Note: Before using this information and the product it supports, read the information in the appendix entitled "Notices."

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

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

© Copyright International Business Machines Corporation 1996, 2002. All rights reserved.

US Government User Restricted Rights—Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
# Table of Contents

## Introduction
- In This Introduction .................................................. 3
- About This Guide .................................................... 3
- Documentation Conventions ........................................... 5
- Customer Support ..................................................... 7
- Related Documentation ................................................ 9
- Additional Documentation ............................................ 11
- IBM Welcomes Your Comments ..................................... 12

## Chapter 1 Aroma, a Database for Decision Support
- In This Chapter ....................................................... 1-3
- Aroma Database Retail Schema ....................................... 1-4
- Basic Aroma Schema ................................................ 1-5
- Period, Product, and Class Dimensions .......................... 1-6
- Store, Market, and Promotion Dimensions .................... 1-7
- Sales Table .......................................................... 1-9
- About the Sales Facts ............................................... 1-11
- Commonly Asked Questions ........................................ 1-12
- Typical Data Warehousing Queries ................................. 1-13
- Summary ............................................................ 1-14

## Chapter 2 Basic Queries
- In This Chapter ........................................................ 2-3
- Using the SELECT Statement to Retrieve Data .................. 2-4
- Using SELECT List to Retrieve Specific Columns ............... 2-7
- Using the WHERE Clause to Retrieve Specific Rows .......... 2-10
- Using AND, NOT, and OR Connectives for Compound Conditions 2-13
- Using the AND Connective to Specify Complex Search Conditions 2-15
- Using the Greater-Than (>) and Less-Than or Equal-To (<=) Operators 2-17
Using the IN Comparison Predicate .......................... 2:19
Using the Percent Sign (%) Wildcard .......................... 2:21
Using Simple Joins ............................................. 2:23
Using the ORDER BY Clause ..................................... 2:26
Calculating Subtotals ............................................ 2:29
Using the SUM, AVG, MAX, MIN, COUNT Set Functions .... 2:32
Using Column Aliases ........................................... 2:35
Using the GROUP BY Clause to Group Rows ................ 2:37
Using the GROUP BY Clause to Produce Multiple Groups .... 2:39
Using the Division Operator (/) ................................. 2:43
Using the HAVING Clause to Exclude Groups ................. 2:46
Removing Rows That Contain NULLs, Zeroes, and Spaces .... 2:49
Summary .......................................................... 2:52

Chapter 3  
**Data Analysis**

In This Chapter ................................................. 3-3
Analytic Functions ............................................. 3-4
Cumulative Totals .............................................. 3-5
Resetting Cumulative Totals .................................. 3-8
Comparing Running Totals with OLAP ....................... 3-11
Moving Averages ................................................ 3-14
Moving Sums ..................................................... 3-17
Ranking Data .................................................... 3-20
Using the WHEN Clause ....................................... 3-23
Ranking Values in Groups: NTILE ............................ 3-25
Using the NTILE Function with a CASE Expression ....... 3-28
Using the TERTILE Function .................................. 3-32
Calculating Ratios .............................................. 3-35
Using the DATEADD Function ................................ 3-38
Using the DATEDIFF Function ................................. 3-42
Using the EXTRACT Function .................................. 3-44
Summary .......................................................... 3-47

Chapter 4  
**Comparison Queries**

In This Chapter ................................................. 4-3
Comparing Data with SQL ....................................... 4-4
Using CASE Expressions ....................................... 4-7
Using Subqueries in the FROM Clause ......................... 4-10
Performing Calculations and Comparisons ................... 4-13
Using Subqueries in the Select List ........................................ 4-15
Using Correlated Subqueries ............................................... 4-18
Using Cross-References ..................................................... 4-21
Calculating Percentages of Quarter and Year ....................... 4-24
Using Subqueries in the WHERE Clause ............................... 4-26
Using the ALL Comparison Predicate ................................. 4-28
Using the EXISTS Predicate ............................................... 4-30
Using the SOME or ANY Predicate ...................................... 4-33
Summary .............................................................................. 4-36

Chapter 5  Joins and Unions

In This Chapter ................................................................. 5-3
Join of Two Tables .......................................................... 5-3
Different Ways to Join Tables ............................................. 5-6
System Table Join ............................................................ 5-9
Self-Joins ............................................................................ 5-10
Outer Join of Two Tables .................................................... 5-14
Fact-to-Fact Join ................................................................ 5-17
Fact-to-Fact Join .............................................................. 5-20
OR Versus UNION ............................................................. 5-23
INTERSECT Operation ...................................................... 5-26
INTERSECT Operation Inside Subquery .............................. 5-28
EXCEPT Operation ............................................................ 5-30
Summary .............................................................................. 5-32

Chapter 6  Macros, Views, and Temporary Tables

In This Chapter ................................................................. 6-3
Basic Macros ....................................................................... 6-4
Embedded Macros ............................................................. 6-7
Macros with Parameters ................................................... 6-10
Multiparameter Macros ..................................................... 6-13
Comparisons ...................................................................... 6-15
Share Comparisons ............................................................ 6-18
Change in Share .................................................................. 6-20
Views .................................................................................. 6-22
INSERT INTO SELECT Statement ....................................... 6-25
Summary .............................................................................. 6-28
Appendix A

The Complete Aroma Database

Notices

Index
Introduction

In This Introduction ........................................ 3
About This Guide ........................................... 3
  Types of Users ........................................... 3
  Software Dependencies ................................. 4
Documentation Conventions ............................. 5
  Typographical Conventions ............................ 5
  Syntax Notation ......................................... 6
Customer Support ........................................... 7
  New Cases ................................................. 7
  Existing Cases ......................................... 8
  Troubleshooting Tips ................................. 8
Related Documentation ................................... 9
Additional Documentation ............................... 11
  Online Documents .................................... 11
  Printed Documents .................................... 11
  Online Help ............................................. 11
IBM Welcomes Your Comments .......................... 12
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 IBM Red Brick Warehouse, Version 6.2, as your database server.

IBM Red Brick Warehouse 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.”

The scripts that you use to install the demonstration database reside in the `redbrick_dir/sample_input` directory, where `redbrick_dir` is the IBM Red Brick Warehouse 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

Typographical Conventions

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

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEYWORD</td>
<td>All primary elements in a programming language statement (keywords) appear in uppercase letters in a serif font.</td>
</tr>
<tr>
<td><strong>italics</strong></td>
<td>Within text, new terms and emphasized words appear in italics.</td>
</tr>
<tr>
<td><strong>italics</strong></td>
<td>Within syntax and code examples, variable values that you are to specify appear in italics.</td>
</tr>
<tr>
<td><strong>boldface</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>monospace</strong></td>
<td>Information that the product displays and information that you enter appear in a monospace typeface.</td>
</tr>
<tr>
<td>KEYSTROKE</td>
<td>Keys that you are to press appear in uppercase letters in a sans serif font.</td>
</tr>
<tr>
<td>♦</td>
<td>This symbol indicates the end of one or more product- or platform-specific paragraphs.</td>
</tr>
<tr>
<td>→</td>
<td>This symbol indicates a menu item. For example, “Choose Tools→Options” means choose the Options item from the Tools menu.</td>
</tr>
</tbody>
</table>
**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.

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

If you have technical questions about IBM Red Brick Warehouse but cannot find the answer in the appropriate document, contact IBM 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)


New Cases

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

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

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

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

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

Troubleshooting Tips

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

- For SQL query problems, try to remove columns or functions or to restate WHERE, ORDER BY, or GROUP BY clauses until you can isolate the part of the statement causing the problem.

- For Table Management Utility load problems, verify the data type mapping between the source file and the target table to ensure compatibility. Try to load a small test set of data to determine whether the problem concerns volume or data format.

- For connectivity problems, issue the ping command from the client to the host to verify that the network is up and running. If possible, try another client tool to see if the same problem arises.
The standard documentation set for IBM Red Brick Warehouse includes the following documents.

<table>
<thead>
<tr>
<th>Document</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Administrator’s Guide</strong></td>
<td>Describes warehouse architecture, supported schemas, and other concepts relevant to databases. Procedural information for implementing and maintaining a database. Includes a description of the system tables and the configuration file.</td>
</tr>
<tr>
<td><strong>Client Installation and Connectivity Guide</strong></td>
<td>Includes procedures for installing ODBC, Red Brick JDBC Driver, RISQL Entry Tool, and RISQL Reporter on client systems. Describes how to access IBM Red Brick Warehouse using ODBC for C and C++ applications and JDBC for Java applications.</td>
</tr>
<tr>
<td><strong>IBM Red Brick Vista User’s Guide</strong></td>
<td>Describes the IBM Red Brick Vista aggregate computation and management system. Illustrates how Vista improves query performance by automatically rewriting queries to use aggregates, describes how the Advisor recommends the best set of aggregates based on data collected daily, and explains how aggregate tables are maintained when their detail tables are updated.</td>
</tr>
<tr>
<td><strong>Installation and Configuration Guide</strong></td>
<td>Provides installation and configuration information, as well as platform-specific material, about IBM Red Brick Warehouse. Customized for either UNIX and Linux platforms or Windows platforms.</td>
</tr>
</tbody>
</table>
## Related Documentation

<table>
<thead>
<tr>
<th>Document</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Messages and Codes Reference Guide</strong></td>
<td>Contains a complete listing of all informational, warning, and error messages generated by IBM Red Brick Warehouse products, including probable causes and recommended responses. Also includes event log messages that are written to the log files.</td>
</tr>
<tr>
<td><strong>Query Performance Guide</strong></td>
<td>This guide describes the determinants of query performance and shows how to tune a database for optimal query performance. Examples show how to evaluate query performance using Red Brick tools: SET STATS, Dynamic Statistic Tables, EXPLAIN, and the Query Performance Monitor.</td>
</tr>
<tr>
<td><strong>Release Notes</strong></td>
<td>Contains information pertinent to the current release that was unavailable when the documents were printed.</td>
</tr>
<tr>
<td><strong>RISQL Entry Tool and RISQL Reporter User’s Guide</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>This guide</strong></td>
<td>Provides an example-based review of SQL and introduction to the RISQL extensions, the macro facility, and Aroma, the sample database.</td>
</tr>
<tr>
<td><strong>Table Management Utility Reference Guide</strong></td>
<td>Describes the Table Management Utility, including activities related to loading, maintaining, and backing up data. Also includes information about data replication and the rb_cm copy management utility.</td>
</tr>
</tbody>
</table>

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 documents
- Printed documents
- Online help

**Online Documents**

A Documentation CD that contains IBM Red Brick Warehouse documents in electronic format is provided with your Red Brick products. You can copy the documentation to your computer or access it directly from the CD.

**Printed Documents**

To order printed documents, contact your sales representative.

**Online Help**

IBM provides online help with each graphical user interface (GUI) that displays information about those interfaces and the functions that they perform. Use the help facilities that each GUI provides to display the online help.
IBM Welcomes Your Comments

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

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

Send electronic mail to us at the following address:

comments@vnet.ibm.com

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

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

For immediate help with a technical problem, contact IBM Customer Support.
Aroma, a Database for Decision Support

In This Chapter ............................................. 1-3
Aroma Database Retail Schema .................. 1-4
Basic Aroma Schema ................................. 1-5
Period, Product, and Class Dimensions ....... 1-6
Store, Market, and Promotion Dimensions ... 1-7
Sales Table ................................................ 1-9
About the Sales Facts ............................... 1-11
Commonly Asked Questions ..................... 1-12
Typical Data Warehousing Queries .............. 1-13
Summary ................................................ 1-14
In This Chapter

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

All the examples in this document and in most of the IBM Red Brick Warehouse 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 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 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.
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 Vista query rewrite system to accelerate the performance of aggregate queries, refer to the IBM Red Brick 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:

<table>
<thead>
<tr>
<th>Perkey</th>
<th>Date</th>
<th>Day</th>
<th>Week</th>
<th>Month</th>
<th>Qtr</th>
<th>YEAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1998-01-01</td>
<td>TH</td>
<td>1</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>2</td>
<td>1998-01-02</td>
<td>FR</td>
<td>1</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>3</td>
<td>1998-01-03</td>
<td>SA</td>
<td>1</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>4</td>
<td>1998-01-04</td>
<td>SU</td>
<td>2</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>5</td>
<td>1998-01-05</td>
<td>MO</td>
<td>2</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>6</td>
<td>1998-01-06</td>
<td>TU</td>
<td>2</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
</tbody>
</table>
...

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:

<table>
<thead>
<tr>
<th>Classkey</th>
<th>Prodkey</th>
<th>Prod_Name</th>
<th>Pkg_Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>Veracruzano</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Xalapa Lapa</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>Colombiano</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>Espresso XO</td>
<td>No pkg</td>
</tr>
</tbody>
</table>

(1 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.

<table>
<thead>
<tr>
<th>Classkey</th>
<th>Class_Type</th>
<th>Class_Desc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bulk_beans</td>
<td>Bulk coffee products</td>
</tr>
<tr>
<td>2</td>
<td>Bulk_tea</td>
<td>Bulk tea products</td>
</tr>
<tr>
<td>3</td>
<td>Bulk_spice</td>
<td>Bulk spices</td>
</tr>
<tr>
<td>4</td>
<td>Pkg_coffee</td>
<td>Individually packaged coffee products</td>
</tr>
<tr>
<td>5</td>
<td>Pkg_tea</td>
<td>Individually packaged tea products</td>
</tr>
<tr>
<td>6</td>
<td>Pkg_spice</td>
<td>Individually packaged spice products</td>
</tr>
</tbody>
</table>

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.
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.

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

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

<table>
<thead>
<tr>
<th>Mktkey</th>
<th>Hq_city</th>
<th>Hq_state</th>
<th>District</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Atlanta</td>
<td>GA</td>
<td>Atlanta</td>
<td>South</td>
</tr>
<tr>
<td>2</td>
<td>Miami</td>
<td>FL</td>
<td>Atlanta</td>
<td>South</td>
</tr>
<tr>
<td>3</td>
<td>New Orleans</td>
<td>LA</td>
<td>New Orleans</td>
<td>South</td>
</tr>
<tr>
<td>4</td>
<td>Houston</td>
<td>TX</td>
<td>New Orleans</td>
<td>South</td>
</tr>
<tr>
<td>5</td>
<td>New York</td>
<td>NY</td>
<td>New York</td>
<td>North</td>
</tr>
</tbody>
</table>

...
**Promotion Table**

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

<table>
<thead>
<tr>
<th>Promokey</th>
<th>Promo_Type</th>
<th>Promo_Desc</th>
<th>Value</th>
<th>Start_Date</th>
<th>End_Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>No promotion</td>
<td>0.00</td>
<td>9999-01-01</td>
<td>9999-01-01</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1998-01-01</td>
<td>1998-01-31</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1998-02-01</td>
<td>1998-02-28</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1998-03-01</td>
<td>1998-03-31</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1998-04-01</td>
<td>1998-04-30</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1998-05-01</td>
<td>1998-05-31</td>
</tr>
</tbody>
</table>

...

**Sales Table**

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

<table>
<thead>
<tr>
<th>Perkey</th>
<th>Classkey</th>
<th>Prodkey</th>
<th>Storekey</th>
<th>Promokey</th>
<th>Quantity</th>
<th>Dollars</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>116</td>
<td>8</td>
<td>34.00</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>12</td>
<td>1</td>
<td>116</td>
<td>9</td>
<td>60.75</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>116</td>
<td>40</td>
<td>270.00</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>30</td>
<td>1</td>
<td>116</td>
<td>16</td>
<td>36.00</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>22</td>
<td>1</td>
<td>116</td>
<td>11</td>
<td>30.25</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>30</td>
<td>1</td>
<td>116</td>
<td>30</td>
<td>187.50</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>116</td>
<td>25</td>
<td>143.75</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>12</td>
<td>87.00</td>
</tr>
</tbody>
</table>

(1 of 2)
The primary-key column is a combination of values from five columns:

- perkey
- classkey
- prodkey
- storekey
- promokey

### Sales Table

<table>
<thead>
<tr>
<th>Perkey</th>
<th>Classkey</th>
<th>Prodkey</th>
<th>Storekey</th>
<th>Promokey</th>
<th>Quantity</th>
<th>Dollars</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>11</td>
<td>2</td>
<td>0</td>
<td>14</td>
<td>115.50</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>22</td>
<td>2</td>
<td>0</td>
<td>18</td>
<td>58.50</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>17</td>
<td>136.00</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>13</td>
<td>74.75</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>30</td>
<td>2</td>
<td>0</td>
<td>14</td>
<td>101.50</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>18</td>
<td>63.00</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>22</td>
<td>3</td>
<td>0</td>
<td>11</td>
<td>99.00</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>46</td>
<td>3</td>
<td>0</td>
<td>6</td>
<td>36.00</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>12</td>
<td>3</td>
<td>0</td>
<td>10</td>
<td>40.00</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>11</td>
<td>3</td>
<td>0</td>
<td>36</td>
<td>279.00</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>11</td>
<td>132.00</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>10</td>
<td>3</td>
<td>0</td>
<td>12</td>
<td>48.00</td>
</tr>
</tbody>
</table>

...
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. IBM Red Brick Warehouse 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 OLAP Functions or 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?
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 standard SQL OLAP functions and RISQL extensions to SQL provide a better solution because they are easy to use, reduce network traffic, and perform a broad range of calculations that execute quickly on the server.
This chapter briefly described the retail schema of the Aroma database and suggested some typical business questions that an IBM Red Brick Warehouse 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 an IBM Red Brick Warehouse 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.”
Basic Queries

In This Chapter ........................................ 2-3
Using the SELECT Statement to Retrieve Data .......... 2-4
Using SELECT List to Retrieve Specific Columns......... 2-7
Using the WHERE Clause to Retrieve Specific Rows ...... 2-10
Using AND, NOT, and OR Connectives for Compound Conditions 2-13
Using the AND Connective to Specify Complex Search Conditions 2-15
Using the Greater-Than (>), Less-Than or Equal-To (<=) Operators 2-17
Using the IN Comparison Predicate .................. 2-19
Using the Percent Sign (%) Wildcard .................. 2-21
Using Simple Joins .................................... 2-23
Using the ORDER BY Clause ............................. 2-26
Calculating Subtotals .................................. 2-29
Using the SUM, AVG, MAX, MIN, COUNT Set Functions .. 2-32
Using Column Aliases .................................. 2-35
Using the GROUP BY Clause to Group Rows .......... 2-37
Using the GROUP BY Clause to Produce Multiple Groups 2-39
Using the Division Operator (/) .......................... 2-43
Using the HAVING Clause to Exclude Groups ............ 2-46
Removing Rows That Contain NULLs, Zeroes, and Spaces 2-49
Summary ............................................ 2-52
In This Chapter

Through a series of simple examples, this chapter illustrates how to retrieve data from an IBM Red Brick Warehouse 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
```sql
select * from market;
```
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.

