SQL Reference Guide

Informix Red Brick Decision Server

Version 6.0
November 1999
Part No. 000-6364

Published by Informix® Press

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

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

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

Documentation Team: Erin Cizina, Kathy Eckardt, Evelyn Eldridge, Robyn King-Nitschke, Jerry Tattershall

GOVERNMENT LICENSE RIGHTS

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 of Defense use, with rights as restricted by vendor's standard license, unless superseded by a negotiated vendor license, as prescribed in DFARS 227.7202.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

Table of Contents

Introduction

In This Introduction . . . 3
About This Guide . . . 3
Types of Users . . . 3
Software Dependencies . . . 4
New Features . . . 5
Documentation Conventions . . . 5
Syntax Notation . . . 6
Syntax Diagrams . . . 7
Keywords and Punctuation . . . 9
Identifiers and Names . . . 9
Icon Conventions . . . 10
Customer Support . . . 11
New Cases . . . 12
Existing Cases . . . 12
Troubleshooting Tips . . . 12
Related Documentation . . . 13
Additional Documentation . . . 16
Online Manuals . . . 16
Printed Manuals . . . 16
Informix Welcomes Your Comments . . . 17

Chapter 1 Overview of Red Brick Decision Server

In This Chapter . . . 1-3
Relational Database Tables . . . 1-4
Multiple Users and Table Locks . . . 1-5
Views of Tables . . . 1-6
Indexes . . . 1-7
Primary and Foreign Keys . . . 1-7
Database Integrity . . . 1-9

Authorization and Privileges 1-9
The System Catalog 1-10
Red Brick Decision Server 1-10
Schema Design 1-11
Specialized Decision-Support Functions 1-11
Comparisons With Subqueries 1-12
Macros 1-12
Segmented Storage 1-13
Localization 1-13
Aroma Database 1-14

Chapter 2 Elements of the Language

In This Chapter 2-3
Names and Identifiers 2-3
Standard Identifiers 2-4
Delimited Identifiers 2-5
Uppercase and Lowercase. 2-6
Aliases and Correlation Names 2-7
Literals 2-10
Character Literals 2-11
Datetime Literals 2-12
Integer Constant 2-14
Decimal Constant 2-15
Floating-Point Constant 2-16
Datatypes 2-18
CHARACTER 2-19
DATETIME Datatypes: DATE, TIME, and TIMESTAMP . . . 2-21
INTEGER 2-23
SMALLINT. 2-23
TINYINT 2-23
DECIMAL and NUMERIC 2-24
REAL 2-25
DOUBLE PRECISION and FLOAT. 2-25
Missing Values and NULLs 2-26
Assignment and Comparison 2-27
Assignment. 2-27
Comparison 2-29

Chapter 3 Expressions and Conditions

In This Chapter 3-3
Expressions 3-4
    Simple Expression 3-4
    Compound Expressions 3-5
    Evaluation of Compound Expressions 3-7
Conditions 3-10
    Comparison Predicates 3-11
    BETWEEN Predicate 3-13
    EXISTS Predicate 3-14
    IN Predicate 3-15
    IS NULL Predicate. 3-16
    LIKE Predicate 3-17
    Search Condition 3-19

Chapter 4 Set Functions

In This Chapter 4-3
    AVG. 4-4
    COUNT 4-6
    MAX 4-8
    MIN. 4-9
    SUM 4-10

Chapter 5 Scalar Functions

In This Chapter 5-3
Conditional Scalar Functions 5-4
    CASE 5-4
    COALESCE 5-8
    DECODE 5-9
    IFNULL 5-11
    NULLIF 5-13
Numeric Scalar Functions 5-15
    ABS 5-15
    CEIL 5-17
    DEC. 5-19
    EXP 5-20
    FLOAT. 5-22
    FLOOR. 5-23
    INT 5-25
    LN 5-27

REAL 5-28
SIGN 5-29
SQRT 5-31
Macros for Statistical Functions 5-32
String Scalar Functions 5-35
CONCAT 5-36
LENGTH 5-37
LENGTHB 5-39
LOWER 5-40
LTRIM 5-41
RTRIM 5-42
STRING 5-44
SUBSTR 5-47
SUBSTRB 5-49
TRIM 5-51
UPPER 5-52
Datetime Scalar Functions 5-54
Dateparts for Datetime Scalar Functions 5-54
CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP 5-56
DATE 5-57
DATEADD 5-58
DATEDIFF 5-59
DATENAME 5-61
EXTRACT 5-63
TIME 5-65
TIMESTAMP 5-66
CURRENT_USER Function 5-67

Chapter 6 RISQL Display Functions

In This Chapter 6-3
CUME 6-5
MOVINGAVG 6-9
MOVINGSUM 6-12
NTILE 6-15
RANK 6-19
RATIOTOREPORT 6-22
TERTILE 6-24

Chapter 7 Query Expressions

In This Chapter . . . 7-3
Join and Non-Join Query Expressions . . . 7-3
Joined Tables . . . 7-6
  Syntax . . . 7-7
  Qualified Joins . . . 7-7
  Cross Joins . . . 7-12
  Table References . . . 7-13
Query Specifications . . . 7-16
  Select List . . . 7-17
  FROM Clause . . . 7-21
  WHERE Clause . . . 7-26
  GROUP BY Clause . . . 7-29
  HAVING Clause . . . 7-32
  WHEN Clause . . . 7-34
UNION, EXCEPT, and INTERSECT Expressions . . . 7-36
SELECT Statements . . . 7-46
  ORDER BY Clause . . . 7-47
  SUPPRESS BY Clause . . . 7-58
  How SELECT Statements Are Processed . . . 7-59
Subqueries . . . 7-60
  Scalar Subqueries and Table Subqueries . . . 7-61
  Correlated Subqueries . . . 7-64

Chapter 8 SQL Commands and RISQL Extensions

In This Chapter . . . 8-5
ALTER DATABASE . . . 8-6
ALTER INDEX . . . 8-14
ALTER MACRO . . . 8-19
ALTER ROLE . . . 8-21
ALTER SEGMENT . . . 8-22
  ALTER SEGMENT—ATTACH Clause . . . 8-23
  ALTER SEGMENT—Other Clauses . . . 8-30
ALTER SYNONYM . . . 8-50
ALTER SYSTEM . . . 8-51
ALTER TABLE . . . 8-62
  Add Column Specification . . . 8-67
  Drop Column Specification . . . 8-70
  Alter Column Specification . . . 8-72
  Add Constraint Specification . . . 8-75

