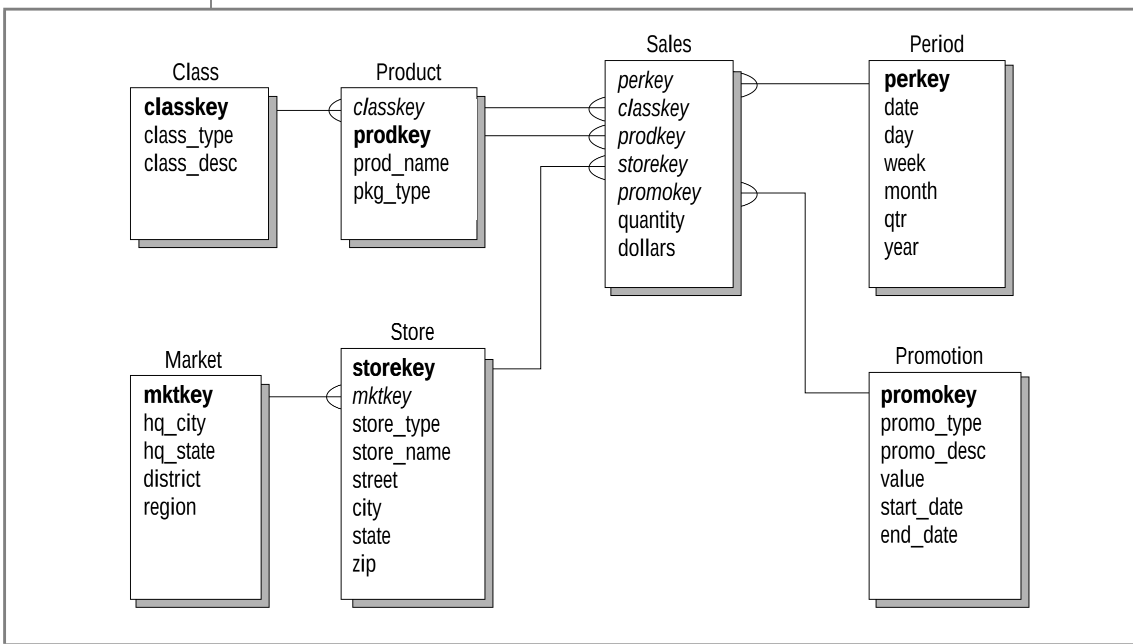
The Complete Aroma Database

Appendix A describes all of the tables in the **Aroma** database, which consists of two schemas—a simple star schema for retail sales information and a multistar schema for purchasing information.

Most of the examples in this document use the tables in the retail schema. The purchasing tables are used in a few examples that require a more flexible schema for adequate illustration.

Aroma Database—Retail Schema

Most of the examples in this guide are based on data from the basic **Aroma** database, which tracks daily retail sales in stores owned by the Aroma Coffee and Tea Company. The following figure illustrates this basic schema.



The crow's-feet in this diagram indicate a one-to-many relationship between the two tables. For example, each distinct value in the **Perkey** column of the **Period** table can occur only once in that table but many times in the **Sales** table.

Basic Aroma Schema

The following tables make up the basic **Aroma** database:

Period	Defines time intervals such as days, months, and years.
Class	Defines classes of products sold at retail stores.
Product	Defines individual products sold at retail stores, including bulk and packaged coffee and tea, coffee machines, and so on.
Market	Defines the geographical markets of the business.
Store	Defines individual retail stores owned and operated by the Aroma Coffee and Tea Company.
Promotion	Defines the types, durations, and values of promotions run on different products.
Sales	Contains the sales figures for Aroma products during time periods at various stores.

The **Period**, **Class**, **Product**, **Market**, **Store**, and **Promotion** tables are examples of typical business dimensions: They are small and contain descriptive data that is familiar to users.

The **Sales** table is a good example of a fact table: It contains thousands of rows, and its largely additive information is accessed in queries by joins to the dimension tables it references.

Sample Data from the Class and Product Tables

Query

```
select * from class;
```

Result

Classkey	Class_Type	Class_Desc
1	Bulk_beans	Bulk coffee products
2	Bulk_tea	Bulk tea products
3	Bulk_spice	Bulk spices
4	Pkg_coffee	Individually packaged coffee products
5	Pkg_tea	Individually packaged tea products
6	Pkg_spice	Individually packaged spice products
7	Hardware	Coffee mugs, teapots, spice jars, espresso machines
8	Gifts	Samplers, gift boxes and baskets, etc.
12	Clothing	T-shirts, caps, etc.