IBM Red Brick Warehouse 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
```sql
select district, region
from market;
```

Result

<table>
<thead>
<tr>
<th>District</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atlanta</td>
<td>South</td>
</tr>
<tr>
<td>Atlanta</td>
<td>South</td>
</tr>
<tr>
<td>New Orleans</td>
<td>South</td>
</tr>
<tr>
<td>New Orleans</td>
<td>South</td>
</tr>
<tr>
<td>New York</td>
<td>North</td>
</tr>
<tr>
<td>New York</td>
<td>North</td>
</tr>
<tr>
<td>Boston</td>
<td>North</td>
</tr>
<tr>
<td>Boston</td>
<td>North</td>
</tr>
<tr>
<td>Chicago</td>
<td>Central</td>
</tr>
<tr>
<td>Chicago</td>
<td>Central</td>
</tr>
<tr>
<td>Minneapolis</td>
<td>Central</td>
</tr>
<tr>
<td>Minneapolis</td>
<td>Central</td>
</tr>
<tr>
<td>San Francisco</td>
<td>West</td>
</tr>
</tbody>
</table>
Result

<table>
<thead>
<tr>
<th>District</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Francisco</td>
<td>West</td>
</tr>
<tr>
<td>San Francisco</td>
<td>West</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>West</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>West</td>
</tr>
</tbody>
</table>

(2 of 2)
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:

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

<table>
<thead>
<tr>
<th>District</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atlanta</td>
<td>South</td>
</tr>
<tr>
<td>Boston</td>
<td>North</td>
</tr>
<tr>
<td>Chicago</td>
<td>Central</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>West</td>
</tr>
<tr>
<td>Minneapolis</td>
<td>Central</td>
</tr>
</tbody>
</table>

(1 of 2)
Using the WHERE Clause to Retrieve Specific Rows

<table>
<thead>
<tr>
<th>District</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Orleans</td>
<td>South</td>
</tr>
<tr>
<td>New York</td>
<td>North</td>
</tr>
<tr>
<td>San Francisco</td>
<td>West</td>
</tr>
</tbody>
</table>

Question

What products are sold without packaging?

Example Query

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

<table>
<thead>
<tr>
<th>Prod_Name</th>
<th>Pkg_Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Veracruzano</td>
<td>No pkg</td>
</tr>
<tr>
<td>Xalapa Lapa</td>
<td>No pkg</td>
</tr>
<tr>
<td>Colombiano</td>
<td>No pkg</td>
</tr>
<tr>
<td>Expresso XO</td>
<td>No pkg</td>
</tr>
<tr>
<td>La Antigua</td>
<td>No pkg</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>No pkg</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>No pkg</td>
</tr>
<tr>
<td>NA Lite</td>
<td>No pkg</td>
</tr>
<tr>
<td>Aroma Roma</td>
<td>No pkg</td>
</tr>
<tr>
<td>Demitasse Ms</td>
<td>No pkg</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>No pkg</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>No pkg</td>
</tr>
<tr>
<td>Assam Grade A</td>
<td>No pkg</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>No pkg</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>No pkg</td>
</tr>
<tr>
<td>English Breakfast</td>
<td>No pkg</td>
</tr>
<tr>
<td>Irish Breakfast</td>
<td>No pkg</td>
</tr>
<tr>
<td>Special Tips</td>
<td>No pkg</td>
</tr>
<tr>
<td>Gold Tips</td>
<td>No pkg</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>No pkg</td>
</tr>
<tr>
<td>Ruby's Allspice</td>
<td>No pkg</td>
</tr>
<tr>
<td>Coffee Mug</td>
<td>No pkg</td>
</tr>
</tbody>
</table>

(1 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. IBM Red Brick Warehouse 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 following condition:

class_type = ‘Bulk_beans’

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 2-33.

---

Using AND, NOT, and OR Connectives for Compound Conditions

Question

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

Example Query

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

<table>
<thead>
<tr>
<th>Connective</th>
<th>Name</th>
<th>Order of Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Parentheses (force order of evaluation)</td>
<td>1</td>
</tr>
<tr>
<td>AND</td>
<td>And</td>
<td>3</td>
</tr>
<tr>
<td>NOT</td>
<td>Negation</td>
<td>2</td>
</tr>
<tr>
<td>OR</td>
<td>Or</td>
<td>4</td>
</tr>
</tbody>
</table>

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, IBM Red Brick Warehouse 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**

```sql
select store_type, store_name, city
from store
where (store_type = 'Large' or store_type = 'Small')
   and (city = 'Los Angeles' or city = 'San Jose');
```
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.
Using the Greater-Than (>) and Less-Than or Equal-To (<=) Operators

**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**

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>equal</td>
</tr>
<tr>
<td></td>
<td>less than</td>
</tr>
<tr>
<td></td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>not equal</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
</tbody>
</table>
**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:

```sql
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:

```sql
(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**

```sql
select hq_city, hq_state, district
from market
where district in
      ('Chicago', 'New York', 'New Orleans');
```
A simple condition can be expressed with the following SQL comparison predicates:

- **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:

```sql
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**

```sql
select district, hq_city
from market
where district like 'Min%';
```
Previous queries have expressed conditions that match complete character strings. With the LIKE predicate and the two wildcard characters, the percent sign (%) and the 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 'L_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

```sql
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

```sql
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

<table>
<thead>
<tr>
<th>Prod_Name</th>
<th>Store_Name</th>
<th>Day</th>
<th>Dollars</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easter Sampler Basket</td>
<td>Olympic Coffee Company</td>
<td>SA</td>
<td>150.00</td>
</tr>
</tbody>
</table>

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

```sql
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

<table>
<thead>
<tr>
<th>Prod_Name</th>
<th>Store_Name</th>
<th>Dollars</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>96.00</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>78.00</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>66.00</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>58.50</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>58.50</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>39.00</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>39.00</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>32.50</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Instant Coffee</td>
<td>48.00</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Instant Coffee</td>
<td>45.50</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Instant Coffee</td>
<td>42.00</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Instant Coffee</td>
<td>32.00</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Instant Coffee</td>
<td>24.00</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Instant Coffee</td>
<td>20.00</td>
</tr>
</tbody>
</table>

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 2-36.
Ordering the Result Table: ORDER BY Clause

**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

```sql
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

<table>
<thead>
<tr>
<th>Prod_Name</th>
<th>Store_Name</th>
<th>Dollars</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>96.00</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>78.00</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>66.00</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>58.50</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>58.50</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>39.00</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>39.00</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>32.50</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>NULL</td>
<td>467.50</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>Instant Coffee</td>
<td>207.00</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>Instant Coffee</td>
<td>168.00</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>Instant Coffee</td>
<td>149.50</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>Instant Coffee</td>
<td>144.00</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>Instant Coffee</td>
<td>138.00</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>Instant Coffee</td>
<td>132.00</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>Instant Coffee</td>
<td>96.00</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>Instant Coffee</td>
<td>69.00</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>Instant Coffee</td>
<td>60.00</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>Instant Coffee</td>
<td>60.00</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>Instant Coffee</td>
<td>48.00</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>NULL</td>
<td>1271.50</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Instant Coffee</td>
<td>48.00</td>
</tr>
</tbody>
</table>

(1 of 2)
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`.

Calculating Subtotals: BREAK BY Clause

<table>
<thead>
<tr>
<th>Prod_Name</th>
<th>Store_Name</th>
<th>Dollars</th>
</tr>
</thead>
<tbody>
<tr>
<td>Earl Grey</td>
<td>Instant Coffee</td>
<td>45.50</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Instant Coffee</td>
<td>42.00</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Instant Coffee</td>
<td>32.00</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Instant Coffee</td>
<td>24.00</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Instant Coffee</td>
<td>20.00</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>NULL</td>
<td>211.50</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>1950.50</td>
</tr>
</tbody>
</table>
**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 3-10.

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

```sql
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

<table>
<thead>
<tr>
<th>Sum</th>
<th>Avg</th>
<th>Max</th>
<th>Min</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>13706.50</td>
<td>171.33125000</td>
<td>376.00</td>
<td>39.00</td>
<td>80</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUM(expression)</td>
<td>Calculates the sum of all the values in expression.</td>
</tr>
<tr>
<td>SUM(DISTINCT expression)</td>
<td>Calculates the sum of distinct values in expression.</td>
</tr>
<tr>
<td>AVG(expression)</td>
<td>Calculates the average of all the values in expression.</td>
</tr>
<tr>
<td>AVG(DISTINCT expression)</td>
<td>Calculates the average of distinct values in expression.</td>
</tr>
<tr>
<td>MAX(expression)</td>
<td>Determines the maximum value in expression.</td>
</tr>
<tr>
<td>MIN(expression)</td>
<td>Determines the minimum value in expression.</td>
</tr>
</tbody>
</table>
### Using Set Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT(*)</td>
<td>Counts the number of rows returned.</td>
</tr>
<tr>
<td>COUNT(expression)</td>
<td>Counts the number of non-null values in expression.</td>
</tr>
<tr>
<td>COUNT(DISTINCT expression)</td>
<td>Counts the number of distinct non-null values in expression.</td>
</tr>
</tbody>
</table>

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 2-38.

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

```sql
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

<table>
<thead>
<tr>
<th>Dol_Sales</th>
<th>Avg_Sales</th>
<th>Max_Dol</th>
<th>Min_Dol</th>
<th>Num_Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>13706.50</td>
<td>171.33125000</td>
<td>376.00</td>
<td>39.00</td>
<td>80</td>
</tr>
</tbody>
</table>
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 2-38.

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

```sql
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

<table>
<thead>
<tr>
<th>District_City</th>
<th>Dol_Sales</th>
<th>Avg_Sales</th>
<th>Max_Sales</th>
<th>Min_Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atlanta</td>
<td>1378.30</td>
<td>35.34102564</td>
<td>98.55</td>
<td>4.00</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>711.60</td>
<td>30.93913043</td>
<td>98.55</td>
<td>9.95</td>
</tr>
<tr>
<td>San Francisco</td>
<td>410.45</td>
<td>25.65312500</td>
<td>54.75</td>
<td>5.00</td>
</tr>
</tbody>
</table>
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 Vista query rewrite system. For details, refer to the *IBM Red Brick 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 2-36.

---

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

(1 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**

```sql
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];
```
Nesting Grouped Results: GROUP BY Clause

**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

```sql
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;
```
<table>
<thead>
<tr>
<th>Prod_Name</th>
<th>Total_Sales</th>
<th>Total_Qty</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gold Tips</td>
<td>38913.75</td>
<td>11563</td>
<td>3.36</td>
</tr>
<tr>
<td>Special Tips</td>
<td>38596.00</td>
<td>11390</td>
<td>3.38</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>41137.00</td>
<td>11364</td>
<td>3.61</td>
</tr>
<tr>
<td>Assam Grade A</td>
<td>39205.00</td>
<td>10767</td>
<td>3.64</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>42295.50</td>
<td>10880</td>
<td>3.88</td>
</tr>
<tr>
<td>English Breakfast</td>
<td>44381.00</td>
<td>10737</td>
<td>4.13</td>
</tr>
<tr>
<td>Irish Breakfast</td>
<td>48759.00</td>
<td>11094</td>
<td>4.39</td>
</tr>
<tr>
<td>Coffee Mug</td>
<td>1054.00</td>
<td>213</td>
<td>4.94</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>62283.25</td>
<td>11539</td>
<td>5.39</td>
</tr>
<tr>
<td>Ruby's Allspice</td>
<td>133188.50</td>
<td>23444</td>
<td>5.68</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>71419.00</td>
<td>11636</td>
<td>6.13</td>
</tr>
<tr>
<td>Colombiano</td>
<td>188474.50</td>
<td>27548</td>
<td>6.84</td>
</tr>
<tr>
<td>Aroma Roma</td>
<td>203544.00</td>
<td>28344</td>
<td>7.18</td>
</tr>
<tr>
<td>La Antigua</td>
<td>197069.50</td>
<td>26826</td>
<td>7.34</td>
</tr>
<tr>
<td>Veracruzano</td>
<td>201230.00</td>
<td>26469</td>
<td>7.60</td>
</tr>
<tr>
<td>Expresso XO</td>
<td>224020.00</td>
<td>28558</td>
<td>7.84</td>
</tr>
<tr>
<td>Aroma baseball cap</td>
<td>15395.35</td>
<td>1953</td>
<td>7.88</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>217994.50</td>
<td>26994</td>
<td>8.07</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>213510.00</td>
<td>26340</td>
<td>8.10</td>
</tr>
<tr>
<td>Aroma Sounds Cassette</td>
<td>5206.00</td>
<td>620</td>
<td>8.39</td>
</tr>
<tr>
<td>Xalapa Lapa</td>
<td>251590.00</td>
<td>29293</td>
<td>8.58</td>
</tr>
<tr>
<td>NA Lite</td>
<td>231845.00</td>
<td>25884</td>
<td>8.95</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>Forces order of evaluation</td>
</tr>
<tr>
<td>+, −</td>
<td>Positive and negative</td>
</tr>
<tr>
<td>*, /</td>
<td>Multiplication and division</td>
</tr>
<tr>
<td>+, −</td>
<td>Addition and subtraction</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prod_Name</th>
<th>Total_Sales</th>
<th>Total_Qty</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demitasse Ms</td>
<td>282385.25</td>
<td>28743</td>
<td>9.82</td>
</tr>
<tr>
<td>Aroma t-shirt</td>
<td>20278.50</td>
<td>1870</td>
<td>10.84</td>
</tr>
<tr>
<td>Travel Mug</td>
<td>1446.35</td>
<td>133</td>
<td>10.87</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>127207.00</td>
<td>10931</td>
<td>11.63</td>
</tr>
<tr>
<td>Spice Sampler</td>
<td>6060.00</td>
<td>505</td>
<td>12.00</td>
</tr>
<tr>
<td>Aroma Sounds CD</td>
<td>7125.00</td>
<td>550</td>
<td>12.95</td>
</tr>
<tr>
<td>French Press, 2-Cup</td>
<td>3329.80</td>
<td>224</td>
<td>14.86</td>
</tr>
<tr>
<td>Spice Jar</td>
<td>4229.00</td>
<td>235</td>
<td>17.99</td>
</tr>
<tr>
<td>French Press, 4-Cup</td>
<td>3323.65</td>
<td>167</td>
<td>19.90</td>
</tr>
<tr>
<td>Tea Sampler</td>
<td>13695.00</td>
<td>550</td>
<td>24.90</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(2 of 2)
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;
```
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.

<table>
<thead>
<tr>
<th>WHERE Clause</th>
<th>HAVING Clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Works on rows of data prior to grouping.</td>
<td>Works on the result set after grouping.</td>
</tr>
<tr>
<td>Conditions cannot be expressed with set functions (for example, SUM or AVG), but column aliases for non-aggregate expressions can be used.</td>
<td>Conditions can be expressed with any set function or column alias.</td>
</tr>
</tbody>
</table>

**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

```sql
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**

<table>
<thead>
<tr>
<th>Supplier</th>
<th>Avg_Deal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Espresso Express</td>
<td>500.00</td>
</tr>
<tr>
<td>Western Emporium</td>
<td>66.66</td>
</tr>
<tr>
<td>Aroma West Mfg.</td>
<td>50.00</td>
</tr>
<tr>
<td>CB Imports</td>
<td>47.50</td>
</tr>
<tr>
<td>Leaves of London</td>
<td>40.00</td>
</tr>
<tr>
<td>Tea Makers, Inc.</td>
<td>20.00</td>
</tr>
<tr>
<td>Colo Coffee</td>
<td>0.00</td>
</tr>
<tr>
<td>Aroma East Mfg.</td>
<td>0.00</td>
</tr>
<tr>
<td>Crashing By Design</td>
<td>0.00</td>
</tr>
</tbody>
</table>

**Example Query 2**

```sql
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;
```
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 2-45, 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 or SQL OLAP functions are computed. Consequently, you cannot suppress rows by referencing a column that contains one of these functions. For examples of queries that include OLAP functions and display functions, refer to Chapter 3, “Data Analysis.”