Drop Constraint Specification 8-79
Alter Constraint Specification 8-80
ALTER USER 8-83
ALTER VIEW 8-84
CHECK INDEX 8-86
CHECK TABLE 8-89
CREATE HIERARCHY 8-92
CREATE INDEX 8-96
Index Specifier 8-102
Segment Specification 8-106
Segment Range Specification. 8-109
CREATE MACRO 8-118
CREATE ROLE 8-125
CREATE SEGMENT 8-126
CREATE SYNONYM 8-131
CREATE TABLE 8-132
Column Definitions 8-135
Primary-Key Reference. 8-139
Foreign-Key Reference 8-141
Primary-Key and Foreign-Key Constraint Names 8-144
Segment Specification 8-145
CREATE TEMPORARY TABLE 8-154
CREATE VIEW 8-157
DELETE 8-166
DROP HIERARCHY 8-171
DROP INDEX 8-172
DROP MACRO 8-175
DROP ROLE 8-177
DROP SEGMENT 8-178
DROP SYNONYM 8-180
DROP TABLE 8-181
DROP VIEW 8-183
EXPAND 8-184
EXPLAIN 8-185
EXPORT 8-187
GRANT Authorization and Role 8-192
GRANT CONNECT 8-198
GRANT Privilege 8-202
INSERT 8-205

LOCK Table 8-212
LOCK DATABASE 8-214
REVOKE Authorization and Role 8-216
REVOKE CONNECT 8-218
REVOKE Privilege 8-219
SELECT 8-222
UNLOCK Table 8-222
UNLOCK DATABASE 8-223
UPDATE 8-224

Chapter 9 SET Commands

In This Chapter 9-5
SET ADVISOR LOGGING 9-6
SET ARITHIGNORE, ARITHABORT 9-7
SET AUTO INVALIDATE PRECOMPUTED VIEWS 9-8
SET COUNT RESULT 9-9
SET CROSS JOIN 9-10
SET DEFAULT DATA SEGMENT 9-11
SET DEFAULT INDEX SEGMENT 9-13
SET EXPORT_DEFAULT_PATH 9-15
SET EXPORT_MAX_FILE_SIZE 9-16
SET FIRST DAYOFWEEK 9-17
SET FORCE TASKS 9-18
    FORCE_SCAN_TASKS 9-20
    FORCE_FETCH_TASKS and FORCE_JOIN_TASKS 9-21
    FORCE_HASHJOIN_TASKS 9-23
    FORCE_AGGREGATION_TASKS 9-24
SET IGNORE OPTICAL INDEXES 9-24
SET IGNORE PARTIAL INDEXES 9-26
SET INDEX TEMPSPACE and SET QUERY TEMPSPACE . . . 9-27
SET INFO MESSAGE LIMIT 9-31
SET LOCK 9-32
SET OPTICAL AVAILABILITY 9-33
SET ORDER BY 9-36
SET PARALLEL_HASHJOIN 9-37
SET PARTIAL AVAILABILITY 9-38
SET PRECOMPUTED VIEW view_name 9-39
SET PRECOMPUTED VIEW QUERY REWRITE 9-40
SET PRECOMPUTED VIEWS FOR detail_table 9-41

SET QUERY MEMORY LIMIT 9-41
SET QUERYPROCS 9-43
SET REPORT_INTERVAL 9-44
SET RESULT BUFFER and SET RESULT BUFFER FULL ACTION . 9-45
SET ROWCOUNT 9-46
SET ROWS_PER...TASK 9-48
ROWS_PER_SCAN_TASK 9-48
ROWS_PER_FETCH_TASK and ROWS_PER_JOIN_TASK . . 9-49
SET SEGMENTS 9-50
SET STATS 9-52
SET TEMPORARY SEGMENT STORAGE PATH 9-54
SET TRANSACTION ISOLATION LEVEL 9-55
SET UNIFORM PROBABILITY FOR ADVISOR 9-56
SET USE INVALID PRECOMPUTED VIEWS 9-57
SET USE LATEST REVISION 9-58
SET VERSIONING 9-59

Appendix A Syntax Summary

Appendix B Reserved Words

Appendix C Alternative Datetime Formats

Index

Introduction

In This Introduction 3

About This Guide 3
Types of Users 3
Software Dependencies 4

New Features 5

Documentation Conventions 5
Syntax Notation 6
Syntax Diagrams 7
Keywords and Punctuation 9
Identifiers and Names 9
Icon Conventions 10
Comment Icons 10
Platform Icons 10

Customer Support 11
New Cases 12
Existing Cases 12
Troubleshooting Tips 12

Related Documentation 13

Additional Documentation 16
Online Manuals 16
Printed Manuals 16

Informix Welcomes Your Comments 17

In This Introduction

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

About This Guide

This guide is a complete language reference for the Informix Red Brick SQL implementation and RISQL extensions for warehouse databases.

Types of Users

This guide is written for the following users:

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

This guide assumes that you have the following background:

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

Software Dependencies

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

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

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

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

New Features

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

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

Documentation Conventions

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

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

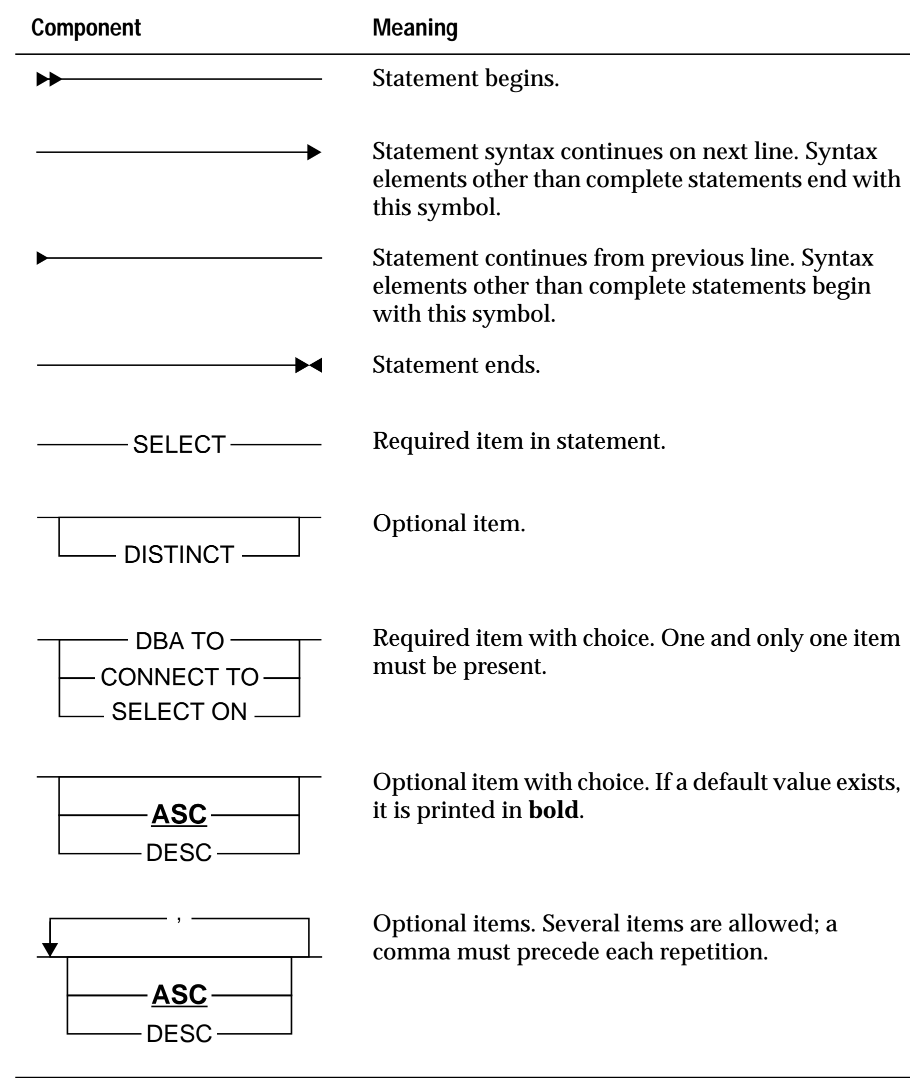
Syntax Notation

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

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

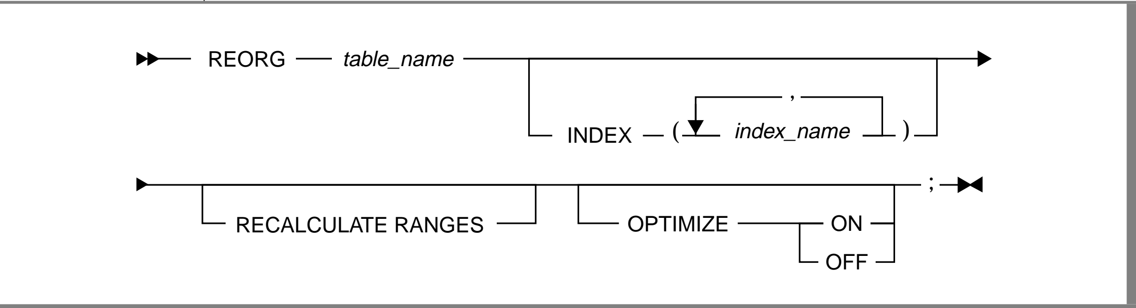
Syntax Diagrams

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

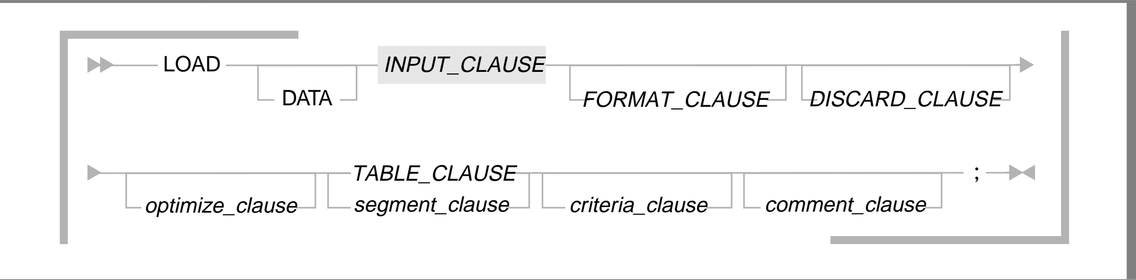


Component	Meaning
	Statement begins.
	Statement syntax continues on next line. Syntax elements other than complete statements end with this symbol.
	Statement continues from previous line. Syntax elements other than complete statements begin with this symbol.
	Statement ends.
SELECT	Required item in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

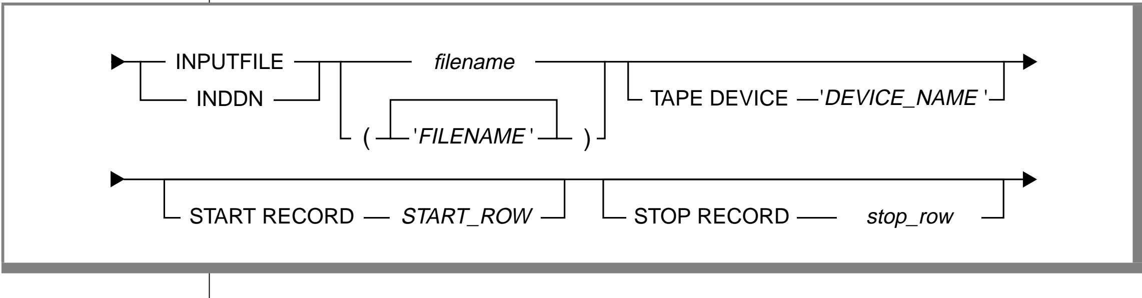
The preceding syntax elements are combined to form a diagram as follows.



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



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



Keywords and Punctuation

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

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

Identifiers and Names

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

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

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

Icon Conventions

Throughout the documentation, you will find text that is identified by several different types of icons. This section describes these icons.

Comment Icons

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

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

Platform Icons

Feature, product, and platform icons identify paragraphs that contain platform-specific information.

Icon	Description
UNIX	Identifies information that is specific to UNIX platforms

Icon	Description
Windows	Identifies information that is specific to Windows NT, Windows 95, and Windows 98 environments
WIN NT	Identifies information that is specific to the Windows NT environment
WIN 95/98	Identifies information that is specific to Windows 95 and Windows 98 environments

These icons can apply to an entire section or to one or more paragraphs within a section. If an icon appears next to a section heading, the information that applies to the indicated feature, product, or platform ends at the next heading at the same or higher level. A ♦ symbol indicates the end of feature-, product-, or platform-specific information that appears within one or more paragraphs within a section.

Customer Support

Please review the following information before contacting Informix Customer Support.

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

Telephone 1-800-274-8184 or 1-913-492-2086
(7 A.M. to 7 P.M. CST, Monday through Friday)

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

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

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

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

New Cases

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

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

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

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

Existing Cases

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

Troubleshooting Tips

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

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

Related Documentation

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

Document	Description
Administrator's Guide	Describes warehouse architecture, supported schemas, and other concepts relevant to databases. Procedural information for designing and implementing a database, maintaining a database, and tuning a database for performance. Includes a description of the system tables and the configuration file.
Installation and Configuration Guide	Provides installation and configuration information, as well as platform-specific material, about Red Brick Decision Server and related products. Customized for either UNIX or Windows NT.
Messages and Codes Reference Guide	Contains a complete listing of all informational, warning, and error messages generated by Informix Red Brick Decision Server products, including probable causes and recommended responses. Also includes event log messages that are written to the log files.