Query

```
select * from product;
```

Result

Classkey	Prodkey	Prod_Name	Pkg_Type
1	0	Veracruzano	No pkg
1	1	Xalapa Lapa	No pkg
1	10	Colombiano	No pkg
1	11	Expresso XO	No pkg
1	12	La Antigua	No pkg
1	20	Lotta Latte	No pkg
1	21	Cafe Au Lait	No pkg
1	22	NA Lite	No pkg
1	30	Aroma Roma	No pkg
1	31	Demitasse Ms	No pkg
2	0	Darjeeling Number 1	No pkg
2	1	Darjeeling Special	No pkg
2	10	Assam Grade A	No pkg
...			

The Class and Product Tables

The **Product** table describes the products defined in the **Aroma** database. The **Class** table describes the classes to which those products belong.

If a dimension table contains *foreign-key* columns that reference other dimension tables, the referenced tables are called *outboard* or *outrigger* tables. The **Classkey** column of the **Product** table is a foreign-key reference to the **Class** table, so the **Class** table is an outboard table.

Column Descriptions: Class Table

Column Name	Contents
classkey	Integer that identifies exactly one row in the **Class** table. **Classkey** is the primary key.
class_type	Character string that identifies a group of products.
class_desc	Character string that describes a group of products.

Column Descriptions: Product Table

Column Name	Contents
classkey	Foreign-key reference to the **Class** table.
prodkey	Integer that combines with the **Classkey** value to identify exactly one row in the **Product** table. **Classkey** / **Prodkey** is a two-column primary key.
prod_name	Character string that identifies a product. The database contains 59 products. A fully populated database would have many more. Although some Aroma products have the same name, they belong to different classes and have different package types.
pkg_type	Character string that identifies the type of packaging for each product.

Sample Data from the Store and Market Tables

Query

```
select * from market;
```

Result

Mktkey	HQ_City	HQ_State	District	Region
1	Atlanta	GA	Atlanta	South
2	Miami	FL	Atlanta	South
3	New Orleans	LA	New Orleans	South
4	Houston	TX	New Orleans	South
5	New York	NY	New York	North
...				

Query

```
select * from store;
```

Result

Storekey	Mktkey	Store_Type	Store_Name	Street	City	State	Zip
1	14	Small	Roasters, Los Gatos	1234 University Ave	Los Gatos	CA	95032
2	14	Large	San Jose Roasting	5678 Bascom Ave	San Jose	CA	95156
3	14	Medium	Cupertino Coffee	987 DeAnza Blvd	Cupertino	CA	97865
4	3	Medium	Moulin Rouge	898 Main Street	New Orleans	LA	70125
5	10	Small	Moon Pennies	98675 University	Detroit	MI	48209
6	9	Small	The Coffee Club	9865 Lakeshore Bl	Chicago	IL	06060
...							

Some columns have been truncated to fit on the page.

The Market and Store Tables

The **Store** table defines the stores that sell Aroma products. The **Market** table describes the U.S. markets to which each store belongs. Each market is identified by a major metropolitan city. The **Market** table is an outboard table, like the **Class** table.

Market Table—Column Descriptions

Column Name	Contents
mktkey	Integer that identifies exactly one row in the **Market** table. **Mktkey** is the primary key.
hq_city	Character string that identifies a city. The **Market** table defines 17 cities. A fully populated database could have thousands.
state	Character string that identifies a state.
district	Character string that identifies a district based on a major metropolitan city. A global database would contain countries and nations or other geographic dimensions.
region	Character string that identifies a region. The **Market** table defines only four regions for the entire United States. A comprehensive database would include numerous regions and probably more districts within a region.

Store Table—Column Descriptions

Column Name	Contents
storekey	Integer that identifies exactly one row in the **Store** table. **Storekey** is the primary key.
mktkey	Foreign-key reference to the **Market** table.
store_type	Character string that identifies stores by size.
store_name	Character string that identifies a store by name.
street, city, state, zip	Columns that identify the address of each store.