---

**Summary**

**The SELECT Statement**

```sql
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 OLAP functions and RISQL extensions to answer business questions that require sequential processing.
In This Chapter ............................................ 3-3
Analytic Functions ....................................... 3-4
Cumulative Totals ....................................... 3-5
Resetting Cumulative Totals. .......................... 3-8
Comparing Running Totals with OLAP .................. 3-11
Moving Averages ......................................... 3-14
Moving Sums ............................................. 3-17
Ranking Data ............................................ 3-20
Using the WHEN Clause .................................. 3-23
Ranking Values in Groups: NTILE ....................... 3-25
Using the NTILE Function with a CASE Expression .................................................. 3-28
Using the TERTILE Function ............................ 3-32
Calculating Ratios ....................................... 3-35
Using the DATEADD Function .......................... 3-38
Using the DATEDIFF Function .......................... 3-42
Using the EXTRACT Function .......................... 3-44
Summary .................................................. 3-47
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?

IBM Red Brick Warehouse provides two efficient ways to answer these kinds of questions:

- Standard SQL OLAP functions
- RISQL display functions

This chapter contains a series of examples, with the equivalent OLAP and RISQL syntax presented in each case. Before running these queries, read the full descriptions of the OLAP and RISQL functions in the SQL Reference Guide.

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

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. Refer to the IBM Red Brick Vista User’s Guide for instructions on creating and using aggregate tables.
The following table lists the SQL OLAP functions and corresponding RISQL display functions:

<table>
<thead>
<tr>
<th>SQL OLAP Function</th>
<th>Equivalent RISQL Display Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANK</td>
<td>RANK</td>
</tr>
<tr>
<td>DENSE_RANK, DENSERANK</td>
<td></td>
</tr>
<tr>
<td>NTILE</td>
<td>NTILE, TERTILE</td>
</tr>
<tr>
<td>RATIO_TO_REPORT, RATIO_TO_REPORT</td>
<td>RATIO_TO_REPORT</td>
</tr>
<tr>
<td>ROW_NUMBER, ROWNUMBER</td>
<td>CUME(1)</td>
</tr>
<tr>
<td>AVG</td>
<td>MOVINGAVG</td>
</tr>
<tr>
<td>COUNT</td>
<td></td>
</tr>
<tr>
<td>MIN</td>
<td></td>
</tr>
<tr>
<td>MAX</td>
<td></td>
</tr>
<tr>
<td>SUM</td>
<td>CUME, MOVINGSUM</td>
</tr>
</tbody>
</table>

For detailed syntax descriptions and usage notes, refer to the SQL Reference Guide.
Cumulative Totals

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?

OLAP Query

```
select date, sum(dollars) as total_dollars,
       sum(total_dollars) over(order by date rows unbounded preceding)
       as run_dollars,
       sum(quantity) as total_qty,
       sum(total_qty) over(order by date rows unbounded preceding)
       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;
```

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

...
OLAP SUM Function

The presence of the OVER() clause differentiates a simple SET function (SUM, MIN, MAX, COUNT, AVG) from an OLAP aggregation function.

The OLAP SUM function produces running totals when the window frame specifies:

```
rows unbounded preceding
```

For detailed information about window frames, refer to the SQL Reference Guide.

The OLAP ORDER BY clause is critical. This specification ensures that the input rows for the OLAP SUM function are sorted correctly (by Date, in this case). The final ORDER BY clause is separate and distinct from the ORDER BY clause for the OLAP function.

RISQL CUME Function

The CUME function also calculates and displays running totals. Place the CUME function in the select list for each numeric column to be summed:

```
CUME(expression)
```

where expression is a numeric column name or expression. The ORDER BY clause is required to ensure that the CUME function operates on an ordered set of rows. To obtain the correct running totals with CUME, always include an ORDER BY clause.

The CUME function does not have to contain a column reference. For example:

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

<table>
<thead>
<tr>
<th>ROW_NUM</th>
<th>ORDER_NO</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3600</td>
<td>1200.46</td>
</tr>
<tr>
<td>2</td>
<td>3601</td>
<td>1535.94</td>
</tr>
<tr>
<td>3</td>
<td>3602</td>
<td>780.00</td>
</tr>
<tr>
<td>4</td>
<td>3603</td>
<td>956.45</td>
</tr>
</tbody>
</table>

... 

The OLAP ROW_NUMBER function can be used to the same effect:

```
select row_number() over() as row_num, order_no, price from orders;
```

Data Analysis 3-7
Resetting Cumulative Totals

Question

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

OLAP Query

```sql
select week, date, sum(dollars) as total_dollars,
    sum(total_dollars) over(partition by week order by date
    rows unbounded preceding) as run_dollars,
    sum(quantity) as total_qty,
    sum(total_qty) over(partition by week order by date
    rows unbounded preceding) 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;
```

RISQL Query

```sql
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

<table>
<thead>
<tr>
<th>Week</th>
<th>Date</th>
<th>Total_Dollars</th>
<th>Run_Dollars</th>
<th>Total_Qty</th>
<th>Run_Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2000-01-02</td>
<td>855.50</td>
<td>855.50</td>
<td>118</td>
<td>118</td>
</tr>
<tr>
<td>2</td>
<td>2000-01-03</td>
<td>536.50</td>
<td>1392.00</td>
<td>74</td>
<td>192</td>
</tr>
<tr>
<td>2</td>
<td>2000-01-04</td>
<td>181.25</td>
<td>1573.25</td>
<td>25</td>
<td>217</td>
</tr>
<tr>
<td>2</td>
<td>2000-01-05</td>
<td>362.50</td>
<td>1935.75</td>
<td>50</td>
<td>267</td>
</tr>
<tr>
<td>2</td>
<td>2000-01-06</td>
<td>667.00</td>
<td>2602.75</td>
<td>92</td>
<td>359</td>
</tr>
<tr>
<td>2</td>
<td>2000-01-07</td>
<td>659.75</td>
<td>3262.50</td>
<td>91</td>
<td>450</td>
</tr>
<tr>
<td>2</td>
<td>2000-01-08</td>
<td>309.50</td>
<td>3572.00</td>
<td>54</td>
<td>504</td>
</tr>
<tr>
<td>3</td>
<td>2000-01-09</td>
<td>195.75</td>
<td>195.75</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>3</td>
<td>2000-01-10</td>
<td>420.50</td>
<td>616.25</td>
<td>58</td>
<td>85</td>
</tr>
<tr>
<td>3</td>
<td>2000-01-11</td>
<td>547.50</td>
<td>1163.75</td>
<td>78</td>
<td>163</td>
</tr>
<tr>
<td>3</td>
<td>2000-01-12</td>
<td>536.50</td>
<td>1700.25</td>
<td>74</td>
<td>237</td>
</tr>
<tr>
<td>3</td>
<td>2000-01-13</td>
<td>638.00</td>
<td>2338.25</td>
<td>88</td>
<td>325</td>
</tr>
<tr>
<td>3</td>
<td>2000-01-14</td>
<td>1057.50</td>
<td>3395.75</td>
<td>150</td>
<td>475</td>
</tr>
<tr>
<td>3</td>
<td>2000-01-15</td>
<td>884.50</td>
<td>4280.25</td>
<td>122</td>
<td>597</td>
</tr>
<tr>
<td>4</td>
<td>2000-01-16</td>
<td>761.25</td>
<td>761.25</td>
<td>105</td>
<td>105</td>
</tr>
<tr>
<td>4</td>
<td>2000-01-17</td>
<td>455.50</td>
<td>1216.75</td>
<td>66</td>
<td>171</td>
</tr>
<tr>
<td>4</td>
<td>2000-01-18</td>
<td>768.50</td>
<td>1985.25</td>
<td>106</td>
<td>277</td>
</tr>
<tr>
<td>4</td>
<td>2000-01-19</td>
<td>746.75</td>
<td>2732.00</td>
<td>103</td>
<td>380</td>
</tr>
<tr>
<td>4</td>
<td>2000-01-20</td>
<td>261.00</td>
<td>2993.00</td>
<td>36</td>
<td>416</td>
</tr>
<tr>
<td>4</td>
<td>2000-01-21</td>
<td>630.75</td>
<td>3623.75</td>
<td>87</td>
<td>503</td>
</tr>
<tr>
<td>4</td>
<td>2000-01-22</td>
<td>813.75</td>
<td>4437.50</td>
<td>115</td>
<td>618</td>
</tr>
</tbody>
</table>

...
OLAP Window Partitions

You can insert blank lines between each partition in a result set by using the RISQL Reporter command SET COLUMN column_name SKIP LINE, where column_name is the PARTITION BY (or RESET BY) column:

```sql
RISQL> set column week skip line;
```

The OLAP PARTITION BY clause inside the OVER() clause provides a means of resetting calculations when values in partitioned columns change. You can partition OLAP calculations by one or more columns.

RISQL Alternative: CUME with RESET BY

You can reset the calculation of cumulative subtotals with CUME by including a RESET BY clause in the final ORDER BY clause. The RESET BY subclause must occur within the ORDER BY clause:

```sql
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];
```

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.

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.
Comparing Running Totals with OLAP

Question

How do the Western and Southern regions compare in terms of running daily sales totals for March 2000?

OLAP Query

```sql
select t1.date, sales_cume_west, sales_cume_south
from
(select date, sum(dollars) as total_sales,
    sum(total_sales) over(order by date rows unbounded preceding)
    as sales_cume_west
from market natural join store
    natural join sales
    natural join period
where year = 2000
    and month = 'MAR'
    and region = 'West'
group by date) as t1
join
(select date, sum(dollars) as total_sales,
    sum(total_sales) over(order by date rows unbounded preceding)
    as sales_cume_south
from market natural join store
    natural join sales
    natural join period
where year = 2000
    and month = 'MAR'
    and region = 'South'
group by date) as t2
using(date)
order by date;
```
### Result

<table>
<thead>
<tr>
<th>Date</th>
<th>Sales_Cume_West</th>
<th>Sales_Cume_South</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-03-01</td>
<td>2529.25</td>
<td>2056.75</td>
</tr>
<tr>
<td>2000-03-02</td>
<td>6809.00</td>
<td>4146.75</td>
</tr>
<tr>
<td>2000-03-03</td>
<td>9068.75</td>
<td>6366.55</td>
</tr>
<tr>
<td>2000-03-04</td>
<td>12679.35</td>
<td>8831.30</td>
</tr>
<tr>
<td>2000-03-05</td>
<td>16228.60</td>
<td>11100.55</td>
</tr>
<tr>
<td>2000-03-06</td>
<td>19653.30</td>
<td>12665.65</td>
</tr>
<tr>
<td>2000-03-07</td>
<td>23515.55</td>
<td>14882.90</td>
</tr>
<tr>
<td>2000-03-08</td>
<td>27207.80</td>
<td>17494.15</td>
</tr>
<tr>
<td>2000-03-09</td>
<td>31725.95</td>
<td>19745.40</td>
</tr>
<tr>
<td>2000-03-10</td>
<td>34836.20</td>
<td>21323.40</td>
</tr>
<tr>
<td>2000-03-11</td>
<td>38257.00</td>
<td>23256.15</td>
</tr>
<tr>
<td>2000-03-12</td>
<td>42498.20</td>
<td>25091.35</td>
</tr>
<tr>
<td>2000-03-13</td>
<td>46691.95</td>
<td>27362.85</td>
</tr>
<tr>
<td>2000-03-14</td>
<td>49797.60</td>
<td>29729.50</td>
</tr>
<tr>
<td>2000-03-15</td>
<td>53692.85</td>
<td>31614.10</td>
</tr>
<tr>
<td>2000-03-16</td>
<td>57925.15</td>
<td>33974.60</td>
</tr>
<tr>
<td>2000-03-17</td>
<td>60357.15</td>
<td>37369.75</td>
</tr>
<tr>
<td>2000-03-18</td>
<td>62478.05</td>
<td>39420.50</td>
</tr>
<tr>
<td>2000-03-19</td>
<td>65415.55</td>
<td>41737.00</td>
</tr>
<tr>
<td>2000-03-20</td>
<td>68076.55</td>
<td>43879.95</td>
</tr>
<tr>
<td>2000-03-21</td>
<td>71173.10</td>
<td>45899.70</td>
</tr>
<tr>
<td>2000-03-22</td>
<td>74768.70</td>
<td>48129.80</td>
</tr>
<tr>
<td>2000-03-23</td>
<td>78084.20</td>
<td>50528.80</td>
</tr>
</tbody>
</table>
One advantage of the OLAP approach to running sequential calculations is the ability to place them inside subqueries. A RISQL CUME function cannot be reliably placed inside a subquery because its results depend on the ordering defined by the ORDER BY clause. (A subquery cannot contain an ORDER BY clause.)

OLAP functions do not have this limitation because OLAP ordering is part of the function itself, and each OLAP function has its own ORDER BY clause, independent of the query’s final ORDER BY clause.

For information about the structure of this query, which contains two subqueries joined in the FROM clause of the outer query, see Chapter 4, “Comparison Queries.”

<table>
<thead>
<tr>
<th>Date</th>
<th>Sales_Cume_West</th>
<th>Sales_Cume_South</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-03-24</td>
<td>82288.80</td>
<td>52381.45</td>
</tr>
<tr>
<td>2000-03-25</td>
<td>86317.65</td>
<td>55180.50</td>
</tr>
<tr>
<td>2000-03-26</td>
<td>90676.55</td>
<td>57325.75</td>
</tr>
<tr>
<td>2000-03-27</td>
<td>94865.55</td>
<td>59063.50</td>
</tr>
<tr>
<td>2000-03-28</td>
<td>97510.50</td>
<td>60975.35</td>
</tr>
<tr>
<td>2000-03-29</td>
<td>100513.85</td>
<td>62891.35</td>
</tr>
<tr>
<td>2000-03-30</td>
<td>104267.40</td>
<td>65378.75</td>
</tr>
<tr>
<td>2000-03-31</td>
<td>107222.15</td>
<td>68100.75</td>
</tr>
</tbody>
</table>
Moving Averages

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.

Question

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

OLAP Query

```sql
select city, week, sum(dollars) as sales,
       string(avg(sales) over(partition by city order by city, week rows 2 preceding),7,2) as mov_avg,
       sum(sales) over(partition by city order by week rows unbounded preceding) 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;
```

RISQL Query

```sql
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;
```

The following result set is the RISQL result set, which contains NULLs for the MOVINGAVG function in the first two rows in each partition. In the OLAP case, these NULLs are replaced with averages calculated over the first row and the first and second rows, respectively.
<table>
<thead>
<tr>
<th>City</th>
<th>Week</th>
<th>Sales</th>
<th>Mov_avg</th>
<th>Run_sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miami</td>
<td>27</td>
<td>1838.55</td>
<td>NULL</td>
<td>1838.55</td>
</tr>
<tr>
<td>Miami</td>
<td>28</td>
<td>4482.15</td>
<td>NULL</td>
<td>6320.70</td>
</tr>
<tr>
<td>Miami</td>
<td>29</td>
<td>4616.70</td>
<td>3645.80</td>
<td>10937.40</td>
</tr>
<tr>
<td>Miami</td>
<td>30</td>
<td>4570.35</td>
<td>4556.40</td>
<td>15507.75</td>
</tr>
<tr>
<td>Miami</td>
<td>31</td>
<td>4681.95</td>
<td>4623.00</td>
<td>20189.70</td>
</tr>
<tr>
<td>Miami</td>
<td>32</td>
<td>3004.50</td>
<td>4085.60</td>
<td>23194.20</td>
</tr>
<tr>
<td>Miami</td>
<td>33</td>
<td>3915.90</td>
<td>3867.45</td>
<td>27110.10</td>
</tr>
<tr>
<td>Miami</td>
<td>34</td>
<td>4119.35</td>
<td>3679.91</td>
<td>31229.45</td>
</tr>
<tr>
<td>Miami</td>
<td>35</td>
<td>2558.90</td>
<td>3531.38</td>
<td>33788.35</td>
</tr>
<tr>
<td>Miami</td>
<td>36</td>
<td>4556.25</td>
<td>3744.83</td>
<td>38344.60</td>
</tr>
<tr>
<td>Miami</td>
<td>37</td>
<td>5648.50</td>
<td>4254.55</td>
<td>43993.10</td>
</tr>
<tr>
<td>Miami</td>
<td>38</td>
<td>5500.25</td>
<td>5235.00</td>
<td>49493.35</td>
</tr>
<tr>
<td>Miami</td>
<td>39</td>
<td>4891.40</td>
<td>5346.71</td>
<td>54384.75</td>
</tr>
<tr>
<td>Miami</td>
<td>40</td>
<td>3693.80</td>
<td>4695.15</td>
<td>58078.55</td>
</tr>
<tr>
<td>San Jose</td>
<td>27</td>
<td>3177.55</td>
<td>NULL</td>
<td>3177.55</td>
</tr>
<tr>
<td>San Jose</td>
<td>28</td>
<td>5825.80</td>
<td>NULL</td>
<td>9003.35</td>
</tr>
<tr>
<td>San Jose</td>
<td>29</td>
<td>8474.80</td>
<td>5826.05</td>
<td>17478.15</td>
</tr>
<tr>
<td>San Jose</td>
<td>30</td>
<td>7976.60</td>
<td>7425.73</td>
<td>25454.75</td>
</tr>
<tr>
<td>San Jose</td>
<td>31</td>
<td>7328.65</td>
<td>7926.68</td>
<td>32783.40</td>
</tr>
<tr>
<td>San Jose</td>
<td>32</td>
<td>6809.75</td>
<td>7371.66</td>
<td>39593.15</td>
</tr>
<tr>
<td>San Jose</td>
<td>33</td>
<td>7116.35</td>
<td>7084.91</td>
<td>46709.50</td>
</tr>
</tbody>
</table>

...
The STRING scalar function is used to truncate the long-numeric values returned for the Mov_Avg column. For details about this function, refer to the SQL Reference Guide. Regardless of the scalar function you use, you might see rounding differences in the results of RISQL and OLAP functions.

**MOVINGAVG Function**

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

\[
\text{MOVINGAVG}(n\_\text{expression}, \; n)
\]

In this example, 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 unexpected results. Therefore, the ORDER BY clause includes both the City column and the Week column.

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

**OLAP AVG Function**

The equivalent of a RISQL MOVINGAVG function is an OLAP AVG function with the following window frame:

\[
\text{rows} \; n \text{ preceding}
\]

where \( n \) is a number that represents the required smoothing factor. The FOLLOWING keyword can also be used, but the PRECEDING syntax produces equivalent results to the RISQL MOVINGAVG function.

The OLAP ORDER BY clause ensures that the AVG function is applied to the correct moving sequence of rows. The PARTITION BY clause is equivalent to the RISQL RESET BY clause.
Moving Sums

Question

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

OLAP Query

\[
\begin{align*}
\text{select} & \quad \text{date, sum(quantity) as day_qty,} \\
& \quad \text{string(sum(day_qty) over(order by date rows 6 preceding),7,2) as mov_sum} \\
\text{from} & \quad \text{store natural join sales natural join period} \\
& \quad \text{natural join product} \\
\text{where} & \quad \text{year = 2000 and month = 'MAR' and prod_name = 'Demitasse Ms'} \\
& \quad \text{group by date} \\
\text{order by date;}
\end{align*}
\]

RISQL Query

\[
\begin{align*}
\text{select} & \quad \text{date, sum(quantity) as day_qty,} \\
& \quad \text{string(movingsum(sum(quantity), 7),7,2) as mov_sum} \\
\text{from} & \quad \text{store natural join sales natural join period} \\
& \quad \text{natural join product} \\
\text{where} & \quad \text{year = 2000 and month = 'MAR' and prod_name = 'Demitasse Ms'} \\
& \quad \text{group by date} \\
\text{order by date;}
\end{align*}
\]

The following result set is the OLAP result set, which contains moving sums for all of the rows. In the RISQL case, the first six rows contain NULLs. The RISQL logic requires seven rows before the first moving sum can be calculated.
<table>
<thead>
<tr>
<th>Date</th>
<th>Day_Qty</th>
<th>Mov_Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-03-01</td>
<td>65</td>
<td>65.00</td>
</tr>
<tr>
<td>2000-03-02</td>
<td>19</td>
<td>84.00</td>
</tr>
<tr>
<td>2000-03-03</td>
<td>92</td>
<td>176.00</td>
</tr>
<tr>
<td>2000-03-04</td>
<td>91</td>
<td>267.00</td>
</tr>
<tr>
<td>2000-03-05</td>
<td>106</td>
<td>373.00</td>
</tr>
<tr>
<td>2000-03-06</td>
<td>92</td>
<td>465.00</td>
</tr>
<tr>
<td>2000-03-07</td>
<td>102</td>
<td>567.00</td>
</tr>
<tr>
<td>2000-03-08</td>
<td>21</td>
<td>523.00</td>
</tr>
<tr>
<td>2000-03-09</td>
<td>74</td>
<td>578.00</td>
</tr>
<tr>
<td>2000-03-10</td>
<td>81</td>
<td>567.00</td>
</tr>
<tr>
<td>2000-03-11</td>
<td>77</td>
<td>553.00</td>
</tr>
<tr>
<td>2000-03-12</td>
<td>127</td>
<td>574.00</td>
</tr>
<tr>
<td>2000-03-13</td>
<td>169</td>
<td>651.00</td>
</tr>
<tr>
<td>2000-03-14</td>
<td>31</td>
<td>580.00</td>
</tr>
<tr>
<td>2000-03-15</td>
<td>56</td>
<td>615.00</td>
</tr>
<tr>
<td>2000-03-16</td>
<td>40</td>
<td>581.00</td>
</tr>
<tr>
<td>2000-03-17</td>
<td>84</td>
<td>584.00</td>
</tr>
<tr>
<td>2000-03-18</td>
<td>34</td>
<td>541.00</td>
</tr>
<tr>
<td>2000-03-19</td>
<td>128</td>
<td>542.00</td>
</tr>
<tr>
<td>2000-03-20</td>
<td>97</td>
<td>470.00</td>
</tr>
<tr>
<td>2000-03-21</td>
<td>50</td>
<td>489.00</td>
</tr>
<tr>
<td>2000-03-22</td>
<td>147</td>
<td>580.00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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.