(1 of 2)

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

(2 of 2)

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

Document	Description
Client Connector Pack Installation Guide	Includes procedures for installing and configuring the Informix Red Brick ODBC Driver, the RISQL Entry Tool, and the RISQL Reporter on client systems. Included for sites that purchase the Client Connector Pack.
SQL-Back Track User's Guide	Is a complete guide to SQL-BackTrack, a command-line interface for backing up and recovering warehouse databases. Includes procedures for defining backup configuration files, performing online and checkpoint backups, and recovering the database to a consistent state.
Informix Vista User's Guide	Describes the Informix Vista aggregate navigation and advisory system. Illustrates how Vista improves the performance of queries by automatically rewriting queries using aggregates, describes how the Advisor recommends the best set of aggregates based on data collected daily, and shows how the system operates in a versioned environment.
JDBC Connectivity Guide	Includes information about Informix Red Brick JDBC Driver and the JDBC API, which allow Java programs to access database management systems.
ODBC Connectivity Guide	Includes information about ODBC conformance levels and instructions for using the Informix Red Brick ODBClib SDK to compile and link an ODBC application.

Additional references you might find helpful include:

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

Additional Documentation

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

Online Manuals

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

Printed Manuals

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

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

Informix Welcomes Your Comments

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

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

Write to us at the following address:

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

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

doc@informix.com

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

We appreciate your suggestions.

Chapter

1

Overview of Red Brick Decision Server

In This Chapter 1-3

Relational Database Tables 1-4
- Multiple Users and Table Locks 1-5
- Views of Tables 1-6
- Indexes 1-7
- Primary and Foreign Keys 1-7
- Database Integrity 1-9
- Authorization and Privileges 1-9
- The System Catalog 1-10

Red Brick Decision Server 1-10
- Schema Design 1-11
- Specialized Decision-Support Functions 1-11
- Comparisons With Subqueries 1-12
- Macros 1-12
- Segmented Storage 1-13
- Localization 1-13
- Aroma Database 1-14

In This Chapter

Informix Red Brick Decision Server is a relational database management system (RDBMS) designed for data warehouse, data mart, and online analytical processing (OLAP) applications. Compared to online transaction processing (OLTP) or "universal" database products, Red Brick Decision Server delivers higher query-processing and data-loading performance, greater ease of administration, and a richer set of specialized features for applications that range from a few gigabytes to well over a terabyte, and from a few users to thousands of users.

Red Brick Decision Server can scale from the workgroup to the enterprise, is built for an open client/server environment using industry-standard open database connectivity (ODBC), and is accessed using industry-standard SQL. The server's RISQL® extensions simplify analyses that require ranks, ratios, and other commonly used business calculations, while the Informix Vista, STARjoin™, STARindex™, TARGETjoin™, and TARGETindex™ technologies provide unparalleled ad hoc query and analysis performance against very large databases with various schema designs. Managers and analysts can pose numerous and creative queries to quickly receive the information they need, and make good business decisions with similar speed and confidence.

This chapter is an overview of the Red Brick Decision Server RDBMS. This overview contains the following sections:

- Relational Database Tables
  - Multiple Users and Table Locks
  - Views of Tables
  - Indexes
  - Primary and Foreign Keys
  - Database Integrity
  - Authorization and Privileges
  - The System Catalog
- Red Brick Decision Server
  - Schema Design
  - Specialized Decision-Support Functions
  - Comparisons With Subqueries
  - Macros
  - Segmented Storage
  - Localization
  - Aroma Database

Relational Database Tables

A relational database is a set of tables that have the following characteristics:

- A table is made up of columns and rows.
- A column is a set of values of the same datatype; a character column, for example, contains character strings and an integer column contains integers.
- A row is a sequence of values such that the *k*th value of the row corresponds to the *k*th column of the table.
- Each row is typically identified by a unique value known as its primary key. (It is possible, though not generally useful, to create a table without a primary key column.)
- A base table is a table created with a CREATE TABLE statement. A base table persists in the database until it is removed with a DROP TABLE statement.

- A result table is returned by a SELECT statement.
- A temporary table is a table that is accessible only during the session in which it is created. A temporary table persists in the database only for the duration of that session or until it is removed with a DROP TABLE statement.

Decision support database designers tend to speak of database tables as either *fact tables* or *dimension tables*. A fact table contains numeric data such as sales figures, numbers of units, percentages, and various other additive measurements, whereas a dimension table contains more descriptive data such as the names of cities, products, trademarks, brands, and time periods.

However, this difference in the contents of fact and dimension tables is an issue of schema design, not of table creation and query processing. When you create a table with Red Brick Decision Server software, the syntax presumes a single table type, and there are no restrictions on the behavior of fact table data versus dimension table data.

For a detailed discussion of schema design, refer to the *Administrator's Guide*.

Multiple Users and Table Locks

In a multi-user system, several users can access a given table simultaneously. If they were also allowed to update the table simultaneously, the table could be corrupted. To prevent corruption and to preserve data integrity, the server automatically controls access to any table that is being updated.

The server controls access by "locking" the table. After the table has been updated, the server automatically unlocks the table; consequently, any operation that modifies a table can make other users wait and thus lengthen expected retrieval times.

A user with appropriate authorization can also explicitly lock a table to control further access.

Views of Tables

A *view* is a virtual table defined on one or more base tables with a CREATE VIEW statement. Views, like tables, are made up of columns and rows; can be referenced in FROM clauses; given correlation names; and assigned specific access privileges. However, they cannot exist independently of base tables.

Views are commonly used to:

- Restrict access to part of a table. For example, one view could be defined that restricts sales figures to the eastern division and another that restricts sales figures to the western division.
- Simplify data. For example, a query that returns sales figures for a product in New York during the first two quarters of 1992 could reference several tables and require a complex query. A view could be defined to represent this data in a single table.
- Provide a different perspective on the data. For example, columns for specific districts, regions, and various calculations can be redefined in a view.

A precomputed view is a special type of view that is linked to a physical database table known as a precomputed table. The view defines a query, and the table contains the precomputed results of the query. Precomputed views are used to optimize query performance by rewriting aggregate queries; for detailed information about query rewriting, refer to the ***Informix Vista User's Guide***.

Indexes

When you use the CREATE TABLE statement to create a new base table, the server automatically creates a B-TREE index on the table's primary key columns.

To ensure fast access to all the table data, you can define additional indexes on a column or group of columns with a CREATE INDEX statement. The following types of indexes are supported:

- B-TREE—the default.
- STAR—for accelerating table joins. You can define a STAR index on any base table that has foreign keys defined.
- TARGET—bit-vector indexes for processing queries with weakly selective constraints. You can create TARGET indexes on single, non-unique columns. TARGET indexes can also be used to enable TARGETjoin processing.