Sample Data from the Period Table

Query

```
select * from period;
```

Result

Perkey	Date	Day	Week	Month	Qtr	Year
1	1998-01-01	TH	1	JAN	Q1_98	1998
2	1998-01-02	FR	1	JAN	Q1_98	1998
3	1998-01-03	SA	1	JAN	Q1_98	1998
4	1998-01-04	SU	2	JAN	Q1_98	1998
5	1998-01-05	MO	2	JAN	Q1_98	1998
6	1998-01-06	TU	2	JAN	Q1_98	1998
7	1998-01-07	WE	2	JAN	Q1_98	1998
8	1998-01-08	TH	2	JAN	Q1_98	1998
9	1998-01-09	FR	2	JAN	Q1_98	1998
10	1998-01-10	SA	2	JAN	Q1_98	1998
11	1998-01-11	SU	3	JAN	Q1_98	1998
12	1998-01-12	MO	3	JAN	Q1_98	1998
13	1998-01-13	TU	3	JAN	Q1_98	1998
14	1998-01-14	WE	3	JAN	Q1_98	1998
15	1998-01-15	TH	3	JAN	Q1_98	1998
16	1998-01-16	FR	3	JAN	Q1_98	1998
17	1998-01-17	SA	3	JAN	Q1_98	1998

(1 of 2)

Perkey	Date	Day	Week	Month	Qtr	Year
18	1998-01-18	SU	4	JAN	Q1_98	1998
19	1998-01-19	MO	4	JAN	Q1_98	1998
20	1998-01-20	TU	4	JAN	Q1_98	1998
...						

(2 of 2)

The Period Table

The **Period** table defines daily, weekly, monthly, quarterly, and yearly time periods for 1998 and 1999 and the first quarter of 2000.

Column Descriptions

Column Name	Contents
perkey	Integer that identifies exactly one row in the **Period** table. **Perkey** is the primary key.
date	Date value that identifies each day from January 1, 1998 through March 31, 2000.
day	Character-string abbreviation of the day of the week.
week	Integer that identifies each week of each year by number (1 through 53, each new week starting on a Sunday).
month	Character-string abbreviation of the name of each month.
qtr	Character string that uniquely identifies each quarter (for example, Q1_98, Q3_99).
year	Integer that identifies the year.

Sample Data from the Promotion Table

Query

```
select * from promotion;
```

Result

Promokey	Promo_Type	Promo_Desc	Value	Start_Date	End_Date
0	1	No promotion	0.00	9999-01-01	9999-01-01
1	100	Aroma catalog coupon	1.00	1998-01-01	1998-01-31
2	100	Aroma catalog coupon	1.00	1998-02-01	1998-02-28
3	100	Aroma catalog coupon	1.00	1998-03-01	1998-03-31
4	100	Aroma catalog coupon	1.00	1998-04-01	1998-04-30
5	100	Aroma catalog coupon	1.00	1998-05-01	1998-05-31
6	100	Aroma catalog coupon	1.00	1998-06-01	1998-06-30
7	100	Aroma catalog coupon	1.00	1998-07-01	1998-07-31
8	100	Aroma catalog coupon	1.00	1998-08-01	1998-08-31
9	100	Aroma catalog coupon	1.00	1998-09-01	1998-09-30
10	100	Aroma catalog coupon	1.00	1998-10-01	1998-10-31
11	100	Aroma catalog coupon	1.00	1998-11-01	1998-11-30
12	100	Aroma catalog coupon	1.00	1998-12-01	1998-12-31
13	100	Aroma catalog coupon	1.00	1999-01-01	1999-01-31
14	100	Aroma catalog coupon	1.00	1999-02-01	1999-02-28
15	100	Aroma catalog coupon	1.00	1999-03-01	1999-03-31
16	100	Aroma catalog coupon	1.00	1999-04-01	1999-04-30

(1 of 2)

Promokey	Promo_Type	Promo_Desc	Value	Start_Date	End_Date
17	100	Aroma catalog coupon	1.00	1999-05-01	1999-05-31
18	100	Aroma catalog coupon	1.00	1999-06-01	1999-06-30
19	100	Aroma catalog coupon	1.00	1999-07-01	1999-07-31
20	100	Aroma catalog coupon	1.00	1999-08-01	1999-08-31
...					

(2 of 2)

The Promotion Table

The **Promotion** table is a dimension table that describes promotions that are run on different products during different time periods. Promotion tables are sometimes referred to as *condition* tables because they indicate the conditions under which goods are sold.