The STRING scalar function is used to truncate the long-numeric values returned for the `Mov_Sum` column. For details about this function, refer to the *SQL Reference Guide*. Regardless of the scalar function you use, you might see rounding differences in the results of RISQL and OLAP functions.

**RISQL MOVINGSUM Function**

The MOVINGSUM 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)
```

If the MOVINGSUM function is applied to results that are not in chronological order, the function will return unexpected 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.

**OLAP SUM Function**

The OLAP SUM function produces moving sums when the window frame specifies:

```
rows n preceding
```

The FOLLOWING keyword can also be used, but the PRECEDING syntax produces equivalent results to the RISQL MOVINGSUM function.

The OLAP ORDER BY clause is critical. This specification ensures that the input rows for the OLAP SUM function are sorted correctly (by ascending `Date`, in this case).
Ranking Data

Question

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

OLAP Query

```sql
select store_name, district, sum(dollars) as total_sales,
    rank() over(order by total_sales desc) 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;
```

RISQL Query

```sql
select store_name, district, sum(dollars) as total_sales,
    rank(total_sales) 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;
```
The daily totals from the Sales table that meet the search conditions in the WHERE clause are summed, then ranked.

**RISQL RANK Function**

To rank a set of values, specify:

\[
\text{RANK(expression)}
\]

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.

You can rank any set of values with the RISQL 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 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:

\[
\text{RANK(-expression)}
\]

For example:

\[
\text{rank(-dollars) as sales_rank}
\]
OLAP RANK Function

For nonnumeric data types, the ranking is based on the collation sequence defined by the database locale.

OLAP RANK Function

IBM Red Brick Warehouse supports a set of OLAP ranking functions, including RANK() and DENSE_RANK().

These functions do not require any arguments; instead, the OLAP ORDER BY clause defines the column or expression on which the rankings are based. Note that the default sort order for OLAP functions is ascending (ASC), which is often not the required ranking for business queries. You must specify the DESC keyword in the OLAP ORDER BY clause to assign ranks from high to low (where 1 is the highest value).

The DENSE_RANK function differs from RANK in one respect: If two or more rows tie, there is no gap in the sequence of ranked values. For example, if two rows are ranked 8, the next ranking is still 9, as shown in the following example:

RISQL> select prod_name, dollars,
>       dense_rank() over(order by dollars desc) as dense_rank,
>       rank() over(order by dollars desc) as tie_rank
> from product natural join sales natural join period
> where date = '01-04-2000'
> ;

<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>DOLLARS</th>
<th>DENSE_RANK</th>
<th>TIE_RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Espresso Machine Italiano</td>
<td>499.75</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>392.00</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Veracruzano</td>
<td>360.00</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>328.00</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>NA Lite</td>
<td>306.00</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Colombiano</td>
<td>283.50</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>207.00</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Colombiano</td>
<td>202.50</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Colombiano</td>
<td>202.50</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Expresso XO</td>
<td>201.50</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>Xalapa Lapa</td>
<td>195.50</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>168.00</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>Xalapa Lapa</td>
<td>162.00</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>Veracruzano</td>
<td>157.50</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>Veracruzano</td>
<td>152.00</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>Demitasse Ms</td>
<td>143.50</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>136.00</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>Colombiano</td>
<td>128.25</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>Aroma Roma</td>
<td>123.25</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>120.00</td>
<td>19</td>
<td>20</td>
</tr>
</tbody>
</table>
Using the WHEN Clause

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?

OLAP Query

```sql
select date, day, dollars as day_sales,
    rank() over(order by day_sales desc) as sales_rank,
    quantity as day_qty,
    rank() over(order by day_qty desc) 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;
```

RISQL Query

```sql
select date, day, dollars as day_sales,
    rank(day_sales) as sales_rank,
    quantity as day_qty,
    rank(day_qty) 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;
```
When values to be ranked are equal, they are assigned the same value. For example, two rows receive a sales rank of 3 in this case.

**WHEN Clause**

To restrict the rows returned in a result table after ranked values are calculated, include a WHEN clause in the query:

```sql
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];
```
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. This functionality applies to both RISQL ranking functions and OLAP ranking functions.

### Ranking Values in Groups: NTILE

#### Question

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

#### OLAP Query

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

#### RISQL 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);
```
The 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.

<table>
<thead>
<tr>
<th>Prod_name</th>
<th>Total_sales</th>
<th>Sales_rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demitasse Ms</td>
<td>304727.00</td>
<td>1</td>
</tr>
<tr>
<td>Xalapa Lapa</td>
<td>263353.00</td>
<td>1</td>
</tr>
<tr>
<td>NA Lite</td>
<td>262162.00</td>
<td>1</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>251713.00</td>
<td>1</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>251086.50</td>
<td>1</td>
</tr>
<tr>
<td>Expresso XO</td>
<td>229201.25</td>
<td>1</td>
</tr>
<tr>
<td>Veracruzano</td>
<td>227769.50</td>
<td>1</td>
</tr>
<tr>
<td>La Antigua</td>
<td>223528.25</td>
<td>1</td>
</tr>
<tr>
<td>Aroma Roma</td>
<td>218574.75</td>
<td>1</td>
</tr>
<tr>
<td>Colombiano</td>
<td>218009.50</td>
<td>1</td>
</tr>
<tr>
<td>Aroma Sounds CD</td>
<td>5937.00</td>
<td>4</td>
</tr>
<tr>
<td>Aroma Sounds Cassette</td>
<td>5323.00</td>
<td>4</td>
</tr>
<tr>
<td>French Press, 4-Cup</td>
<td>4570.50</td>
<td>4</td>
</tr>
<tr>
<td>Spice Jar</td>
<td>4073.00</td>
<td>4</td>
</tr>
<tr>
<td>French Press, 2-Cup</td>
<td>3042.75</td>
<td>4</td>
</tr>
<tr>
<td>Travel Mug</td>
<td>1581.75</td>
<td>4</td>
</tr>
<tr>
<td>Easter Sampler Basket</td>
<td>1500.00</td>
<td>4</td>
</tr>
<tr>
<td>Coffee Mug</td>
<td>1258.00</td>
<td>4</td>
</tr>
<tr>
<td>Christmas Sampler</td>
<td>1230.00</td>
<td>4</td>
</tr>
</tbody>
</table>
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.

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.

**RISQL NTILE Function**

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*.

**OLAP NTILE Function**

Like the OLAP RANK function, the NTILE function depends on an expression in the OLAP ORDER BY clause. The function itself requires only one argument, the tile value:

```
NTILE(n) OVER(ORDER BY expression DESC)
```

As in the previous example with the OLAP RANK function, the DESC keyword must be specified if you want the rankings to be assigned based on high-to-low `expression` values.

**Important:** By using an NTILE function inside a CASE expression, you can redistribute ranked values into unequal groups and replace the default numeric NTILE 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?

OLAP Query

```sql
select prod_name, sum(quantity) as quantity,
       sum(dollars) as sales,
       case ntile(5) over(order by sales desc)
           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;
```

RISQL Query

```sql
select prod_name, sum(quantity) as quantity,
       sum(dollars) as sales,
       case ntile(sales, 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;
```
<table>
<thead>
<tr>
<th>Prod_Name</th>
<th>Qty</th>
<th>Sales</th>
<th>Grp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expresso XO</td>
<td>368</td>
<td>2887.00</td>
<td>TOP_20</td>
</tr>
<tr>
<td>Aroma Roma</td>
<td>246</td>
<td>1783.50</td>
<td>TOP_20</td>
</tr>
<tr>
<td>Colombiano</td>
<td>257</td>
<td>1757.75</td>
<td>TOP_20</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>143</td>
<td>1655.00</td>
<td>TOP_20</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>198</td>
<td>1621.00</td>
<td>TOP_20</td>
</tr>
<tr>
<td>La Antigua</td>
<td>213</td>
<td>1589.25</td>
<td>TOP_20</td>
</tr>
<tr>
<td>Demitasse Ms</td>
<td>151</td>
<td>1503.75</td>
<td>MID_60</td>
</tr>
<tr>
<td>Xalapa Lapa</td>
<td>163</td>
<td>1395.50</td>
<td>MID_60</td>
</tr>
<tr>
<td>Ruby's Allspice</td>
<td>183</td>
<td>1018.50</td>
<td>MID_60</td>
</tr>
<tr>
<td>Veracruzano</td>
<td>120</td>
<td>925.50</td>
<td>MID_60</td>
</tr>
<tr>
<td>NA Lite</td>
<td>100</td>
<td>900.00</td>
<td>MID_60</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>106</td>
<td>869.50</td>
<td>MID_60</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>104</td>
<td>636.50</td>
<td>MID_60</td>
</tr>
<tr>
<td>English Breakfast</td>
<td>137</td>
<td>561.50</td>
<td>MID_60</td>
</tr>
<tr>
<td>Aroma t-shirt</td>
<td>44</td>
<td>481.80</td>
<td>MID_60</td>
</tr>
<tr>
<td>Coffee Sampler</td>
<td>16</td>
<td>480.00</td>
<td>MID_60</td>
</tr>
<tr>
<td>Assam Grade A</td>
<td>114</td>
<td>380.00</td>
<td>MID_60</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>69</td>
<td>378.25</td>
<td>MID_60</td>
</tr>
<tr>
<td>Irish Breakfast</td>
<td>81</td>
<td>345.75</td>
<td>MID_60</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>82</td>
<td>322.00</td>
<td>MID_60</td>
</tr>
<tr>
<td>Gold Tips</td>
<td>91</td>
<td>320.75</td>
<td>MID_60</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>80</td>
<td>305.00</td>
<td>MID_60</td>
</tr>
</tbody>
</table>
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. This functionality applies to both the RISQL NTILE function and the OLAP NTILE function.

This 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 \textit{TOP\_20}, the middle 60 percent as \textit{MID\_60}, and the bottom 20 percent as \textit{LOW\_20}. When the expression evaluates to 2, 3, or 4, those values are replaced with \textit{MID\_60}, and when it evaluates to 5, it is replaced with \textit{LOW\_20}.

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

\texttt{when grp = 'MID\_60'}
**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**: An optional default value used 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 4-7.
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?

RISQL Query

```sql
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;
```
You can rank each value in a group of numeric values as High, Middle, or Low with the RISQL 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.

This 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.

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.

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

\[ \text{TERTILE(expression)} \]

expression: A column name or a numeric expression.
If expression is NULL, TERTILE returns NULL. Even though the column referenced by the TERTILE function must be numeric, the result of this function is always a character column.

**OLAP NTILE for TERTILE**

There is no equivalent OLAP function for the TERTILE function. However, the TERTILE function is really a special form of NTILE, so the NTILE(3) function can be used with a CASE expression to produce the same result:

```sql
select city, sum(quantity) as qty_1999,
    case ntile(3) over(order by qty_1999 desc)
        when 1 then 'H'
        when 2 then 'M'
        when 3 then 'L'
        end as q_rk
from ...
```
Calculating Ratios

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?

OLAP Query

```sql
select city, month, sum(dollars) as total_sales,
       dec(ratiotobase(total_sales) over()*100,5,2) 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;
```

RISQL Query

```sql
select city, month, sum(dollars) as total_sales,
       dec(ratiotobase(total_sales)*100,5,2) 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;
```
The results of OLAP and RISQL queries that calculate ratios sometimes differ slightly. The RISQL RATIOTOREPORT function always returns a floating-point inexact datatype. The OLAP RATIOTOREPORT calculation depends on the function argument (if the function argument is exact, the result will be exact with greater precision, but if the function argument is inexact, the result will be inexact).

### Ratios as Percentages: RATIOTOREPORT*100

The RATIOSEXPRESSION 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.

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

```
*100
```

This 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 values in the Pct_of_Sales column add up to exactly 100.

<table>
<thead>
<tr>
<th>City</th>
<th>Month</th>
<th>Total_Sales</th>
<th>Pct_of_Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Jose</td>
<td>JUL</td>
<td>2499.50</td>
<td>26.99</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>JUL</td>
<td>1627.00</td>
<td>17.57</td>
</tr>
<tr>
<td>San Jose</td>
<td>AUG</td>
<td>1004.00</td>
<td>10.84</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>AUG</td>
<td>995.00</td>
<td>10.74</td>
</tr>
<tr>
<td>San Jose</td>
<td>SEP</td>
<td>1802.00</td>
<td>19.46</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>SEP</td>
<td>1334.00</td>
<td>14.40</td>
</tr>
</tbody>
</table>
**RISQL RATIOREPORT Function**

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

```
RATIOREPORT(expression)
```

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

The RATIOREPORT 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 3-10.

**OLAP RATIOREPORT or RATIO_TO_REPORT Function**

The OLAP RATIOREPORT and RATIO_TO_REPORT functions are synonymous. Like the RISQL function, they must have a numeric expression as their argument:

```
RATIO_TO_REPORT(expression) OVER()
```

The OLAP ORDER BY and PARTITION BY clauses are optional, but the empty `OVER` clause must be specified.
Using the DATEADD Function

Question

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

Example Query

```sql
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

<table>
<thead>
<tr>
<th>Due_Date</th>
<th>Cur_Date</th>
<th>Past_Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>1999-10-03</td>
<td>2000-01-01</td>
<td>2000-03-31</td>
</tr>
<tr>
<td>1999-10-04</td>
<td>2000-01-02</td>
<td>2000-04-01</td>
</tr>
<tr>
<td>1999-10-05</td>
<td>2000-01-03</td>
<td>2000-04-02</td>
</tr>
<tr>
<td>1999-10-06</td>
<td>2000-01-04</td>
<td>2000-04-03</td>
</tr>
<tr>
<td>1999-10-08</td>
<td>2000-01-06</td>
<td>2000-04-05</td>
</tr>
<tr>
<td>1999-10-09</td>
<td>2000-01-07</td>
<td>2000-04-06</td>
</tr>
<tr>
<td>1999-10-10</td>
<td>2000-01-08</td>
<td>2000-04-07</td>
</tr>
<tr>
<td>1999-10-11</td>
<td>2000-01-09</td>
<td>2000-04-08</td>
</tr>
<tr>
<td>1999-10-12</td>
<td>2000-01-10</td>
<td>2000-04-09</td>
</tr>
<tr>
<td>Due_Date</td>
<td>Cur_Date</td>
<td>Past_Due</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>-----------</td>
</tr>
<tr>
<td>1999-10-14</td>
<td>2000-01-12</td>
<td>2000-04-11</td>
</tr>
<tr>
<td>1999-10-17</td>
<td>2000-01-15</td>
<td>2000-04-14</td>
</tr>
<tr>
<td>1999-10-19</td>
<td>2000-01-17</td>
<td>2000-04-16</td>
</tr>
</tbody>
</table>

(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)

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATEADD(day, 90, '07-01-99')</td>
<td>1999-09-29</td>
</tr>
<tr>
<td>DATEADD(month, 3, '07-01-99')</td>
<td>1999-10-01</td>
</tr>
<tr>
<td>DATEADD(year, 1, '07-01-99')</td>
<td>2000-07-01</td>
</tr>
</tbody>
</table>

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:

```sql
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';
```

<table>
<thead>
<tr>
<th>Prior</th>
<th>Cur</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>October</td>
<td>January</td>
<td>March</td>
</tr>
<tr>
<td>October</td>
<td>January</td>
<td>April</td>
</tr>
<tr>
<td>October</td>
<td>January</td>
<td>April</td>
</tr>
<tr>
<td>October</td>
<td>January</td>
<td>April</td>
</tr>
<tr>
<td>October</td>
<td>January</td>
<td>April</td>
</tr>
<tr>
<td>October</td>
<td>January</td>
<td>April</td>
</tr>
</tbody>
</table>

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

```sql
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

<table>
<thead>
<tr>
<th>Promo_Desc</th>
<th>Year</th>
<th>Days_on_Promo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Christmas special</td>
<td>1999</td>
<td>31</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATEDIFF(day, '07-01-00', '01-01-00')</td>
<td>182</td>
</tr>
<tr>
<td>DATEDIFF(month, '07-01-00', '01-01-00')</td>
<td>6</td>
</tr>
<tr>
<td>DATEDIFF(quarter, '07-01-00', '01-01-00')</td>
<td>2</td>
</tr>
</tbody>
</table>

**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:

\[
\text{datediff(day, end_date, start_date)+1}
\]