Primary and Foreign Keys

Database tables typically have a primary key that uniquely identifies each row. A primary key can be one value from a single column or a combination of values from multiple columns.

Any base table in a Red Brick Decision Server database can have a multi-column primary key. When a table is created, the column (or columns) declared as the primary key must satisfy the following conditions:

- It must contain a value for each row of the table; that is, it must be declared NOT NULL.
- Its values (or combination of values) must be unique for each row.

Automobile identification numbers, employee ID numbers, and social security numbers are examples of primary keys that consist of meaningful values. In many cases, however, the primary key consists of a value that is not meaningful; it may be simply a unique series of numbers or characters.

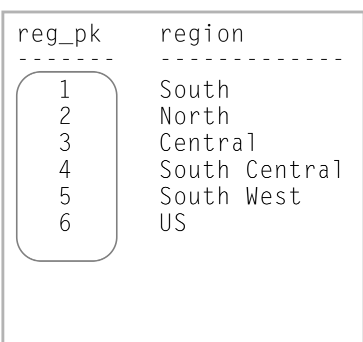
A foreign-key column contains only the values of a primary-key column of another table. The values in a foreign key column establish a relationship between two tables because they refer to one or more rows of another table.

Unlike a primary-key column, a foreign-key column can contain duplicate values, as shown in the following example.

Example

The table on the right contains foreign key references to the table on the left.

Figure 1-1
Foreign-key References



reg_pk	region
1	South
2	North
3	Central
4	South Central
5	South West
6	US

state_pk	state	foreign_k
1	Florida	1
2	Georgia	1
3	Alabama	1
4	Mississippi	1
5	Louisiana	1
6	Chicago	3
7	Illinois	3
8	Minnesota	3
0	Total US	6

The Informix Red Brick Decision Server server is designed to manage databases with various schema designs. Therefore, the relationship between the tables is flexible; any table can be foreign-key referenced by any other table, and any table can have a multi-column primary key. The server can navigate multiple join paths between tables as long as the intended join is explicitly stated in each query and the joining columns have comparable datatypes.

Database Integrity

A relational database should conform to both of the following integrity rules:

- Entity integrity: The primary key columns for each table must contain a unique value.
- Referential integrity: Each value in a foreign key column must exist as a primary key of the table it references.

When the Table Management Utility (TMU) loads a database, it enforces both of these rules. These rules are further enforced by cascaded locking and delete operations as databases are modified by INSERT, UPDATE, and DELETE statements.

With the Parallel TMU, referential integrity enforcement can be done simultaneously with the data loading to improve overall load performance.

For information about loading databases, refer to the *Table Management Utility Reference Guide.*

Authorization and Privileges

Before users can execute Structured Query Language (SQL) statements, they must have the proper authority. Authority is controlled by system roles, task authorizations, and object privileges:

- A database user is assigned a specific system role (CONNECT, RESOURCE, DBA). Members of a system role can perform the tasks defined for that role. Database users can be assigned individual task authorizations.
- A table is protected by object privileges that can be granted or denied to selected database users (SELECT, INSERT, UPDATE, DELETE).

These system roles and object privileges are set with GRANT and REVOKE statements.

The System Catalog

The database server maintains a set of system tables referred to as the *system catalog*. These tables define all the data that resides in the database and all the control information required to manage and protect the database. The system tables can be queried with standard SELECT statements by users who have the appropriate privileges. For details, refer to the *Informix Red Brick Decision Server Administrator's Guide*.

Red Brick Decision Server

Red Brick Decision Server is a client/server decision support RDBMS for information systems (IS) and business managers who want to improve the quality and performance of their decision-support applications. The superior performance of Red Brick Decision Server is based on:

- Indexing and joining technologies designed to accelerate the retrieval of database information.
- Specialized decision-support functions that can perform sequential calculations efficiently.
- Scalar subqueries that can compare multiple values efficiently.
- Macro commands that generalize and simplify complex queries.
- Segmented storage that accommodates large tables and increases availability of data.
- Database structures that are easy to understand, maintain, and access.
- Query rewriting technology designed to accelerate the performance of aggregate queries.

Schema Design

A schema defines the structure of a database: the tables, their columns and primary keys, and all foreign key relationships. A schema can usually be designed and optimized for update operations but only at the expense of retrieval. Conversely, a schema can be optimized for retrieval but only at the expense of updates. Data warehousing schemas are designed and optimized for retrieval.

For example, star schemas are designed for dimensional analysis of data by end users. In a star schema, tables tend to contain either dimensional data—descriptive data that reflects the various dimensions of a business—or facts—mostly numeric data that tracks the business. Rapid retrieval of data is accomplished in part by indexes built on primary and foreign keys.

Red Brick Decision Server databases are flexible enough to accommodate many variations on the star schema, and to allow queries against *arbitrary schemas* that do not adhere to the relationships implicit in a star. Although the star schema methodology tends to optimize query-processing performance, there are no limits on the type of schema you can design.

Specialized Decision-Support Functions

The RISQL extensions to SQL contain functions that simplify the expression of questions commonly asked of decision-support systems. RISQL display functions can calculate:

- Running totals
- Moving averages
- Moving sums
- Ranks
- Ranks by arbitrary tiling, such as thirds or tenths
- Ratios

These functions return values for an entire result table or for groups of rows in a result table. They can be used inside expressions (which may themselves contain other display functions) and as arguments to scalar functions.

Comparisons With Subqueries

To simplify the writing of queries that compare data, you can include subqueries or CASE expressions in a query's select list or subqueries in the FROM clause. For example, you can compare:

- Sales figures this month versus the last six months
- Sales figures this year versus last year, and the year before last
- Sales figures in the West versus the East, South, and Midwest

and display the results in a readable spreadsheet format. With a version of SQL that does not support this kind of expressibility, such routine comparisons are extremely difficult to make (and their results are much more difficult to read).

Macros

A CREATE MACRO command is available for creating macro names that abbreviate a specified character string. When the server finds a macro name in an SQL statement, it substitutes the character string for the macro name.

A macro name can also be created for a set of character strings—which is specified as a character string that contains parameters. For example, the macro name

```
sales(mon, day, yr)
```

could be created to abbreviate a complex condition. When the macro is used within a query, it is written with a specific month, day, and year:

```
sales(12, 25, 2000)
```

The server substitutes 12 for the month, 25 for the day, and 2000 for the year when it replaces the string with the macro definition. This kind of parameterized macro hides complexity from users and allows for the construction of highly generalized, reusable queries.