Column Descriptions

Column Name	Contents
promokey	Integer that identifies exactly one row in the **Promotion** table. **Promokey** is the primary key.
promo_type	Integer that identifies the promotion by number (or code).
promo_desc	Character string that describes the promotion type.
value	Decimal number that represents the dollar value of the promotion, such as a price reduction or the value of a coupon.
start_date, end_date	Date values that indicate when each promotion begins and ends.

Sample Data from the Sales Table

Query

```
select * from sales;
```

Result

Perkey	Classkey	Prodkey	Storekey	Promokey	Quantity	Dollars
2	2	0	1	116	8	34.00
2	4	12	1	116	9	60.75
2	1	11	1	116	40	270.00
2	2	30	1	116	16	36.00
2	5	22	1	116	11	30.25
2	1	30	1	116	30	187.50
2	1	10	1	116	25	143.75
2	4	10	2	0	12	87.00
2	4	11	2	0	14	115.50
2	2	22	2	0	18	58.50
2	4	0	2	0	17	136.00
2	5	0	2	0	13	74.75
2	4	30	2	0	14	101.50
2	2	10	2	0	18	63.00
2	1	22	3	0	11	99.00
2	6	46	3	0	6	36.00
2	5	12	3	0	10	40.00

(1 of 2)

Perkey	Classkey	Prodkey	Storekey	Promokey	Quantity	Dollars
2	1	11	3	0	36	279.00
2	5	1	3	0	11	132.00
2	5	10	3	0	12	48.00
...						

(2 of 2)

The Sales Table

The **Sales** table is a *fact table*; as such, it is by far the largest table in the **Aroma** database and its data is stored in two distinct areas of the database (known as *segments*). For information about segments, refer to the *Administrator's Guide*. The **Sales** table is large compared with the other Aroma tables, but small compared with typical fact tables at customer sites, which usually contain millions of rows.

Multipart Primary Key

The **Sales** table contains a multipart primary key: Each of its five columns is a *foreign-key* reference to the primary key of another table:

```
perkey, classkey, prodkey, storekey, promokey
```

This primary key links the **Sales** table data to the **Period**, **Product**, **Store**, and **Promotion** dimensions.

To improve query performance, a STARindex™ structure is built on the composite primary key of the **Sales** table. The presence of the STAR index makes STARjoin™ processing possible when the retail tables are joined in queries. For detailed examples of queries that require joins, refer to Chapter 5, "Joins and Unions." For detailed information about STAR indexes, refer to the *Administrator's Guide*.