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:

\[
\text{where p.start_date = d.date}
\]
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**

```sql
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**

<table>
<thead>
<tr>
<th>Day_Name</th>
<th>Day_Num</th>
<th>Day</th>
<th>Day_Yr</th>
<th>Mo_Name</th>
<th>Mo_Num</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thursday</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Friday</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Saturday</td>
<td>7</td>
<td>3</td>
<td>3</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Sunday</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Monday</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>January</td>
<td>1</td>
</tr>
</tbody>
</table>

(1 of 2)
<table>
<thead>
<tr>
<th>Day_Name</th>
<th>Day_Num</th>
<th>Day</th>
<th>Day_Yr</th>
<th>Mo_Name</th>
<th>Mo_Num</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuesday</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Wednesday</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Thursday</td>
<td>5</td>
<td>8</td>
<td>8</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Friday</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Saturday</td>
<td>7</td>
<td>10</td>
<td>10</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Sunday</td>
<td>1</td>
<td>11</td>
<td>11</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Monday</td>
<td>2</td>
<td>12</td>
<td>12</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Tuesday</td>
<td>3</td>
<td>13</td>
<td>13</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Wednesday</td>
<td>4</td>
<td>14</td>
<td>14</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Thursday</td>
<td>5</td>
<td>15</td>
<td>15</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Friday</td>
<td>6</td>
<td>16</td>
<td>16</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Saturday</td>
<td>7</td>
<td>17</td>
<td>17</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Sunday</td>
<td>1</td>
<td>18</td>
<td>18</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Monday</td>
<td>2</td>
<td>19</td>
<td>19</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Tuesday</td>
<td>3</td>
<td>20</td>
<td>20</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Wednesday</td>
<td>4</td>
<td>21</td>
<td>21</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Thursday</td>
<td>5</td>
<td>22</td>
<td>22</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Friday</td>
<td>6</td>
<td>23</td>
<td>23</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Saturday</td>
<td>7</td>
<td>24</td>
<td>24</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Sunday</td>
<td>1</td>
<td>25</td>
<td>25</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Monday</td>
<td>2</td>
<td>26</td>
<td>26</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Tuesday</td>
<td>3</td>
<td>27</td>
<td>27</td>
<td>January</td>
<td>1</td>
</tr>
</tbody>
</table>
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)

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>extract(weekday from date_col)</td>
<td>Weekday as an integer value from the set (1, 2, ..., 7)</td>
</tr>
<tr>
<td>extract(day from date_col)</td>
<td>Day of month as an integer from the set (1, 2, ..., 31)</td>
</tr>
</tbody>
</table>

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 database locale. For information about locale specifications, refer to the Administrator’s Guide and the Installation and Configuration Guide.
Summary

This chapter described how to:

- Use RISQL display functions and SQL OLAP functions to perform data analysis
- Redistribute ranked values into unequal groups
- Use DATETIME scalar functions to calculate and extract date information from DATETIME columns

Analytic Functions

SQL OLAP functions and RISQL display functions can be used to answer a wide range of business questions that require ranks, ratios, moving sums and averages, and so on. The examples in this chapter concentrate on calculations that can be handled by either an OLAP function or a RISQL function; however, in general, the OLAP functions are more flexible, especially with respect to their window frame functionality.

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 analytic functions.

For another use of the CASE expression, see “Using CASE Expressions” on page 4-7.
### DATETIME Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATEADD</td>
<td>Adds interval to datetime value.</td>
</tr>
<tr>
<td>DATEDIFF</td>
<td>Subtracts the difference between two datetime values.</td>
</tr>
<tr>
<td>DATENAME</td>
<td>Extracts datepart component from datetime value as character string.</td>
</tr>
<tr>
<td>EXTRACT</td>
<td>Extracts datepart from datetime value as integer.</td>
</tr>
</tbody>
</table>
Comparison Queries

In This Chapter .................................. 4-3
Comparing Data with SQL .................... 4-4
Using CASE Expressions ..................... 4-7
Using Subqueries in the FROM Clause ........ 4-10
Performing Calculations and Comparisons ... 4-13
Using Subqueries in the Select List .......... 4-15
Using Correlated Subqueries ................. 4-18
Using Cross-References ..................... 4-21
Calculating Percentages of Quarter and Year . 4-24
Using Subqueries in the WHERE Clause ..... 4-26
Using the ALL Comparison Predicate ....... 4-28
Using the EXISTS Predicate .................. 4-30
Using the SOME or ANY Predicate .......... 4-33
Summary ........................................ 4-36
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

```sql
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;
```
<table>
<thead>
<tr>
<th>Store_Name</th>
<th>Prod_Name</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beaches Brew</td>
<td>Aroma Roma</td>
<td>3483.50</td>
</tr>
<tr>
<td>Beaches Brew</td>
<td>Cafe Au Lait</td>
<td>3129.50</td>
</tr>
<tr>
<td>Beaches Brew</td>
<td>Colombiano</td>
<td>2298.25</td>
</tr>
<tr>
<td>Beaches Brew</td>
<td>Demitasse Ms</td>
<td>4529.25</td>
</tr>
<tr>
<td>Beaches Brew</td>
<td>Expresso XO</td>
<td>4132.75</td>
</tr>
<tr>
<td>Beaches Brew</td>
<td>La Antigua</td>
<td>4219.75</td>
</tr>
<tr>
<td>Beaches Brew</td>
<td>Lotta Latte</td>
<td>3468.00</td>
</tr>
<tr>
<td>Beaches Brew</td>
<td>NA Lite</td>
<td>4771.00</td>
</tr>
<tr>
<td>Beaches Brew</td>
<td>Veracruzano</td>
<td>4443.00</td>
</tr>
<tr>
<td>Beaches Brew</td>
<td>Xalapa Lapa</td>
<td>4304.00</td>
</tr>
<tr>
<td>Cupertino Coffee Supply</td>
<td>Aroma Roma</td>
<td>4491.00</td>
</tr>
<tr>
<td>Cupertino Coffee Supply</td>
<td>Cafe Au Lait</td>
<td>4375.50</td>
</tr>
<tr>
<td>Cupertino Coffee Supply</td>
<td>Colombiano</td>
<td>2653.50</td>
</tr>
<tr>
<td>Cupertino Coffee Supply</td>
<td>Demitasse Ms</td>
<td>3936.50</td>
</tr>
<tr>
<td>Cupertino Coffee Supply</td>
<td>Expresso XO</td>
<td>4689.25</td>
</tr>
<tr>
<td>Cupertino Coffee Supply</td>
<td>La Antigua</td>
<td>2932.00</td>
</tr>
<tr>
<td>Cupertino Coffee Supply</td>
<td>Lotta Latte</td>
<td>5146.00</td>
</tr>
<tr>
<td>Cupertino Coffee Supply</td>
<td>NA Lite</td>
<td>4026.00</td>
</tr>
<tr>
<td>Cupertino Coffee Supply</td>
<td>Veracruzano</td>
<td>3285.00</td>
</tr>
<tr>
<td>Cupertino Coffee Supply</td>
<td>Xalapa Lapa</td>
<td>5784.00</td>
</tr>
<tr>
<td>Instant Coffee</td>
<td>Aroma Roma</td>
<td>3485.25</td>
</tr>
<tr>
<td>Instant Coffee</td>
<td>Cafe Au Lait</td>
<td>3599.50</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Store_Name</th>
<th>Prod_Name</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instant Coffee</td>
<td>Colombiano</td>
<td>3321.75</td>
</tr>
<tr>
<td>Instant Coffee</td>
<td>Demitasse Ms</td>
<td>5422.25</td>
</tr>
<tr>
<td>Instant Coffee</td>
<td>Expresso XO</td>
<td>2851.00</td>
</tr>
<tr>
<td>Instant Coffee</td>
<td>La Antigua</td>
<td>2937.25</td>
</tr>
<tr>
<td>Instant Coffee</td>
<td>Lotta Latte</td>
<td>4783.50</td>
</tr>
<tr>
<td>Instant Coffee</td>
<td>NA Lite</td>
<td>3740.00</td>
</tr>
<tr>
<td>Instant Coffee</td>
<td>Veracruzano</td>
<td>4712.00</td>
</tr>
<tr>
<td>Instant Coffee</td>
<td>Xalapa Lapa</td>
<td>3698.00</td>
</tr>
</tbody>
</table>

...
Using CASE Expressions

Question

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

Example Query

```sql
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;
```
<table>
<thead>
<tr>
<th>Prod_Name</th>
<th>Beaches</th>
<th>Cupertino</th>
<th>RoastLG</th>
<th>SJRoastCo</th>
<th>JavaJudy</th>
<th>Instant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aroma Roma</td>
<td>3483.50</td>
<td>4491.00</td>
<td>4602.00</td>
<td>4399.25</td>
<td>3748.25</td>
<td>3485.25</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>3129.50</td>
<td>4375.50</td>
<td>4199.00</td>
<td>3620.00</td>
<td>4864.50</td>
<td>3599.50</td>
</tr>
<tr>
<td>Colombiano</td>
<td>2298.25</td>
<td>2653.50</td>
<td>4205.00</td>
<td>3530.75</td>
<td>3509.00</td>
<td>3321.75</td>
</tr>
<tr>
<td>Demitasse Ms</td>
<td>4529.25</td>
<td>3936.50</td>
<td>4347.75</td>
<td>5699.00</td>
<td>6395.25</td>
<td>5422.25</td>
</tr>
<tr>
<td>Expresso XO</td>
<td>4132.75</td>
<td>4689.25</td>
<td>4234.50</td>
<td>3811.00</td>
<td>5012.25</td>
<td>2851.00</td>
</tr>
<tr>
<td>La Antigua</td>
<td>4219.75</td>
<td>2932.00</td>
<td>3447.50</td>
<td>4323.00</td>
<td>2410.25</td>
<td>2937.25</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>3468.00</td>
<td>5146.00</td>
<td>4469.50</td>
<td>5103.50</td>
<td>4003.00</td>
<td>4783.50</td>
</tr>
<tr>
<td>NA Lite</td>
<td>4771.00</td>
<td>4026.00</td>
<td>3250.00</td>
<td>2736.00</td>
<td>4791.00</td>
<td>3740.00</td>
</tr>
<tr>
<td>Veracruzano</td>
<td>4443.00</td>
<td>3285.00</td>
<td>4467.00</td>
<td>3856.00</td>
<td>4510.00</td>
<td>4712.00</td>
</tr>
<tr>
<td>Xalapa Lapa</td>
<td>4304.00</td>
<td>5784.00</td>
<td>3906.00</td>
<td>3645.00</td>
<td>3182.00</td>
<td>3698.00</td>
</tr>
</tbody>
</table>
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 3-28.

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

```sql
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

```sql
select product, jan_98_sales, total_98_sales
from
(select pl.prod_name, sum(dollars) as jan_98_sales
from product pl 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 pl.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

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

...
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

```sql
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;
```
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:

\[ \frac{100 \times \text{monthly sales}}{\text{annual sales}} \]

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.

### Calculations with FROM Clause Subqueries

<table>
<thead>
<tr>
<th>Product</th>
<th>Jan_98_Sales</th>
<th>Total_98_Sales</th>
<th>Pct_of_98</th>
<th>Rank_Pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aroma Sheffield Steel Teapot</td>
<td>120.00</td>
<td>1122.00</td>
<td>10.69</td>
<td>4</td>
</tr>
<tr>
<td>Aroma t-shirt</td>
<td>470.85</td>
<td>4470.50</td>
<td>10.53</td>
<td>5</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>608.25</td>
<td>6394.75</td>
<td>9.51</td>
<td>9</td>
</tr>
<tr>
<td>Colombiano</td>
<td>2148.00</td>
<td>22528.50</td>
<td>9.53</td>
<td>8</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>867.50</td>
<td>8590.00</td>
<td>10.09</td>
<td>7</td>
</tr>
<tr>
<td>Espresso Machine Italiano</td>
<td>899.55</td>
<td>4397.80</td>
<td>20.45</td>
<td>1</td>
</tr>
<tr>
<td>Expresso XO</td>
<td>2935.50</td>
<td>27362.00</td>
<td>10.72</td>
<td>3</td>
</tr>
<tr>
<td>Irish Breakfast</td>
<td>703.25</td>
<td>7455.50</td>
<td>9.43</td>
<td>10</td>
</tr>
<tr>
<td>La Antigua</td>
<td>2643.25</td>
<td>22244.50</td>
<td>11.88</td>
<td>2</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>3195.00</td>
<td>31200.00</td>
<td>10.24</td>
<td>6</td>
</tr>
</tbody>
</table>
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:

\[
\text{dec}(100 \times \frac{\text{jan_98_sales}}{\text{total_98_sales}},7,2) \text{ as pct_of_98}
\]

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

\[
\text{rank(pct_of_98)} \text{ as rank_pct}
\]

For more examples of ranking 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

```sql
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

<table>
<thead>
<tr>
<th>Prod_Name</th>
<th>Store_Name</th>
<th>Date</th>
<th>Sales_99</th>
<th>Avg_98</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lotta Latte</td>
<td>San Jose Roasting Company</td>
<td>1999-12-09</td>
<td>153.00</td>
<td>154.72</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>San Jose Roasting Company</td>
<td>1999-12-28</td>
<td>144.50</td>
<td>154.72</td>
</tr>
</tbody>
</table>
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**

<table>
<thead>
<tr>
<th>Prod_Name</th>
<th>Jan_98_Sales</th>
<th>Total_98_Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aroma Roma</td>
<td>1653.00</td>
<td>21697.50</td>
</tr>
<tr>
<td>Aroma Sheffield Steel Teapot</td>
<td>120.00</td>
<td>1122.00</td>
</tr>
<tr>
<td>Aroma Sounds Cassette</td>
<td>58.50</td>
<td>866.00</td>
</tr>
<tr>
<td>Aroma baseball cap</td>
<td>7.95</td>
<td>2960.15</td>
</tr>
<tr>
<td>Aroma t-shirt</td>
<td>470.85</td>
<td>4470.50</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>652.00</td>
<td>11375.00</td>
</tr>
</tbody>
</table>

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

(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 4-10, 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;
```
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:

```sql
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:

```sql
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

```sql
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;
```
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**

```sql
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');
```
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.

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.
Using the ALL Comparison Predicate

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

```sql
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');
```
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.)

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

**Comparison Predicates in Subqueries**

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Evaluates to “true” when</th>
<th>When no value is returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>The comparison is true for all values returned by the subquery.</td>
<td>Evaluates to true.</td>
</tr>
<tr>
<td>ANY, SOME</td>
<td>The comparison is true for at least one of the values returned by the subquery.</td>
<td>Evaluates to false.</td>
</tr>
<tr>
<td>EXISTS</td>
<td>The subquery produces at least one row.</td>
<td>Evaluates to false.</td>
</tr>
</tbody>
</table>

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.

### Result

<table>
<thead>
<tr>
<th>Prod_Name</th>
<th>Date</th>
<th>Dollars</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA Lite</td>
<td>2000-01-24</td>
<td>414.00</td>
</tr>
</tbody>
</table>

**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.
Using the EXISTS Predicate

Usage Notes

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

```sql
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

```sql
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);
```
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.

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

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 3-44.)
**Usage Notes**

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

```sql
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:

```sql
... 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

```sql
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;
```
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.

### Result

<table>
<thead>
<tr>
<th>Supplier_Name</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aroma West Mfg.</td>
<td>4425.00</td>
</tr>
<tr>
<td>Espresso Express</td>
<td>30250.00</td>
</tr>
<tr>
<td>Espresso Express</td>
<td>25100.00</td>
</tr>
<tr>
<td>Espresso Express</td>
<td>26400.00</td>
</tr>
<tr>
<td>Espresso Express</td>
<td>22700.00</td>
</tr>
<tr>
<td>Western Emporium</td>
<td>10234.50</td>
</tr>
</tbody>
</table>
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 4-30.

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.
Joins and Unions

In This Chapter ........................................... 5-3
Join of Two Tables ........................................ 5-3
Different Ways to Join Tables ............................ 5-6
System Table Join .......................................... 5-9
Self-Joins ..................................................... 5-10
Outer Join of Two Tables ................................. 5-14
Fact-to-Fact Join ........................................... 5-17
Fact-to-Fact Join ........................................... 5-20
OR Versus UNION .......................................... 5-23
INTERSECT Operation .................................... 5-26
INTERSECT Operation Inside Subquery ................. 5-28
EXCEPT Operation ........................................ 5-30
Summary .................................................... 5-32
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

<table>
<thead>
<tr>
<th>State Table</th>
<th>Region Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>City</strong></td>
<td><strong>City</strong></td>
</tr>
<tr>
<td>Jacksonville</td>
<td>Jacksonville</td>
</tr>
<tr>
<td>Miami</td>
<td>Miami</td>
</tr>
<tr>
<td>Nashville</td>
<td>New Orleans</td>
</tr>
<tr>
<td>State</td>
<td>Area</td>
</tr>
<tr>
<td>FL</td>
<td>South</td>
</tr>
<tr>
<td>FL</td>
<td>South</td>
</tr>
<tr>
<td>TN</td>
<td>South</td>
</tr>
</tbody>
</table>
Example Query

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

**Cartesian Product (join predicate not specified)**

<table>
<thead>
<tr>
<th>City</th>
<th>State</th>
<th>City</th>
<th>Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacksonville</td>
<td>FL</td>
<td>Jacksonville</td>
<td>South</td>
</tr>
<tr>
<td>Jacksonville</td>
<td>FL</td>
<td>Miami</td>
<td>South</td>
</tr>
<tr>
<td>Jacksonville</td>
<td>FL</td>
<td>New Orleans</td>
<td>South</td>
</tr>
<tr>
<td>Miami</td>
<td>FL</td>
<td>Jacksonville</td>
<td>South</td>
</tr>
<tr>
<td>Miami</td>
<td>FL</td>
<td>Miami</td>
<td>South</td>
</tr>
<tr>
<td>Miami</td>
<td>FL</td>
<td>New Orleans</td>
<td>South</td>
</tr>
<tr>
<td>Nashville</td>
<td>TN</td>
<td>Jacksonville</td>
<td>South</td>
</tr>
<tr>
<td>Nashville</td>
<td>TN</td>
<td>Miami</td>
<td>South</td>
</tr>
<tr>
<td>Nashville</td>
<td>TN</td>
<td>New Orleans</td>
<td>South</td>
</tr>
</tbody>
</table>

Example Query

```sql
select * from state, region
where state.city = region.city;
```
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 $m \times n$ 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**

```sql
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

<table>
<thead>
<tr>
<th>Promo_Desc</th>
<th>Year</th>
<th>Sales</th>
<th>Days_on_Promo</th>
<th>Per_Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>Christmas special</td>
<td>1999</td>
<td>1230.00</td>
<td>31</td>
<td>39.67</td>
</tr>
<tr>
<td>Christmas special</td>
<td>1998</td>
<td>690.00</td>
<td>31</td>
<td>22.25</td>
</tr>
</tbody>
</table>

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)
Joins in the FROM Clause

**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 3-42.
- 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:

  \[
  \frac{\text{sales}}{\text{promo\_days}}
  \]

**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 3-43. For a complete discussion of join syntax, refer to the **SQL Reference Guide**.
What are the names of the segments and physical storage units (PSUs) used to store the Aroma Sales table?