Segmented Storage

Database tables and indexes can reside in a default segment or in one or more segments explicitly defined by the database administrator. A segment is a set of files defined with a CREATE SEGMENT statement. After a segment has been created, it can be used to store the data from one table or one of the table's indexes.

Segments simplify the administration of large databases and can be allocated to improve database performance and to support tables too large to fit in a single file. A single table or index can exist in multiple segments, allowing the administrator to control the table or index at the segment level.

Localization

Red Brick Decision Server provides full product functionality independent of locale. After specifying a locale for the server, users can:

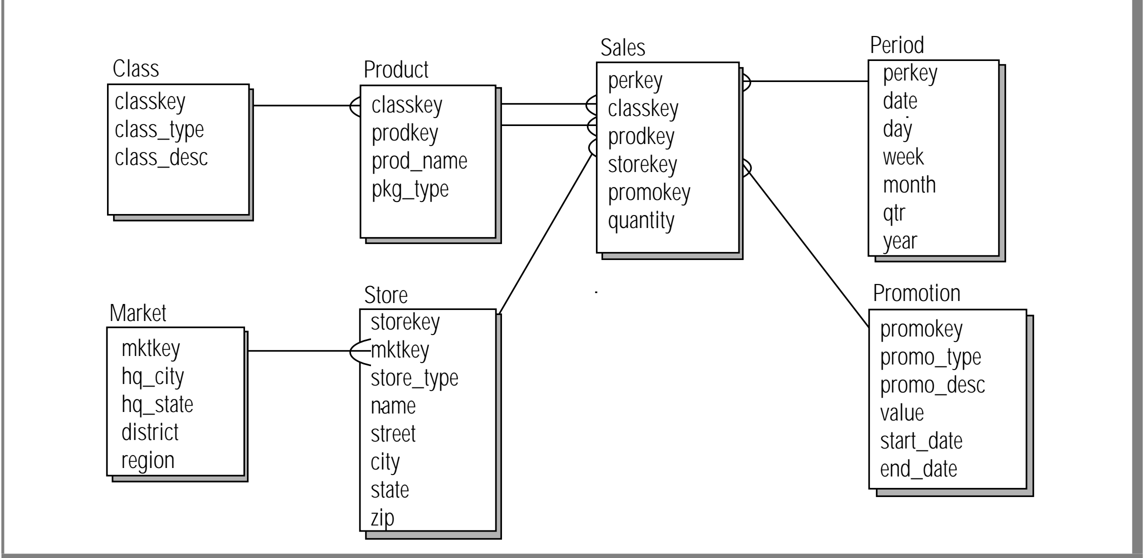
- Load, store, index, and query single-byte or multibyte character data.
- Receive error, warning, and information messages in any supported language.
- Give database objects such as tables, columns, indexes, segments, roles, and users language-specific names, using single-byte or multibyte characters.
- Use SQL constraints and RISQL functions that sort according to a localized collation sequence and return results that are correct for the designated locale.
- Load numeric, date, and time data using localized formats.
- Apply date, time, and string scalar functions to localized data.

For information about defining the server locale, refer to the *Installation and Configuration Guide*.

Aroma Database

Most of the examples in this guide are based on data from the basic Aroma database, which tracks daily retail sales data in stores owned by the Aroma Coffee and Tea Company. This database has four dimension tables—Period, Product, Store, and Promotion—and a Sales fact table, as well as two outboard tables, Class and Market. Figure 1-2 illustrates this basic schema.

Figure 1-2
Aroma Database Schema



A few of the examples are based on an additional purchasing schema used to track orders that the Aroma Company receives from its suppliers. For details about the complete Aroma database, refer to the *SQL Self-Study Guide*.

Chapter

2

Elements of the Language

In This Chapter 2-3

Names and Identifiers 2-3
Standard Identifiers 2-4
Delimited Identifiers 2-5
Uppercase and Lowercase 2-6
Aliases and Correlation Names 2-7
Column Aliases 2-7
Table Correlation Names 2-7
Qualified Column Names 2-8

Literals 2-10
Character Literals 2-11
Datetime Literals 2-12
Integer Constant 2-14
Decimal Constant 2-15
Floating-Point Constant 2-16

Datatypes 2-18
CHARACTER 2-19
DATETIME Datatypes: DATE, TIME, and TIMESTAMP 2-21
INTEGER 2-23
SMALLINT 2-23
TINYINT 2-23
DECIMAL and NUMERIC 2-24
REAL 2-25
DOUBLE PRECISION and FLOAT 2-25

Missing Values and NULLs 2-26

Assignment and Comparison 2-27
Assignment 2-27
Character String 2-28
Datetime Values 2-28
Numeric Values 2-29
Comparison 2-29
Character Strings 2-29
Datetime Values 2-29
Numeric Values 2-31

In This Chapter

SQL statements are constructed from a well-defined set of language elements and in accordance with a basic set of rules. This chapter describes these elements and rules in the following sections:

- Names and Identifiers
- Literals
- Datatypes
- Missing Values and NULLs
- Assignment and Comparison

Names and Identifiers

A name identifies a database object, a database user, or a password. The words *name* and *identifier* are synonymous.

Names of database objects can be specified using single- or multibyte characters. This flexibility allows users to create meaningful names in their native languages for the following objects:

- Tables, table columns, views, and synonyms
- Indexes
- Segments
- Macros
- Roles
- Database usernames

With the exception of database usernames, names used to communicate outside the database *cannot* contain multibyte characters. Therefore, passwords and filenames are restricted to single-byte characters, regardless of the server locale.

If the single-byte characters are a subset of the character set specified in the database locale, object names can contain both single-byte and multibyte characters.

Standard Identifiers

Each name that occurs in an SQL statement must:

- Contain only letters (*A–Z* and *a–z* in English), digits (0–9), or single-byte ASCII underscore characters (_). Multibyte underscore characters return a syntax error.
- Begin with an alphabetic character.
- Have a minimum length of one byte and a maximum of 128 bytes.
- Contain no quotation marks and no spaces.
- Not be a reserved word (as listed in Appendix B, “Reserved Words”).

Refer to “Delimited Identifiers” on page 2-5 for information about using identifiers that do not conform to these rules.

The following are valid database identifiers:

Identifier	Database Object
product	Name of a table
dollars	Name of a column
top_25_brands	Name of a column

Delimited Identifiers

Red Brick Decision Server supports double quotation marks (") around a string of characters to define delimited identifiers, consistent with the ANSI SQL-92 standard. This allows you to create an identifier that contains any string. The double quotation marks are necessary to define identifiers that do not follow the rules outlined in "Standard Identifiers" on page 2-4. If you want to create an identifier that contains the double quotation-mark character ("), you must escape the double quotation-mark character with another double quotation-mark character.

If you have any existing applications that use double quotation marks (") to mark the beginning and the end of a literal, the double quotation marks in that application must be replaced with single quotation marks (').