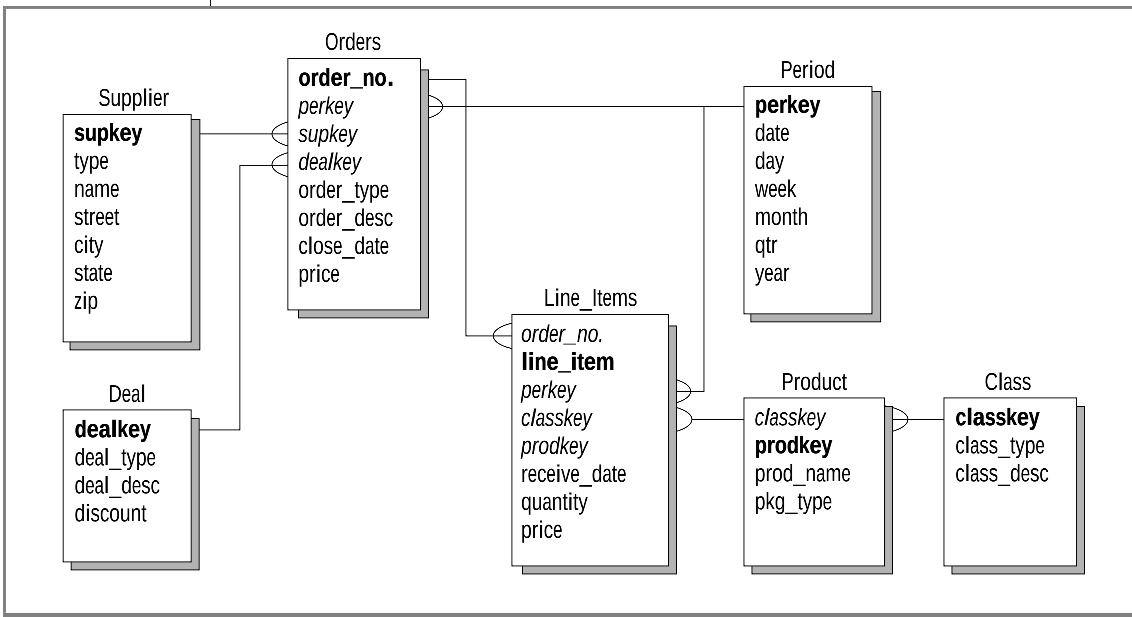
Column Descriptions

Column Name	Contents
perkey	Foreign-key reference to the **Period** table.
classkey	Foreign-key reference to the **Product** table.
prodkey	Foreign-key reference to the **Product** table.
storekey	Foreign-key reference to the **Store** table.
promokey	Foreign-key reference to the **Promotion** table.
quantity	Integer that represents the total quantity sold (per day).
dollars	Decimal number that represents dollar sales figures (per day).

Purchasing Schema of the Aroma Database

A few of the examples in this guide are based on tables used to track product orders that the Aroma Company receives from its suppliers. This purchasing schema uses the same **Product**, **Class**, and **Period** dimensions as the retail schema but has two dimensions of its own, **Deal** and **Supplier**. The **Line_Items** and **Orders** tables both contain facts, but the **Orders** table can also be queried as a dimension table referenced by the **Line_Items** table.

The following figure illustrates the tables in the purchasing schema.



Multistar Schema

The primary keys of the **Line_Items** and **Orders** tables do not match the set of their respective dimension-table foreign keys. Any given combination of dimension table primary keys can point to more than one row in these fact tables; this type of table is known as a *multistar* fact table or *data list*.

For example, multiple order numbers in the **Orders** table can refer to the same set of **Supplier**, **Deal**, and **Period** characteristics:

Order_No	Perkey	Supkey	Dealkey
3699	817	1007	0
3700	817	1007	0

Purchasing Tables

The purchasing schema contains similar kinds of facts to those stored in the **Sales** table, prices and quantities. The prices are dollar values representing amounts paid to suppliers for whole orders or specific line items within orders. The quantities represent units of product ordered.

You can use this schema to ask interesting questions about the purchasing history of Aroma, for example, which suppliers give the best deal on which products, or which suppliers have the best record for closing orders.

The Aroma Company sells the same products throughout its stores as it orders through its suppliers; therefore, you can write queries that span both schemas to compare what was ordered with what was sold or to calculate simple profit margins.

The following tables make up the purchasing schema of the **Aroma** database:

Period	Defines time intervals such as days, months, and years.
Class	Defines classes of products, both sold at retail stores and ordered from suppliers.
Product	Defines individual products, both sold at retail stores and ordered from suppliers.
Supplier	Defines the suppliers of products ordered by the Aroma Company.
Deal	Defines the discount deals applied to orders by suppliers.
Line_Items	Contains the line-item detail information for product orders, including the price and quantity of each item on each order.
Orders	Contains information about product orders, such as the full price of each order, the types of products ordered, and so on.

The **Supplier** and **Deal** tables are exclusive to the purchasing schema and are referenced by the **Orders** table.

Tip: *The purchasing schema contains data for the first quarter of 2000 only.*

Sample Data from the Supplier and Deal Tables

Query

```
select * from supplier;
```

Result

Supkey	Type	Name	Street	City	State	Zip
1001	Bulk coffee	CB Imports	100 Church Stre	Mountain View	CA	94001
1002	Bulk tea	Tea Makers,	1555 Hicks Rd.	San Jose	CA	95124
...						

Some columns have been truncated to fit on the page.

Query

```
select * from deal;
```

Result

Dealkey	Deal_Type	Deal_Desc	Discount
0	1000	No deal	0.00
1	100	Orders over $10,000	100.00
2	100	Orders over $20,000	500.00
3	100	Supplier catalog coupon	50.00
4	100	Supplier catalog coupon	100.00
37	200	Supplier coffee special	75.00

(1 of 2)

Dealkey	Deal_Type	Deal_Desc	Discount
38	200	Supplier coffee special	50.00
39	200	Supplier tea special	40.00
40	200	Supplier tea special	20.00

(2 of 2)

The Supplier and Deal Tables

Column Descriptions: Supplier Table

Column Name	Contents
supkey	Integer that identifies exactly one row in the **Supplier** table. **Supkey** is the primary key.
type	Character string that indicates the type of products supplied.
name	Character string that identifies the supplier by name.
street, city, state, zip	Columns that identify the address of the supplier.