```
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;
```

<table>
<thead>
<tr>
<th>Storage</th>
<th>PSU_Location</th>
<th>Table_Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT_SEGMENT_23</td>
<td>dfltseg23-psu1</td>
<td>SALES</td>
</tr>
<tr>
<td>DAILY_DATA1</td>
<td>sales-psu1</td>
<td>SALES</td>
</tr>
<tr>
<td>DAILY_DATA1</td>
<td>sales-psu2</td>
<td>SALES</td>
</tr>
<tr>
<td>DAILY_DATA2</td>
<td>sales-psu3</td>
<td>SALES</td>
</tr>
<tr>
<td>DAILY_DATA2</td>
<td>sales-psu4</td>
<td>SALES</td>
</tr>
</tbody>
</table>

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, IBM Red Brick Warehouse system tables can be joined in the same way as all other database tables.
Self-Joins

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:

```sql
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

```sql
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;
```
<table>
<thead>
<tr>
<th>Product</th>
<th>Pkg_Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aroma Roma</td>
<td>No pkg</td>
</tr>
<tr>
<td>Aroma Roma</td>
<td>One-pound bag</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>No pkg</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Qtr-pound bag</td>
</tr>
<tr>
<td>Assam Grade A</td>
<td>No pkg</td>
</tr>
<tr>
<td>Assam Grade A</td>
<td>Qtr-pound bag</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>No pkg</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>Qtr-pound bag</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>No pkg</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>One-pound bag</td>
</tr>
<tr>
<td>Colombiano</td>
<td>No pkg</td>
</tr>
<tr>
<td>Colombiano</td>
<td>One-pound bag</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>No pkg</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>Qtr-pound bag</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>No pkg</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>Qtr-pound bag</td>
</tr>
<tr>
<td>Demitasse Ms</td>
<td>No pkg</td>
</tr>
<tr>
<td>Demitasse Ms</td>
<td>One-pound bag</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>No pkg</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Qtr-pound bag</td>
</tr>
<tr>
<td>English Breakfast</td>
<td>No pkg</td>
</tr>
<tr>
<td>English Breakfast</td>
<td>Qtr-pound bag</td>
</tr>
</tbody>
</table>
### Product | Pkg_Type
---|---
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
... | ...
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:

```sql
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:

```sql
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)

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

Result

<table>
<thead>
<tr>
<th>State:City</th>
<th>State:State</th>
<th>Region:City</th>
<th>Region:Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacksonville</td>
<td>FL</td>
<td>Jacksonville</td>
<td>South</td>
</tr>
<tr>
<td>Miami</td>
<td>FL</td>
<td>Miami</td>
<td>South</td>
</tr>
<tr>
<td>Nashville</td>
<td>TN</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Example Query (right outer join)

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

Result

<table>
<thead>
<tr>
<th>State:City</th>
<th>State:State</th>
<th>Region:City</th>
<th>Region:Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacksonville</td>
<td>FL</td>
<td>Jacksonville</td>
<td>South</td>
</tr>
<tr>
<td>Miami</td>
<td>FL</td>
<td>Miami</td>
<td>South</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>New Orleans</td>
<td>South</td>
</tr>
</tbody>
</table>
Example Query (full outer join)

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

### Result

<table>
<thead>
<tr>
<th>State:City</th>
<th>State:State</th>
<th>Region:City</th>
<th>Region:Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacksonville</td>
<td>FL</td>
<td>Jacksonville</td>
<td>South</td>
</tr>
<tr>
<td>Miami</td>
<td>FL</td>
<td>Miami</td>
<td>South</td>
</tr>
<tr>
<td>Nashville</td>
<td>TN</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>New Orleans</td>
<td>South</td>
</tr>
</tbody>
</table>

**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.
Outer Joins

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

```sql
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

<table>
<thead>
<tr>
<th>Order_Num</th>
<th>Order_Type</th>
<th>Full_Cost</th>
<th>Line_Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>3619</td>
<td>Tea</td>
<td>4325.25</td>
<td>725.25</td>
</tr>
<tr>
<td>3619</td>
<td>Tea</td>
<td>4325.25</td>
<td>400.00</td>
</tr>
<tr>
<td>3619</td>
<td>Tea</td>
<td>4325.25</td>
<td>400.00</td>
</tr>
<tr>
<td>3619</td>
<td>Tea</td>
<td>4325.25</td>
<td>400.00</td>
</tr>
<tr>
<td>3619</td>
<td>Tea</td>
<td>4325.25</td>
<td>400.00</td>
</tr>
<tr>
<td>3619</td>
<td>Tea</td>
<td>4325.25</td>
<td>400.00</td>
</tr>
<tr>
<td>3619</td>
<td>Tea</td>
<td>4325.25</td>
<td>400.00</td>
</tr>
<tr>
<td>3619</td>
<td>Tea</td>
<td>4325.25</td>
<td>400.00</td>
</tr>
<tr>
<td>3619</td>
<td>Tea</td>
<td>4325.25</td>
<td>400.00</td>
</tr>
<tr>
<td>3619</td>
<td>Tea</td>
<td>4325.25</td>
<td>NULL</td>
</tr>
<tr>
<td>3620</td>
<td>Tea</td>
<td>4325.25</td>
<td>NULL</td>
</tr>
<tr>
<td>3621</td>
<td>Spice</td>
<td>10234.50</td>
<td>10234.50</td>
</tr>
<tr>
<td>3622</td>
<td>Spice</td>
<td>10234.50</td>
<td>10234.50</td>
</tr>
<tr>
<td>3623</td>
<td>Hardware</td>
<td>4425.00</td>
<td>400.00</td>
</tr>
<tr>
<td>3623</td>
<td>Hardware</td>
<td>4425.00</td>
<td>400.00</td>
</tr>
<tr>
<td>3623</td>
<td>Hardware</td>
<td>4425.00</td>
<td>500.00</td>
</tr>
<tr>
<td>3623</td>
<td>Hardware</td>
<td>4425.00</td>
<td>450.00</td>
</tr>
<tr>
<td>3623</td>
<td>Hardware</td>
<td>4425.00</td>
<td>500.00</td>
</tr>
<tr>
<td>3623</td>
<td>Hardware</td>
<td>4425.00</td>
<td>275.00</td>
</tr>
<tr>
<td>3623</td>
<td>Hardware</td>
<td>4425.00</td>
<td>650.00</td>
</tr>
<tr>
<td>3623</td>
<td>Hardware</td>
<td>4425.00</td>
<td>1250.00</td>
</tr>
<tr>
<td>3624</td>
<td>Hardware</td>
<td>4425.00</td>
<td>400.00</td>
</tr>
<tr>
<td>3624</td>
<td>Hardware</td>
<td>4425.00</td>
<td>500.00</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Order_Num</th>
<th>Order_Type</th>
<th>Full_Cost</th>
<th>Line_Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>3624</td>
<td>Hardware</td>
<td>4425.00</td>
<td>450.00</td>
</tr>
<tr>
<td>3624</td>
<td>Hardware</td>
<td>4425.00</td>
<td>400.00</td>
</tr>
<tr>
<td>3624</td>
<td>Hardware</td>
<td>4425.00</td>
<td>500.00</td>
</tr>
<tr>
<td>3624</td>
<td>Hardware</td>
<td>4425.00</td>
<td>275.00</td>
</tr>
<tr>
<td>3624</td>
<td>Hardware</td>
<td>4425.00</td>
<td>650.00</td>
</tr>
<tr>
<td>3624</td>
<td>Hardware</td>
<td>4425.00</td>
<td>1250.00</td>
</tr>
<tr>
<td>3625</td>
<td>Clothing</td>
<td>3995.95</td>
<td>2500.00</td>
</tr>
<tr>
<td>3625</td>
<td>Clothing</td>
<td>3995.95</td>
<td>1495.95</td>
</tr>
<tr>
<td>3626</td>
<td>Hardware</td>
<td>16500.00</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Fact-to-Fact Join

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**

```sql
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;
```
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.
Full Outer Join with ORDER BY, BREAK BY

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

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

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**

```sql
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:

```sql
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%'
intersection
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;
```
You can use the INTERSECT operator to return only those rows that are common to the results returned by two or more query expressions.

**Finding Common Rows: INTERSECT**

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.

**About the Query**

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**

```sql
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;
```
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.

<table>
<thead>
<tr>
<th>Product</th>
<th>Cost_of_Orders</th>
<th>Revenue_North</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aroma Roma</td>
<td>7300.00</td>
<td>3190.00</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>7300.00</td>
<td>3975.50</td>
</tr>
<tr>
<td>Colombiano</td>
<td>7300.00</td>
<td>3907.50</td>
</tr>
<tr>
<td>Demitasse Ms</td>
<td>8500.00</td>
<td>6081.25</td>
</tr>
<tr>
<td>Expresso XO</td>
<td>7300.00</td>
<td>4218.50</td>
</tr>
<tr>
<td>La Antigua</td>
<td>7300.00</td>
<td>3510.50</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>7300.00</td>
<td>4273.00</td>
</tr>
<tr>
<td>NA Lite</td>
<td>7300.00</td>
<td>6480.00</td>
</tr>
<tr>
<td>Veracruzano</td>
<td>7300.00</td>
<td>4055.00</td>
</tr>
<tr>
<td>Xalapa Lapa</td>
<td>7300.00</td>
<td>6896.50</td>
</tr>
</tbody>
</table>
**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**

```sql
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;
```
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:

```
e except_cities(city)
```

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

### Result

<table>
<thead>
<tr>
<th>City</th>
<th>Store_Name</th>
<th>Sales_99</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cupertino</td>
<td>Cupertino Coffee Supply</td>
<td>196439.75</td>
</tr>
<tr>
<td>Los Gatos</td>
<td>Roasters, Los Gatos</td>
<td>175048.75</td>
</tr>
</tbody>
</table>

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

```sql
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**

```sql
query_expression
UNION | INTERSECT | EXCEPT [ALL]
query_expression
[ORDER BY order_list]
[SUPPRESS BY suppress_list];
```
Macros, Views, and Temporary Tables

In This Chapter ........................................... 6-3
Basic Macros ............................................. 6-4
Embedded Macros ........................................ 6-7
Macros with Parameters ................................. 6-10
Multiparameter Macros ................................. 6-13
Comparisons .............................................. 6-15
Share Comparisons ....................................... 6-18
Change in Share .......................................... 6-20
Views ...................................................... 6-22
INSERT INTO SELECT Statement ...................... 6-25
Summary .................................................. 6-28
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

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

Example Query

```sql
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

<table>
<thead>
<tr>
<th>Prod_Name</th>
<th>Class</th>
<th>Sales_99</th>
</tr>
</thead>
<tbody>
<tr>
<td>Darjeeling Special</td>
<td>Bulk Tea</td>
<td>80610.50</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>Pkg Tea</td>
<td>51266.00</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Bulk Tea</td>
<td>42329.00</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>Bulk Tea</td>
<td>34592.75</td>
</tr>
<tr>
<td>Irish Breakfast</td>
<td>Bulk Tea</td>
<td>27763.75</td>
</tr>
<tr>
<td>Prod_Name</td>
<td>Class</td>
<td>Sales_99</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------</td>
<td>-----------</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Pkg Tea</td>
<td>27192.50</td>
</tr>
<tr>
<td>English Breakfast</td>
<td>Bulk Tea</td>
<td>25848.00</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>Bulk Tea</td>
<td>24594.00</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>Pkg Tea</td>
<td>24232.00</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Bulk Tea</td>
<td>23269.50</td>
</tr>
<tr>
<td>Special Tips</td>
<td>Bulk Tea</td>
<td>22326.00</td>
</tr>
<tr>
<td>Assam Grade A</td>
<td>Bulk Tea</td>
<td>21964.00</td>
</tr>
<tr>
<td>Gold Tips</td>
<td>Bulk Tea</td>
<td>21584.50</td>
</tr>
<tr>
<td>Irish Breakfast</td>
<td>Pkg Tea</td>
<td>20084.00</td>
</tr>
<tr>
<td>English Breakfast</td>
<td>Pkg Tea</td>
<td>18955.00</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>Pkg Tea</td>
<td>17031.50</td>
</tr>
<tr>
<td>Gold Tips</td>
<td>Pkg Tea</td>
<td>16783.25</td>
</tr>
<tr>
<td>Special Tips</td>
<td>Pkg Tea</td>
<td>16773.25</td>
</tr>
<tr>
<td>Assam Grade A</td>
<td>Pkg Tea</td>
<td>16724.00</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Pkg Tea</td>
<td>16108.00</td>
</tr>
</tbody>
</table>

(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**

```sql
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:

```sql
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**

```sql
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**

```sql
tea_totals;
```
<table>
<thead>
<tr>
<th>Prod_Name</th>
<th>Class</th>
<th>Sales_99</th>
</tr>
</thead>
<tbody>
<tr>
<td>Darjeeling Special</td>
<td>Bulk Tea</td>
<td>80610.50</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>Pkg Tea</td>
<td>51266.00</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Bulk Tea</td>
<td>42329.00</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>Bulk Tea</td>
<td>34592.75</td>
</tr>
<tr>
<td>Irish Breakfast</td>
<td>Bulk Tea</td>
<td>27763.75</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Pkg Tea</td>
<td>27192.50</td>
</tr>
<tr>
<td>English Breakfast</td>
<td>Bulk Tea</td>
<td>25848.00</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>Bulk Tea</td>
<td>24594.00</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>Pkg Tea</td>
<td>24232.00</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Bulk Tea</td>
<td>23269.50</td>
</tr>
<tr>
<td>Special Tips</td>
<td>Bulk Tea</td>
<td>22326.00</td>
</tr>
<tr>
<td>Assam Grade A</td>
<td>Bulk Tea</td>
<td>21964.00</td>
</tr>
<tr>
<td>Gold Tips</td>
<td>Bulk Tea</td>
<td>21584.50</td>
</tr>
<tr>
<td>Irish Breakfast</td>
<td>Pkg Tea</td>
<td>20084.00</td>
</tr>
<tr>
<td>English Breakfast</td>
<td>Pkg Tea</td>
<td>18955.00</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>Pkg Tea</td>
<td>17031.50</td>
</tr>
<tr>
<td>Gold Tips</td>
<td>Pkg Tea</td>
<td>16783.25</td>
</tr>
<tr>
<td>Special Tips</td>
<td>Pkg Tea</td>
<td>16773.25</td>
</tr>
<tr>
<td>Assam Grade A</td>
<td>Pkg Tea</td>
<td>16724.00</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Pkg Tea</td>
<td>16108.00</td>
</tr>
</tbody>
</table>
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.
What were the total sales of tea products during any given year?

**CREATE MACRO Statement**

```sql
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**

```sql
tea_sales(1998);
```
### Result

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

```sql
create macro top_rank(yr, locn, nbr) as
select prod_name, city, year, sum(dollars) as sales,
    rank() over (order by sales desc) 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 ranking <= nbr;
```

Example Query 1 and Result

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

<table>
<thead>
<tr>
<th>Prod_Name</th>
<th>City</th>
<th>Year</th>
<th>Sales</th>
<th>Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xalapa Lapa</td>
<td>Los Angeles</td>
<td>1998</td>
<td>14930.00</td>
<td>1</td>
</tr>
<tr>
<td>Demitasse Ms</td>
<td>Los Angeles</td>
<td>1998</td>
<td>14402.25</td>
<td>2</td>
</tr>
<tr>
<td>Ruby's Allspice</td>
<td>Los Angeles</td>
<td>1998</td>
<td>14339.00</td>
<td>3</td>
</tr>
<tr>
<td>Aroma Roma</td>
<td>Los Angeles</td>
<td>1998</td>
<td>14253.25</td>
<td>4</td>
</tr>
<tr>
<td>Expresso XO</td>
<td>Los Angeles</td>
<td>1998</td>
<td>13179.50</td>
<td>5</td>
</tr>
</tbody>
</table>
Example Query 2 and Result

top_rank(1999, ‘San Jose’, 1);

<table>
<thead>
<tr>
<th>Prod_Name</th>
<th>City</th>
<th>Year</th>
<th>Sales</th>
<th>Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demitasse Ms</td>
<td>San Jose</td>
<td>1999</td>
<td>32887.75</td>
<td>1</td>
</tr>
</tbody>
</table>

Example Query 3 and Result

top_rank(2000, ‘Hartford’, 3);

<table>
<thead>
<tr>
<th>Prod_Name</th>
<th>City</th>
<th>Year</th>
<th>Sales</th>
<th>Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA Lite</td>
<td>Hartford</td>
<td>2000</td>
<td>5061.00</td>
<td>1</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>Hartford</td>
<td>2000</td>
<td>4665.00</td>
<td>2</td>
</tr>
<tr>
<td>Xalapa Lapa</td>
<td>Hartford</td>
<td>2000</td>
<td>4610.00</td>
<td>3</td>
</tr>
</tbody>
</table>

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 used in this query is an SQL OLAP function. For more examples of OLAP functions, see 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**

```sql
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;

<table>
<thead>
<tr>
<th>Prod_Name</th>
<th>Month</th>
<th>Sales_99</th>
<th>Sales_00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lotta Latte</td>
<td>JAN</td>
<td>1611.00</td>
<td>3475.00</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>FEB</td>
<td>3162.50</td>
<td>2409.50</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>MAR</td>
<td>2561.50</td>
<td>2831.50</td>
</tr>
</tbody>
</table>

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;

<table>
<thead>
<tr>
<th>Prod_Name</th>
<th>Month</th>
<th>Sales_99</th>
<th>Sales_00</th>
<th>Qty_99</th>
<th>Qty_00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lotta Latte</td>
<td>JAN</td>
<td>1611.00</td>
<td>3475.00</td>
<td>197</td>
<td>426</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>FEB</td>
<td>3162.50</td>
<td>2409.50</td>
<td>391</td>
<td>298</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>MAR</td>
<td>2561.50</td>
<td>2831.50</td>
<td>314</td>
<td>348</td>
</tr>
</tbody>
</table>
**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:

\[ \text{lotta\_sales(facts, yr)} \]

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:

\[ \text{lotta\_sales(dollars, year+1)} \]

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

\[ \text{year+1} \]

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

```sql
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

```sql
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_qtr_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;
```
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:

\[
100 \times \left( \frac{\text{monthly_sales}}{\text{quarterly_sales}} \right)
\]

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:

\[
lotta_sales(\text{facts, yr})
\]

The following macro is another subquery that calculates, in this case, quarterly sales dollars for the specified year:

\[
lotta_qtr_sales(\text{facts, yr})
\]
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

```sql
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

<table>
<thead>
<tr>
<th>Prod_Name</th>
<th>Month</th>
<th>Sales_99</th>
<th>Sales_00</th>
<th>Sales_Chg</th>
<th>Share_Chg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lotta Latte</td>
<td>JAN</td>
<td>1611.00</td>
<td>3475.00</td>
<td>115.70</td>
<td>17.90</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>FEB</td>
<td>3162.50</td>
<td>2409.50</td>
<td>-23.81</td>
<td>-15.47</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>MAR</td>
<td>2561.50</td>
<td>2831.50</td>
<td>10.54</td>
<td>-2.43</td>
</tr>
</tbody>
</table>

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:

\[
100\times\frac{(\text{monthly\_sales\_00} - \text{monthly\_sales\_99})}{\text{monthly\_sales\_99}}
\]

Similarly, the change in the product share for a quarter can be calculated as follows:

\[
100\times\frac{(\text{monthly\_sales\_00}/\text{quarterly\_sales\_00})}{(\text{monthly\_sales\_99}/\text{quarterly\_sales\_99})}
\]
Neither percentage is difficult to calculate, but macros simplify the construction of queries that return 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.

Question

What were the sales totals and ranks by store for Assam Gold Blend tea in 1999?

CREATE VIEW Statement

```sql
create view tea_sales99
    as select prod_name, store_name, sum(dollars) as tea_dollars, rank() over(order by tea_dollars desc) 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

```sql
select prod_name, store_name, tea_dollars, tea_rank
    from tea_sales99
    where prod_name like 'Assam Gold%';
```
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.