Examples

The following table contains examples that present the syntax for delimited identifiers, the resulting output of the identifier, and a brief explanation of why the double quotation marks are needed:

SQL Syntax	Result	Explanation
`"table"`	`table`	TABLE is a reserved word, so an identifier with that name must be surrounded by double quotation marks.
`"""SELECT"""`	`"SELECT"`	SELECT is a reserved word, so an identifier with that name must be surrounded by double quotation marks. To include the double quotation-mark characters in the output, each double quotation-mark character must be escaped by double quotation marks.
`"Column Name"`	`Column Name`	The double quotation marks are required because the identifier contains a space and upper- and lowercase letters.
`"The ""STAR"""`	`The "STAR"`	The first quotation mark starts the identifier, the second quotation mark escapes the third quotation mark, the fourth quotation mark escapes the fifth quotation mark, and the sixth quotation mark ends the identifier.

To create a table named *table* with a column named *The "STAR"*, enter the following SQL command:

```
create table "table" (
          "The ""STAR""" int) ;
```

To select from this table, enter the following:

```
select "The ""STAR"""
from "table" ;
```

Uppercase and Lowercase

For storage in the system tables, the Red Brick Decision Server converts the lowercase letters of a name into uppercase letters. For example, the name *MaRkeT* is converted to MARKET and *fred* to FRED. Consequently, the name *Market* is equivalent to MARKET, and the condition

```
MarKet = MARKET
```

is true unless the value of MARKET is missing or unknown (NULL). When the value of a name is missing, the statement evaluates to "unknown."

Characters that do not have uppercase variants, such as *kanji* ideographs, are stored as entered.

Example

The following query returns a list of all table names created by *fred*. Because the server converts *fred* to uppercase letters, FRED must be used in the search condition.

```
select name
from rbw_tables
where creator = 'FRED'
```

Aliases and Correlation Names

Columns and tables are assigned names when they are created, but these names can be substituted with column aliases and table correlation names for the duration of a query. These temporary names exist only during the execution of a specific statement.

Column Aliases

An alias for a column is defined in the select list with the optional AS keyword, and changes the column's name for the duration of the query, or provides a name if none is already specified. When a column is assigned an alias, subsequent clauses of the query can refer to the column by this name.

For example, the query

```
select prod_name as brand, sum(dollars) as sales
from product natural join sales
where brand = 'Aroma Roma'
group by brand;
```

gives the names *brand* and *sales* to the columns in the select list, and *brand* is used in both the WHERE clause and the GROUP BY clause.

Table Correlation Names

A correlation name for a table is defined immediately after the table reference in the FROM clause (the AS keyword is optional). Correlation names eliminate ambiguity whenever a query and a subquery reference the same table in a correlation condition or when a table is being joined to itself. They also provide a means of naming tables that derive from subqueries and other query expressions.

The behavior of an alias for a table differs from the behavior of a synonym. For information about synonyms, see "CREATE SYNONYM" on page 8-131.

Derived Column Lists

When a correlation name is used for a table reference in the FROM clause, it can be followed by a list of column names. In the case of a simple table reference, the column list temporarily changes the names of the columns in the table. In the case of a query expression, the column list is a mechanism for supplying unique names for the columns in the derived table. In both cases, the column list is in effect for the duration of the query.

The column list, if specified, must include a name for each column in the table. When column lists are used, care must be taken in the specification of natural joins and named-columns joins (joins that include the USING subclause) because the joining columns will be based on the names specified in the column list.

Also note that a natural join or named-columns join derives to a table that contains *one* column for each pair of joined columns, whereas a join specified with the ON syntax derives to a table in which *both* joining columns are present.

The syntax for column aliases is discussed on page 7-19; for correlation names and column lists, on page 7-13; and for different types of joins, on page 7-6.

Qualified Column Names

A qualified column name is simply a column name that is qualified by the name of the table, view, or synonym to which it belongs. The name of the table is either the name defined in the CREATE statement or a correlation name specified in the FROM clause.

The column name and the table reference must be separated by a period. For example:

```
sales.dollars
s.dollars
period.perkey
t1.perkey
view_sales99.quantity
v99.quantity
```

For a full description of table references, refer to page 7-13.

Examples

The following examples illustrate the use of column aliases and correlation names. (Both examples are described in more detail in Chapters 4 and 5, respectively, of the *SQL Self-Study Guide*.)

In the first example, correlation names are assigned in the FROM clause of both the subquery and the main query. Qualified column names distinguish the column references in both the subquery's correlation conditions and the main query's select list and GROUP BY clause.

```
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 year = 1999
    and 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;
```

The column references in the WHERE clause of the main query are not ambiguous and do not need to be qualified; they reference only the tables listed in the FROM clause of the main query.

The second example uses a correlation name and column list to name the table and columns derived from an outer join of two subqueries in the FROM clause. Also, a column alias (*wk_no*) is used in the select list to identify the column resulting from the expression

```
extract(week from date)
```

This alias is subsequently referred to in the ORDER BY and BREAK BY clauses.

```
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 3, 4;
```

The outer join is neither a named-columns join nor a a natural join, so the derived table, *t3*, has four columns, not three.

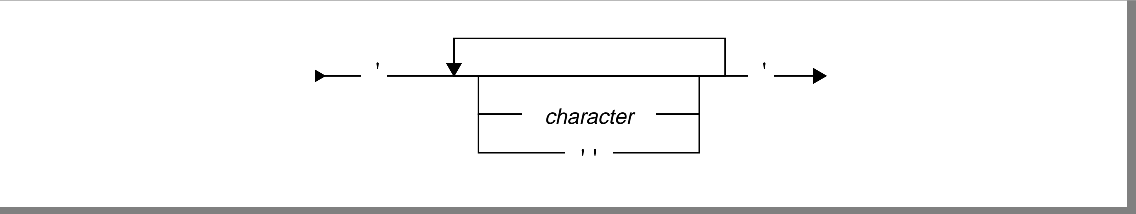
Literals

A literal is a fixed sequence of characters or a numeric constant. Character literals are often referred to as character constants, character strings, or strings. Datetime literals, a special type of character literal, are a fixed sequence of characters that represent a date, a time, or both.

Numeric constants can be integer, decimal, or floating-point, and they have a sign, a precision, and a scale. The precision of a numeric constant is the total number of digits in the constant, and the scale is the number of digits to the right of the decimal point.

Character Literals

A single quote (') marks the beginning and the end of a character literal. The length of a character literal is the number of characters in the literal. The following syntax diagram shows how to construct a character literal.



The character set used by the server is defined during installation as part of the locale specification. For more information about locales, refer to the *Informix Red Brick Decision Server Administrator's Guide*.