Column Descriptions: Deal Table

Column Name	Contents
dealkey	Integer that identifies exactly one row in the **Deal** table. **Dealkey** is the primary key.
deal_type	Integer that identifies the type of deal (a code number).
deal_desc	Character string that describes the type of deal.
discount	Decimal value that indicates the dollar amount of the deal applied to an order.

Shared Dimensions

The purchasing schema shares the **Period**, **Product**, and **Class** tables with the retail schema.

As well as querying the retail and purchasing schemas independently, you can pose some interesting questions that involve tables from both schemas. For example, you can join the **Sales** and **Line_Items** tables to compare quantities of products ordered with quantities of products sold. A query like this uses the shared dimensions to constrain products and periods.

Sample Data from the Orders and Line_Items Tables

Query

```
select * from orders;
```

Result

Order_No	Perkey	Supkey	Dealkey	Order_Type	Order_Desc	Close_Date	Price
3600	731	1001	37	Coffee	Whole coffee b	2000-01-07	1200.46
3601	732	1001	37	Coffee	Whole coffee b	2000-01-07	1535.94
3602	733	1001	0	Tea	Loose tea, bul	2000-01-07	780.00
3603	740	1001	39	Tea	Loose tea, bul	2000-01-21	956.45
3604	744	1005	0	Spice	Pre-packed spi	2000-01-16	800.66
3605	768	1003	2	Coffee	Whole-bean and	2000-02-12	25100.00
3606	775	1003	2	Coffee	Whole-bean and	2000-02-19	25100.00
3607	782	1003	2	Coffee	Whole-bean and	2000-02-25	25100.00

(1 of 2)

Order_No	Perkey	Supkey	Dealkey	Order_Type	Order_Desc	Close_Date	Price
3608	789	1003	2	Coffee	Whole-bean and	2000-03-03	30250.00
3609	796	1003	2	Coffee	Whole-bean and	2000-03-15	25100.00
...							

(2 of 2)

Query

```
select * from line_items;
```

Result

Order_No	Line_Item	Perkey	Classkey	Prodkey	Receive_Da	Qty	Price
3600	1	731	1	1	2000-01-07	40	180.46
3600	2	731	2	10	2000-01-07	150	300.00
3600	3	731	2	11	2000-01-07	80	240.00
3600	4	731	2	12	2000-01-07	150	240.00
3600	5	731	1	20	2000-01-07	60	240.00
3601	1	732	1	0	2000-01-07	60	240.00
3601	2	732	1	1	2000-01-07	60	240.00
3601	3	732	1	10	2000-01-07	60	240.00
3601	4	732	1	11	2000-01-07	60	240.00
3601	5	732	1	12	2000-01-07	60	240.00
3601	6	732	1	31	2000-01-07	70	335.94
3602	1	733	2	0	2000-01-08	70	130.00
3602	2	733	2	1	2000-01-08	70	130.00
...							

The Orders and Line_Items Tables

The **Orders** and **Line_Items** tables contain the purchasing facts. For more details about these tables, see page A-16.

Column Descriptions: Orders Table

Column Name	Contents
order_no	Integer that identifies exactly one row in the **Orders** table. Order_No is the primary key.
perkey	Foreign-key reference to the **Period** table.
supkey	Foreign-key reference to the **Supplier** table.
dealkey	Foreign-key reference to the **Deal** table.
order_type	Character string that defines the types of products on the order.
order_desc	Character string that describes the type of order.
close_date	Date value that identifies when the order was completed or closed.
price	Decimal value that indicates the full cost of the order.

Column Descriptions: Line_Items Table

Column Name	Contents
order_no	Integer that identifies exactly one row in the **Orders** table. **Order_No** is the primary key.
line_item	Integer that identifies each item listed on the order by number.
perkey	Foreign-key reference to the **Period** table.
classkey	Foreign-key reference to the **Product** table.
prodkey	Foreign-key reference to the **Product** table.

(1 of 2)

Column Name	Contents
receive_date	Date value that identifies when the line-item was received.
quantity	Integer that indicates the quantity of products ordered for each line-item.
price	Decimal value that indicates the cost of the line-item.

(2 of 2)

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

Symbols