### Result

<table>
<thead>
<tr>
<th>Prod_Name</th>
<th>Store_Name</th>
<th>Tea_Dollars</th>
<th>Tea_Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assam Gold Blend</td>
<td>Beans of Boston</td>
<td>6201.50</td>
<td>15</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Beaches Brew</td>
<td>6080.00</td>
<td>16</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Texas Teahouse</td>
<td>5422.50</td>
<td>17</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Olympic Coffee Company</td>
<td>5350.50</td>
<td>18</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Cupertino Coffee Supply</td>
<td>5277.00</td>
<td>19</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Moroccan Moods</td>
<td>5178.50</td>
<td>20</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Coffee Brewers</td>
<td>5151.00</td>
<td>21</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Moulin Rouge Roasting</td>
<td>4977.00</td>
<td>22</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>East Coast Roast</td>
<td>4769.00</td>
<td>25</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Miami Espresso</td>
<td>4506.50</td>
<td>28</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Roasters, Los Gatos</td>
<td>4414.50</td>
<td>29</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>San Jose Roasting Company</td>
<td>4226.50</td>
<td>32</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>4190.50</td>
<td>33</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Java Judy’s</td>
<td>3776.50</td>
<td>40</td>
</tr>
</tbody>
</table>

### 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 *IBM Red Brick 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 constrain 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.
FUNCTIONAL INDEX

**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;
```

**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;
```
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 analytic functions in temporary tables and issue SELECT statements against them to further constrain the result data.

**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.

<table>
<thead>
<tr>
<th>Date</th>
<th>Prod_Name</th>
<th>Dollars</th>
<th>Cume_Tot</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-08</td>
<td>Aroma t-shirt</td>
<td>197.10</td>
<td>308</td>
</tr>
<tr>
<td>2000-01-18</td>
<td>Aroma t-shirt</td>
<td>131.40</td>
<td>131</td>
</tr>
<tr>
<td>2000-01-18</td>
<td>Aroma baseball cap</td>
<td>135.15</td>
<td>266</td>
</tr>
<tr>
<td>2000-01-23</td>
<td>Aroma baseball cap</td>
<td>15.90</td>
<td>15</td>
</tr>
<tr>
<td>2000-02-01</td>
<td>Aroma t-shirt</td>
<td>175.20</td>
<td>175</td>
</tr>
<tr>
<td>2000-02-04</td>
<td>Aroma t-shirt</td>
<td>164.25</td>
<td>164</td>
</tr>
</tbody>
</table>
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 IBM Red Brick Warehouse 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
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

```sql
select * from class;
```

Result

<table>
<thead>
<tr>
<th>Classkey</th>
<th>Class_Type</th>
<th>Class_Desc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bulk_beans</td>
<td>Bulk coffee products</td>
</tr>
<tr>
<td>2</td>
<td>Bulk_tea</td>
<td>Bulk tea products</td>
</tr>
<tr>
<td>3</td>
<td>Bulk_spice</td>
<td>Bulk spices</td>
</tr>
<tr>
<td>4</td>
<td>Pkg_coffee</td>
<td>Individually packaged coffee products</td>
</tr>
<tr>
<td>5</td>
<td>Pkg_tea</td>
<td>Individually packaged tea products</td>
</tr>
<tr>
<td>6</td>
<td>Pkg_spice</td>
<td>Individually packaged spice products</td>
</tr>
<tr>
<td>7</td>
<td>Hardware</td>
<td>Coffee mugs, teapots, spice jars, espresso machines</td>
</tr>
<tr>
<td>8</td>
<td>Gifts</td>
<td>Samplers, gift boxes and baskets, etc.</td>
</tr>
<tr>
<td>12</td>
<td>Clothing</td>
<td>T-shirts, caps, etc.</td>
</tr>
</tbody>
</table>

Query

```sql
select * from product;
```
### Result

<table>
<thead>
<tr>
<th>Classkey</th>
<th>Prodkey</th>
<th>Prod_Name</th>
<th>Pkg_Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>Veracruzano</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Xalapa Lapa</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>Colombiano</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>Expresso XO</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>La Antigua</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>Lotta Latte</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>Cafe Au Lait</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>22</td>
<td>NA Lite</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
<td>Aroma Roma</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>31</td>
<td>Demitasse Ms</td>
<td>No pkg</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>Darjeeling Number 1</td>
<td>No pkg</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Darjeeling Special</td>
<td>No pkg</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>Assam Grade A</td>
<td>No pkg</td>
</tr>
</tbody>
</table>

### 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

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>classkey</td>
<td>Integer that identifies exactly one row in the Class table. Classkey is the primary key.</td>
</tr>
<tr>
<td>class_type</td>
<td>Character string that identifies a group of products.</td>
</tr>
<tr>
<td>class_desc</td>
<td>Character string that describes a group of products.</td>
</tr>
</tbody>
</table>

### Column Descriptions: Product Table

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>classkey</td>
<td>Foreign-key reference to the Class table.</td>
</tr>
<tr>
<td>prodkey</td>
<td>Integer that combines with the Classkey value to identify exactly one row in the Product table. Classkey/Prodkey is a two-column primary key.</td>
</tr>
<tr>
<td>prod_name</td>
<td>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.</td>
</tr>
<tr>
<td>pkg_type</td>
<td>Character string that identifies the type of packaging for each product.</td>
</tr>
</tbody>
</table>
Sample Data from the Store and Market Tables

Query

```sql
select * from market;
```

Result

<table>
<thead>
<tr>
<th>Mktkey</th>
<th>HQ_City</th>
<th>HQ_State</th>
<th>District</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Atlanta</td>
<td>GA</td>
<td>Atlanta</td>
<td>South</td>
</tr>
<tr>
<td>2</td>
<td>Miami</td>
<td>FL</td>
<td>Atlanta</td>
<td>South</td>
</tr>
<tr>
<td>3</td>
<td>New Orleans</td>
<td>LA</td>
<td>New Orleans</td>
<td>South</td>
</tr>
<tr>
<td>4</td>
<td>Houston</td>
<td>TX</td>
<td>New Orleans</td>
<td>South</td>
</tr>
<tr>
<td>5</td>
<td>New York</td>
<td>NY</td>
<td>New York</td>
<td>North</td>
</tr>
</tbody>
</table>

... 

Query

```sql
select * from store;
```
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

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>mktkey</td>
<td>Integer that identifies exactly one row in the Market table. Mktkey is the primary key.</td>
</tr>
<tr>
<td>hq_city</td>
<td>Character string that identifies a city. The Market table defines 17 cities. A fully populated database could have thousands.</td>
</tr>
<tr>
<td>state</td>
<td>Character string that identifies a state.</td>
</tr>
<tr>
<td>district</td>
<td>Character string that identifies a district based on a major metropolitan city. A global database would contain countries and nations or other geographic dimensions.</td>
</tr>
<tr>
<td>region</td>
<td>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.</td>
</tr>
</tbody>
</table>

### Store Table—Column Descriptions

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>storekey</td>
<td>Integer that identifies exactly one row in the Store table. Storekey is the primary key.</td>
</tr>
<tr>
<td>mktkey</td>
<td>Foreign-key reference to the Market table.</td>
</tr>
<tr>
<td>store_type</td>
<td>Character string that identifies stores by size.</td>
</tr>
<tr>
<td>store_name</td>
<td>Character string that identifies a store by name.</td>
</tr>
<tr>
<td>street, city, state, zip</td>
<td>Columns that identify the address of each store.</td>
</tr>
</tbody>
</table>
Sample Data from the Period Table

Query

```sql
select * from period;
```
### Result

<table>
<thead>
<tr>
<th>Perkey</th>
<th>Date</th>
<th>Day</th>
<th>Week</th>
<th>Month</th>
<th>Qtr</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1998-01-01</td>
<td>TH</td>
<td>1</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>2</td>
<td>1998-01-02</td>
<td>FR</td>
<td>1</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>3</td>
<td>1998-01-03</td>
<td>SA</td>
<td>1</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>4</td>
<td>1998-01-04</td>
<td>SU</td>
<td>2</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>5</td>
<td>1998-01-05</td>
<td>MO</td>
<td>2</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>6</td>
<td>1998-01-06</td>
<td>TU</td>
<td>2</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>7</td>
<td>1998-01-07</td>
<td>WE</td>
<td>2</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>8</td>
<td>1998-01-08</td>
<td>TH</td>
<td>2</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>9</td>
<td>1998-01-09</td>
<td>FR</td>
<td>2</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>10</td>
<td>1998-01-10</td>
<td>SA</td>
<td>2</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>11</td>
<td>1998-01-11</td>
<td>SU</td>
<td>3</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>12</td>
<td>1998-01-12</td>
<td>MO</td>
<td>3</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>13</td>
<td>1998-01-13</td>
<td>TU</td>
<td>3</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>14</td>
<td>1998-01-14</td>
<td>WE</td>
<td>3</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>15</td>
<td>1998-01-15</td>
<td>TH</td>
<td>3</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>16</td>
<td>1998-01-16</td>
<td>FR</td>
<td>3</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>17</td>
<td>1998-01-17</td>
<td>SA</td>
<td>3</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>18</td>
<td>1998-01-18</td>
<td>SU</td>
<td>4</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>19</td>
<td>1998-01-19</td>
<td>MO</td>
<td>4</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
<tr>
<td>20</td>
<td>1998-01-20</td>
<td>TU</td>
<td>4</td>
<td>JAN</td>
<td>Q1_98</td>
<td>1998</td>
</tr>
</tbody>
</table>

...
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

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>perkey</td>
<td>Integer that identifies exactly one row in the Period table. Perkey is the primary key.</td>
</tr>
<tr>
<td>date</td>
<td>Date value that identifies each day from January 1, 1998 through March 31, 2000.</td>
</tr>
<tr>
<td>day</td>
<td>Character-string abbreviation of the day of the week.</td>
</tr>
<tr>
<td>week</td>
<td>Integer that identifies each week of each year by number (1 through 53, each new week starting on a Sunday).</td>
</tr>
<tr>
<td>month</td>
<td>Character-string abbreviation of the name of each month.</td>
</tr>
<tr>
<td>qtr</td>
<td>Character string that uniquely identifies each quarter (for example, Q1_98, Q3_99).</td>
</tr>
<tr>
<td>year</td>
<td>Integer that identifies the year.</td>
</tr>
</tbody>
</table>

Sample Data from the Promotion Table

Query

```sql
    select * from promotion;
```
**Result**

<table>
<thead>
<tr>
<th>Promokey</th>
<th>Promo_Type</th>
<th>Promo_Desc</th>
<th>Value</th>
<th>Start_Date</th>
<th>End_Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>No promotion</td>
<td>0.00</td>
<td>9999-01-01</td>
<td>9999-01-01</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1998-01-01</td>
<td>1998-01-31</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1998-02-01</td>
<td>1998-02-28</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1998-03-01</td>
<td>1998-03-31</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1998-04-01</td>
<td>1998-04-30</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1998-05-01</td>
<td>1998-05-31</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1998-06-01</td>
<td>1998-06-30</td>
</tr>
<tr>
<td>7</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1998-07-01</td>
<td>1998-07-31</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1998-08-01</td>
<td>1998-08-31</td>
</tr>
<tr>
<td>9</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1998-09-01</td>
<td>1998-09-30</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1998-10-01</td>
<td>1998-10-31</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1998-11-01</td>
<td>1998-11-30</td>
</tr>
<tr>
<td>12</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1998-12-01</td>
<td>1998-12-31</td>
</tr>
<tr>
<td>13</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1999-01-01</td>
<td>1999-01-31</td>
</tr>
<tr>
<td>14</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1999-02-01</td>
<td>1999-02-28</td>
</tr>
<tr>
<td>15</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1999-03-01</td>
<td>1999-03-31</td>
</tr>
<tr>
<td>16</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1999-04-01</td>
<td>1999-04-30</td>
</tr>
<tr>
<td>17</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1999-05-01</td>
<td>1999-05-31</td>
</tr>
<tr>
<td>18</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1999-06-01</td>
<td>1999-06-30</td>
</tr>
<tr>
<td>19</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1999-07-01</td>
<td>1999-07-31</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1999-08-01</td>
<td>1999-08-31</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>promokey</td>
<td>Integer that identifies exactly one row in the Promotion table. Promokey is the primary key.</td>
</tr>
<tr>
<td>promo_type</td>
<td>Integer that identifies the promotion by number (or code).</td>
</tr>
<tr>
<td>promo_desc</td>
<td>Character string that describes the promotion type.</td>
</tr>
<tr>
<td>value</td>
<td>Decimal number that represents the dollar value of the promotion, such as a price reduction or the value of a coupon.</td>
</tr>
<tr>
<td>start_date, end_date</td>
<td>Date values that indicate when each promotion begins and ends.</td>
</tr>
</tbody>
</table>
Sample Data from the Sales Table

Query

```sql
select * from sales;
```
<table>
<thead>
<tr>
<th>Perkey</th>
<th>Classkey</th>
<th>Prodkey</th>
<th>Storekey</th>
<th>Promokey</th>
<th>Quantity</th>
<th>Dollars</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>116</td>
<td>8</td>
<td>34.00</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>12</td>
<td>1</td>
<td>116</td>
<td>9</td>
<td>60.75</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>116</td>
<td>40</td>
<td>270.00</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>30</td>
<td>1</td>
<td>116</td>
<td>16</td>
<td>36.00</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>22</td>
<td>1</td>
<td>116</td>
<td>11</td>
<td>30.25</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>30</td>
<td>1</td>
<td>116</td>
<td>30</td>
<td>187.50</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>116</td>
<td>25</td>
<td>143.75</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>12</td>
<td>87.00</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>11</td>
<td>2</td>
<td>0</td>
<td>14</td>
<td>115.50</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>22</td>
<td>2</td>
<td>0</td>
<td>18</td>
<td>58.50</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>17</td>
<td>136.00</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>13</td>
<td>74.75</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>30</td>
<td>2</td>
<td>0</td>
<td>14</td>
<td>101.50</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>18</td>
<td>63.00</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>22</td>
<td>3</td>
<td>0</td>
<td>11</td>
<td>99.00</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>46</td>
<td>3</td>
<td>0</td>
<td>6</td>
<td>36.00</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>12</td>
<td>3</td>
<td>0</td>
<td>10</td>
<td>40.00</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>11</td>
<td>3</td>
<td>0</td>
<td>36</td>
<td>279.00</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>11</td>
<td>132.00</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>10</td>
<td>3</td>
<td>0</td>
<td>12</td>
<td>48.00</td>
</tr>
</tbody>
</table>

...
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.
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.

### Column Descriptions

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>perkey</td>
<td>Foreign-key reference to the Period table.</td>
</tr>
<tr>
<td>classkey</td>
<td>Foreign-key reference to the Product table.</td>
</tr>
<tr>
<td>prodkey</td>
<td>Foreign-key reference to the Product table.</td>
</tr>
<tr>
<td>storekey</td>
<td>Foreign-key reference to the Store table.</td>
</tr>
<tr>
<td>promokey</td>
<td>Foreign-key reference to the Promotion table.</td>
</tr>
<tr>
<td>quantity</td>
<td>Integer that represents the total quantity sold (per day).</td>
</tr>
<tr>
<td>dollars</td>
<td>Decimal number that represents dollar sales figures (per day).</td>
</tr>
</tbody>
</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:

<table>
<thead>
<tr>
<th>Order_No</th>
<th>Perkey</th>
<th>Supkey</th>
<th>Dealkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>3699</td>
<td>817</td>
<td>1007</td>
<td>0</td>
</tr>
<tr>
<td>3700</td>
<td>817</td>
<td>1007</td>
<td>0</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Supkey</th>
<th>Type</th>
<th>Name</th>
<th>Street</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Bulk coffee</td>
<td>CB Imports</td>
<td>100 Church St</td>
<td>Mountain View</td>
<td>CA</td>
<td>94001</td>
</tr>
<tr>
<td>1002</td>
<td>Bulk tea</td>
<td>Tea Makers,</td>
<td>1555 Hicks Rd.</td>
<td>San Jose</td>
<td>CA</td>
<td>95124</td>
</tr>
</tbody>
</table>

Some columns have been truncated to fit on the page.

Query

```
select * from deal;
```
Result

<table>
<thead>
<tr>
<th>Dealkey</th>
<th>Deal_Type</th>
<th>Deal_Desc</th>
<th>Discount</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>No deal</td>
<td>0.00</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>Orders over $10,000</td>
<td>100.00</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>Orders over $20,000</td>
<td>500.00</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>Supplier catalog coupon</td>
<td>50.00</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>Supplier catalog coupon</td>
<td>100.00</td>
</tr>
<tr>
<td>37</td>
<td>200</td>
<td>Supplier coffee special</td>
<td>75.00</td>
</tr>
<tr>
<td>38</td>
<td>200</td>
<td>Supplier coffee special</td>
<td>50.00</td>
</tr>
<tr>
<td>39</td>
<td>200</td>
<td>Supplier tea special</td>
<td>40.00</td>
</tr>
<tr>
<td>40</td>
<td>200</td>
<td>Supplier tea special</td>
<td>20.00</td>
</tr>
</tbody>
</table>

The Supplier and Deal Tables

Column Descriptions: Supplier Table

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>supkey</td>
<td>Integer that identifies exactly one row in the Supplier table. Supkey is the primary key.</td>
</tr>
<tr>
<td>type</td>
<td>Character string that indicates the type of products supplied.</td>
</tr>
<tr>
<td>name</td>
<td>Character string that identifies the supplier by name.</td>
</tr>
<tr>
<td>street, city, state, zip</td>
<td>Columns that identify the address of the supplier.</td>
</tr>
</tbody>
</table>
The Supplier and Deal Tables

**Column Descriptions: Deal Table**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>dealkey</td>
<td>Integer that identifies exactly one row in the Deal table. Dealkey is the primary key.</td>
</tr>
<tr>
<td>deal_type</td>
<td>Integer that identifies the type of deal (a code number).</td>
</tr>
<tr>
<td>deal_desc</td>
<td>Character string that describes the type of deal.</td>
</tr>
<tr>
<td>discount</td>
<td>Decimal value that indicates the dollar amount of the deal applied to an order.</td>
</tr>
</tbody>
</table>

**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

```sql
select * from orders;
```

Result

<table>
<thead>
<tr>
<th>Order_No</th>
<th>Perkey</th>
<th>Supkey</th>
<th>Dealkey</th>
<th>Order_Type</th>
<th>Order_Desc</th>
<th>Close_Date</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>3600</td>
<td>731</td>
<td>1001</td>
<td>37</td>
<td>Coffee</td>
<td>Whole coffee b</td>
<td>2000-01-07</td>
<td>1200.46</td>
</tr>
<tr>
<td>3601</td>
<td>732</td>
<td>1001</td>
<td>37</td>
<td>Coffee</td>
<td>Whole coffee b</td>
<td>2000-01-07</td>
<td>1535.94</td>
</tr>
<tr>
<td>3602</td>
<td>733</td>
<td>1001</td>
<td>0</td>
<td>Tea</td>
<td>Loose tea, bul</td>
<td>2000-01-07</td>
<td>780.00</td>
</tr>
<tr>
<td>3603</td>
<td>740</td>
<td>1001</td>
<td>39</td>
<td>Tea</td>
<td>Loose tea, bul</td>
<td>2000-01-21</td>
<td>956.45</td>
</tr>
<tr>
<td>3604</td>
<td>744</td>
<td>1005</td>
<td>0</td>
<td>Spice</td>
<td>Pre-packed spi</td>
<td>2000-01-16</td>
<td>800.66</td>
</tr>
<tr>
<td>3605</td>
<td>768</td>
<td>1003</td>
<td>2</td>
<td>Coffee</td>
<td>Whole-bean and</td>
<td>2000-02-12</td>
<td>25100.00</td>
</tr>
<tr>
<td>3606</td>
<td>775</td>
<td>1003</td>
<td>2</td>
<td>Coffee</td>
<td>Whole-bean and</td>
<td>2000-02-19</td>
<td>25100.00</td>
</tr>
<tr>
<td>3607</td>
<td>782</td>
<td>1003</td>
<td>2</td>
<td>Coffee</td>
<td>Whole-bean and</td>
<td>2000-02-25</td>
<td>25100.00</td>
</tr>
<tr>
<td>3608</td>
<td>789</td>
<td>1003</td>
<td>2</td>
<td>Coffee</td>
<td>Whole-bean and</td>
<td>2000-03-03</td>
<td>30250.00</td>
</tr>
<tr>
<td>3609</td>
<td>796</td>
<td>1003</td>
<td>2</td>
<td>Coffee</td>
<td>Whole-bean and</td>
<td>2000-03-15</td>
<td>25100.00</td>
</tr>
</tbody>
</table>

Query

```sql
select * from line_items;
```
Result

<table>
<thead>
<tr>
<th>Order_No</th>
<th>Line_Item</th>
<th>Perkey</th>
<th>Classkey</th>
<th>Prodkey</th>
<th>Receive_Da</th>
<th>Qty</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>3600</td>
<td>1</td>
<td>731</td>
<td>1</td>
<td>1</td>
<td>2000-01-07</td>
<td>40</td>
<td>180.46</td>
</tr>
<tr>
<td>3600</td>
<td>2</td>
<td>731</td>
<td>2</td>
<td>10</td>
<td>2000-01-07</td>
<td>150</td>
<td>300.00</td>
</tr>
<tr>
<td>3600</td>
<td>3</td>
<td>731</td>
<td>2</td>
<td>11</td>
<td>2000-01-07</td>
<td>80</td>
<td>240.00</td>
</tr>
<tr>
<td>3600</td>
<td>4</td>
<td>731</td>
<td>2</td>
<td>12</td>
<td>2000-01-07</td>
<td>150</td>
<td>240.00</td>
</tr>
<tr>
<td>3600</td>
<td>5</td>
<td>731</td>
<td>1</td>
<td>20</td>
<td>2000-01-07</td>
<td>60</td>
<td>240.00</td>
</tr>
<tr>
<td>3601</td>
<td>1</td>
<td>732</td>
<td>1</td>
<td>0</td>
<td>2000-01-07</td>
<td>60</td>
<td>240.00</td>
</tr>
<tr>
<td>3601</td>
<td>2</td>
<td>732</td>
<td>1</td>
<td>1</td>
<td>2000-01-07</td>
<td>60</td>
<td>240.00</td>
</tr>
<tr>
<td>3601</td>
<td>3</td>
<td>732</td>
<td>1</td>
<td>10</td>
<td>2000-01-07</td>
<td>60</td>
<td>240.00</td>
</tr>
<tr>
<td>3601</td>
<td>4</td>
<td>732</td>
<td>1</td>
<td>11</td>
<td>2000-01-07</td>
<td>60</td>
<td>240.00</td>
</tr>
<tr>
<td>3601</td>
<td>5</td>
<td>732</td>
<td>1</td>
<td>12</td>
<td>2000-01-07</td>
<td>60</td>
<td>240.00</td>
</tr>
<tr>
<td>3601</td>
<td>6</td>
<td>732</td>
<td>1</td>
<td>31</td>
<td>2000-01-07</td>
<td>70</td>
<td>335.94</td>
</tr>
<tr>
<td>3602</td>
<td>1</td>
<td>733</td>
<td>2</td>
<td>0</td>
<td>2000-01-08</td>
<td>70</td>
<td>130.00</td>
</tr>
<tr>
<td>3602</td>
<td>2</td>
<td>733</td>
<td>2</td>
<td>1</td>
<td>2000-01-08</td>
<td>70</td>
<td>130.00</td>
</tr>
</tbody>
</table>

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-18.
### Column Descriptions: Orders Table

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>order_no</td>
<td>Integer that identifies exactly one row in the <strong>Orders</strong> table. <strong>Order_No</strong> is the primary key.</td>
</tr>
<tr>
<td>perkey</td>
<td>Foreign-key reference to the <strong>Period</strong> table.</td>
</tr>
<tr>
<td>supkey</td>
<td>Foreign-key reference to the <strong>Supplier</strong> table.</td>
</tr>
<tr>
<td>dealkey</td>
<td>Foreign-key reference to the <strong>Deal</strong> table.</td>
</tr>
<tr>
<td>order_type</td>
<td>Character string that defines the types of products on the order.</td>
</tr>
<tr>
<td>order_desc</td>
<td>Character string that describes the type of order.</td>
</tr>
<tr>
<td>close_date</td>
<td>Date value that identifies when the order was completed or closed.</td>
</tr>
<tr>
<td>price</td>
<td>Decimal value that indicates the full cost of the order.</td>
</tr>
</tbody>
</table>

### Column Descriptions: Line_Items Table

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>order_no</td>
<td>Integer that identifies exactly one row in the <strong>Orders</strong> table. <strong>Order_No</strong> is the primary key.</td>
</tr>
<tr>
<td>line_item</td>
<td>Integer that identifies each item listed on the order by number.</td>
</tr>
<tr>
<td>perkey</td>
<td>Foreign-key reference to the <strong>Period</strong> table.</td>
</tr>
<tr>
<td>classkey</td>
<td>Foreign-key reference to the <strong>Product</strong> table.</td>
</tr>
<tr>
<td>prodkey</td>
<td>Foreign-key reference to the <strong>Product</strong> table.</td>
</tr>
<tr>
<td>receive_date</td>
<td>Date value that identifies when the line-item was received.</td>
</tr>
<tr>
<td>quantity</td>
<td>Integer that indicates the quantity of products ordered for each line-item.</td>
</tr>
<tr>
<td>price</td>
<td>Decimal value that indicates the cost of the line-item.</td>
</tr>
</tbody>
</table>
IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user’s responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan
The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.
Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.
Trademarks

AIX; DB2; DB2 Universal Database; Distributed Relational Database Architecture; NUMA-Q; OS/2, OS/390, and OS/400; IBM Informix®, C-ISAM®, Foundation.2000™; IBM Informix® 4GL; IBM Informix® DataBlade® Module; Client SDK™; Cloudscape™; Cloudsync™; IBM Informix® Connect; IBM Informix® Driver for JDBC; Dynamic Connect™; IBM Informix® Dynamic Scalable Architecture™ (DSA); IBM Informix® Dynamic Server™; IBM Informix® Enterprise Gateway Manager (Enterprise Gateway Manager); IBM Informix® Extended Parallel Server™; i.Financial Services™; J/Foundation™; MaxConnect™; Object Translator™; Red Brick™; IBM Informix® SE; IBM Informix® SQL; InformiXML™; RedBack®; SystemBuilder™; U2™; UniData®; UniVerse®; wintegrate® are trademarks or registered trademarks of International Business Machines Corporation.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Windows, Windows NT, and Excel are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names used in this publication may be trademarks or service marks of others.
Index

A
Addition operator 2-45
Aggregate queries 2-39
Aggregate tables
not in Aroma database 1-5
queries that require 3-3
Aggregation functions
OLAP 3-7
set functions 2-33
Aliases, column 2-27, 2-36
ALL comparison predicate 2-20,
4-29
Analytic functions 3-4
AND connective 2-14
ANY comparison predicate 2-20,
4-34
Arithmetic operators, listed 2-45
Aroma database 1-4, A-2
Class table 1-6, A-4
column descriptions A-6 to A-26
Deal table A-21
dimension tables
purchasing schema A-20
retail schema A-4
fact tables
purchasing schema A-25
retail schema 1-9, A-15
illustrated
purchasing schema A-19
retail schema 1-4, A-2
Line_Items table A-24
Market table 1-8, A-7
Orders table A-24
Period table 1-6, A-10
Product table 1-6, A-4
Promotion table 1-9, A-12
purchasing schema A-18
retail schema 1-4, A-2
Sales table 1-11, A-15
Store table 1-8, A-7
Supplier table A-21
AS keyword, for column
aliases 2-36
ASC keyword 2-27
Ascending order, sorting results
in 2-27
Averages, moving 3-16
AVG function
SQL OLAP function 3-14
SQL set function 2-33

B
BETWEEN predicate 2-20
Boldface type 5
BREAK BY subclause
of ORDER BY clause 2-31
outer join with 5-22

C
Calculations
aggregate functions 2-33
BREAK BY subtotals 2-31
datetime functions 3-40
RISQL and OLAP 3-3
share 6-21
with subqueries 4-14
CASE expressions
in comparison queries 4-9
in macros 6-6
with NTILE function 3-30
Cases tracked by Technical Support 8
Change in share, macros that calculate 6-21
Characters
literals 2-13
patterns 2-22
wildcard 2-22
Child queries. See Subqueries.
Class table, Aroma database 1-6, A-4
COALESCE function 5-19
Columns
aliases 2-27, 2-36
in Aroma tables A-6 to A-26 selecting 2-9
Comparison macros 6-17
Comparison operators, list of 2-18
Comparison predicates 2-20
ALL 2-20, 4-29
ANY 2-20, 4-34
BETWEEN 2-20
EXISTS 2-20, 4-31
IN 2-20
in subqueries 4-29
IS NOT NULL 2-20
IS NULL 2-20
LIKE 2-20
NOT EXISTS 4-32
SOME 2-20, 4-34
Comparison queries 4-3 to 4-36 macros for 6-15
OLAP 3-11
share 6-17, 6-18
Conditions
compound 2-14, 2-16
correlation 4-20
cross-references as 4-20
Connectives, logical 2-14
Conventions
syntax 6
typographical 5
Correlated subqueries 4-20
Correlation names
derived tables 4-12
tables 4-20
COUNT function 2-34
CREATE MACRO statement 6-4,
6-6, 6-7, 6-10, 6-13
CREATE TEMPORARY TABLE
statement 6-25
CREATE VIEW statement 6-23
Cross-references, as correlation conditions 4-20
Cumulative totals calculating 3-7
resetting 3-10
Customer Support 7

D
Data list A-19
Databases
Aroma 1-4 to 1-11, A-1 to A-26
decision-support 1-5
dateadd function 3-40
datediff function 3-43, 5-8
datename function 3-41
Dates, incrementing or decrementing 3-40
Datetime scalar functions 3-41
deal table, Aroma database A-21
dec function 2-52
decision-support data analysis introduction 1-3
distinct keyword in select list 2-9
in set functions 2-34
division operator 2-45
documentation list for IBM Red Brick Warehouse 9
online 11

E
Embedded macros 6-9
Environment variables 5
Equijoins, inner 5-5
except operator example 5-31
syntax 5-24
exists predicate 2-20, 4-31
Explicit tables 2-6
Expressions, as cross-references 4-22
extract function 3-46, 4-31

F
Fact tables, Aroma database 1-9, A-15, A-25
Fact-to-fact joins 5-17
Foreign-key references 1-11, A-17
from clause
basic syntax 2-6
joins in 5-5
subqueries in 4-12
Full outer joins 5-15, 5-21
Functions. See Display functions, RISQL; RISQL functions; OLAP functions; and Scalar functions.

G
Generalized macros 6-12
GROUP BY clause 2-38, 2-41
Named-columns joins 5-7
Natural joins 2-25, 5-8
Negative operator (-) 2-45
NOT connective 2-14
NOT EXISTS predicate 4-32
NTILE function
inside a CASE expression 3-30
ranking values in groups 3-27
NULL comparison predicate 2-20
NULLs ignored by set functions 2-34
removing with SUPPRESS BY clause 2-51
Numeric literals 2-13

OLAP functions
AVG 3-14
DENSE_RANK 3-22
NTILE 3-25, 3-28, 3-34
ORDER BY clause 3-13
OVER() clause 3-7
PARTITION BY clause 3-10
RANK 3-20, 3-23
RATIOTOREPORT 3-35
ROW_NUMBER 3-7
SUM 3-7, 3-8, 3-11, 3-17
WHEN clause 3-23

Operators
arithmetic 2-45
comparison 2-18
UNION, EXCEPT, and INTERSECT 5-24

Order of precedence
arithmetic operators 2-45
logical connectives 2-14
parentheses 2-17, 2-45
Orders table, Aroma database A-24
Outer joins 5-15
Outer queries 4-12
Outer references. See Cross-references.

Parameterized macros 6-12
Parent queries 4-12
Parentheses, order of precedence 2-45
PARTITION BY clause, OLAP 3-10
Percentages
calculating with RATIOTOREPORT 3-36
change in share 6-21
monthly, yearly 4-25
Period table, Aroma database 1-6, A-10
Positive operator (+) 2-45

Predicates
ALL 2-20
ANY 2-20
BETWEEN 2-20
comparison 2-18, 2-20
EXISTS 2-20
IN 2-20
IS NOT NULL 2-20
IS NULL 2-20
LIKE 2-20, 2-22
SOME 2-20
wildcard characters 2-22

Primary keys 1-11, A-17
Product table, Aroma database 1-6, A-4
Promotion table, Aroma database 1-9, A-12
Purchase schema, Aroma database A-18

Qualified column names 4-22
Query expressions
CREATE VIEW statements 6-24
flexibility of 4-12
FROM clause 4-12
SELECT statements 2-5
UNION, INTERSECT, and
EXCEPT operators 5-24
Query processing, order of 2-38
Query rewrite system, Vista 2-39

Rank function 3-21
Ranking
top ten 3-25
values as high, middle, low 3-33
values into unequal groups 3-27
Ratios, calculating 3-36
RATIOTOREPORT function 3-36
Resetting cumulative totals 3-10
Retail schema, Aroma database 1-4, A-2
Right outer joins 5-15
RISQL functions
AVG 2-33
COALESCE 5-19
COUNT 2-34
DATEADD 3-40
DATEDIFF 3-43
DATENAME 3-41
DEC 2-52
display functions 3-3, 3-7, 3-16, 3-37, 3-47
DISTINCT 2-34
EXTRACT 3-46
MAX 2-33
MIN 2-33
MOVINGAVG 3-16
MOVINGSUM 3-19
NTILE 3-27
RANK 3-21
RATIOTOREPORT 3-36
set functions 2-33
STRING 2-46, 5-8
SUBSTR 6-20
SUM 2-33
TERTILE 3-33
See also Scalar functions, OLAP functions, and Display functions.
RISQL macros. See Macros, RISQL. ROWS PRECEDING/
FOLLOWING clause 3-7
Rows, selecting 2-12
ROW_NUMBER function 3-7
Running totals, calculating 3-7

S
Sales table, Aroma database 1-11, A-15
Sample database, Aroma 1-4, A-2
Scalar functions
COALESCE 5-19
DATEADD 3-40
DATEDIFF 5-8
DATENAME 3-41
DEC 2-52
EXTRACT 3-46, 4-31
STRING 2-46, 5-8
SUBSTR 6-20
Scalar subqueries 4-17
Schemas
Aroma database 1-5, A-3
multistar A-19
Segments, for Aroma Sales table 1-11, A-17
Select expressions. See Query expressions.
SELECT statement
BREAK BY subclause 2-31, 5-22
FROM clause 2-6
joins in 5-7
subqueries in 4-12
GROUP BY clause 2-38, 2-41
HAVING clause 2-47
ORDER BY clause 2-27, 5-22
processing order 2-38
summary of syntax 2-38
SUPPRESS BY clause 2-51
WHEN clause 3-24, 3-30
WHERE clause
description 2-12
join specification in 2-24
subqueries in 4-27
Select-list subqueries 4-17
Self-joins 5-13
Set functions 2-33
Share calculations
macros for 6-19
subqueries for 4-14, 4-25
Shared dimensions, in outer join queries 5-20
Share, change 6-20
SOME predicate 2-20, 4-34
Sorting result sets 2-27
Spaces, removing with SUPPRESS BY clause 2-51
SQL OLAP functions. See OLAP functions 3-3
STAR indexes A-17
Store table, Aroma database 1-8, A-7
STRING function, for truncating numeric values 2-46, 5-8
Subqueries
comparison predicates in 4-29
correlated 4-20
defined 4-12
FROM clause 4-12
select-list 4-17
spreadsheet-style comparisons 4-9, 4-17
WHERE clause 4-27
SUBSTR function 6-20
Subtotals, calculating 2-31
Subtraction operator 2-45
SUM function
moving sums 3-19
SQL OLAP function 3-7
SQL set function 2-33
Supplier table, Aroma database A-21
SUPPRESS BY clause 2-51
Syntax notation 6
System tables, joining 5-9

T
Table expressions. See Query expressions.
Tables
aggregate 1-5
Aroma database 1-4 to 1-11, A-2 to A-26
joining 2-24, 5-5
system 5-9
temporary 6-26, 6-27
See also Derived tables.
Technical support 7
Temporary tables, creating 6-26
TERTILE function 3-33
Top-ten rankings 3-25
Troubleshooting 8

U
Unary operators 2-45
UNION, INTERSECT, and EXCEPT operators
eamples 5-23 to 5-32
syntax 5-24
UNION versus OR 5-23
Users, types of 3

V
Views
creating 6-22
selecting from 6-23
Vista query rewrite system 2-39

W
WHEN clause
example with NTILE 3-30
example with RANK 3-24
WHERE clause 2-24
subqueries 4-27
syntax 2-12
versus HAVING clause 2-48
Wildcard characters 2-22
Window frames 3-7
Window partitions, OLAP 3-10
Z

Zeroes, removing with SUPPRESS
  BY clause 2-51

Symbols

( ), for order of precedence 2-45
%, SQL wildcard character 2-22
+, arithmetic operator 2-45
-, arithmetic operator 2-45
_, SQL wildcard character 2-22