Usage Notes

- A character can be alphabetic, numeric, or any special character.
- The length of a character literal must be within the range of `0` and `1024`.
- The character string of length zero, two single quotes (''), is called the "empty string." It is not the same as NULL.
- To represent a single quote mark (') in a string, use two single quote marks ('').
- Character literals are sensitive to case: *Chicago* is not the same as *chicago*.

Examples

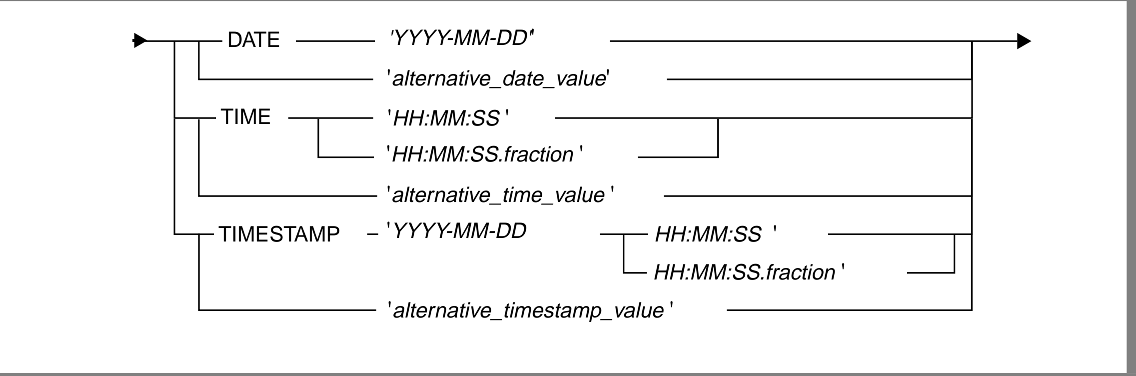
The following character strings are valid character literals:

```
'Informix Software'
'Scarlet O''hara'
```

Datetime Literals

The keywords DATE, TIME, and TIMESTAMP indicate that the literal that follows complies with the ANSI SQL-92 datetime datatype.

The following syntax diagram shows how to construct datetime literals:



Because certain query tools generate SQL that does not comply with ANSI-92 datetime literal definitions, *Red Brick Decision Server* provides limited support for some SQL server datetime formats. These formats can be used only when the server locale specifies the language as English and the territory as United States; in other locales, ANSI datetime literals must be used. For detailed information about alternative datetime formats, refer to Appendix C, “Alternative Datetime Formats.”

Usage Notes

- A single quote (') denotes the beginning and end of a datetime literal.
- All date and time elements are unsigned integers with the following ranges:

Element	Interpretation	Start	Stop
YYYY	Year	0001	9999
MM	Month	01	12
DD	Day of the month	01	31
HH	Hour	00	23
MM	Minutes	00	59
SS	Seconds	00	59
fraction	Fraction of a second	0	999999

- Date elements are separated by a dash (-): YYYY-MM-DD. No spaces are allowed between the date elements.
- Time elements are separated by a colon (:) or a period (.):

  HH:MM:SS
  HH:MM:SS.fraction

  No spaces are allowed between the time elements.
- *fraction* represents 0–6 digits that represent fractional seconds.
- Spaces are allowed but not required between the date component and the time component of a timestamp.

Examples

The following examples are valid datetime literals:

```
DATE'1993-12-25'
TIME'08:23:16'
TIME'14:23:16.5'
TIMESTAMP'1993-12-25 08:23:16'
```

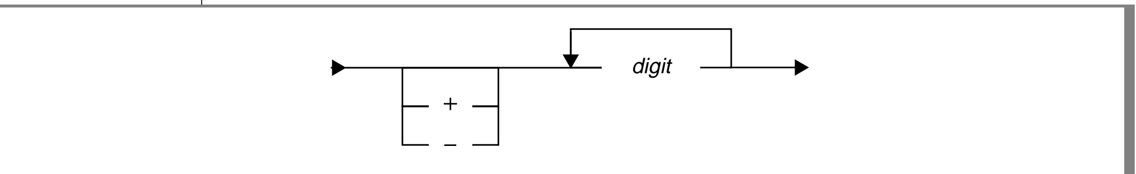
The following example constrains a DATE datatype column, using an ANSI SQL-92 datetime literal:

```
select * from period where date_col = DATE '1999-12-25'
```

To use alternative datetime literals, you might have to set the DATEFORMAT variable to the format you want to use, as described in Appendix C, "Alternative Datetime Formats."

Integer Constant

An integer constant is a sequence of digits (0–9); a positive (+) or negative sign (–) can precede the digits. The following syntax diagram shows how to construct an integer constant:



Usage Notes

- An integer constant must contain no more than nine digits; otherwise, it is interpreted as a floating-point constant.
- The precision of an integer constant is its length in digits. It has an implied scale of zero.
- An integer constant cannot contain commas. For example, 1,000 cannot be used.
- Only single-byte encodings for digits are accepted as legal numeric characters.

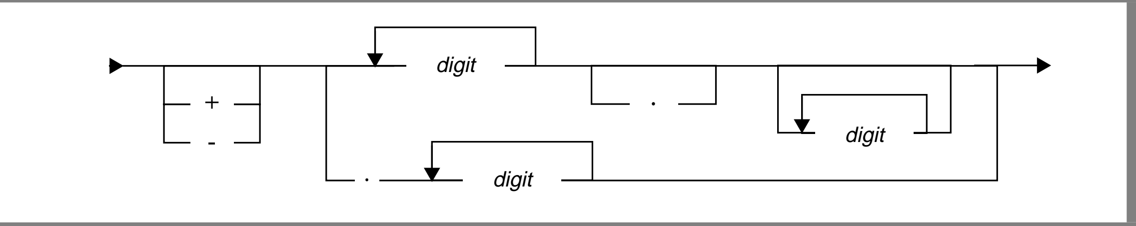
Examples

The following values are valid integer constants:

```
234
-5280333
+2274
+999990001
```

Decimal Constant

A decimal constant is a sequence of digits that can include a decimal point. A positive (+) or negative sign (-) can precede a decimal constant. The following syntax diagram shows how to construct a decimal constant:



Usage Notes

- A decimal constant must contain no more than 38 digits excluding the sign and decimal point, or it is interpreted as a floating-point constant.
- The precision of a decimal constant is the length in digits. The scale is the number of digits following the decimal point.
- A decimal constant is not required to have a decimal point. If no decimal point is present, the scale is interpreted as 0.
- A decimal constant cannot contain commas (for example, 1,100.10).
- The decimal point character must be the period (.), regardless of the server locale.
- Only single-byte encodings for digits are accepted as legal numeric characters.

Examples

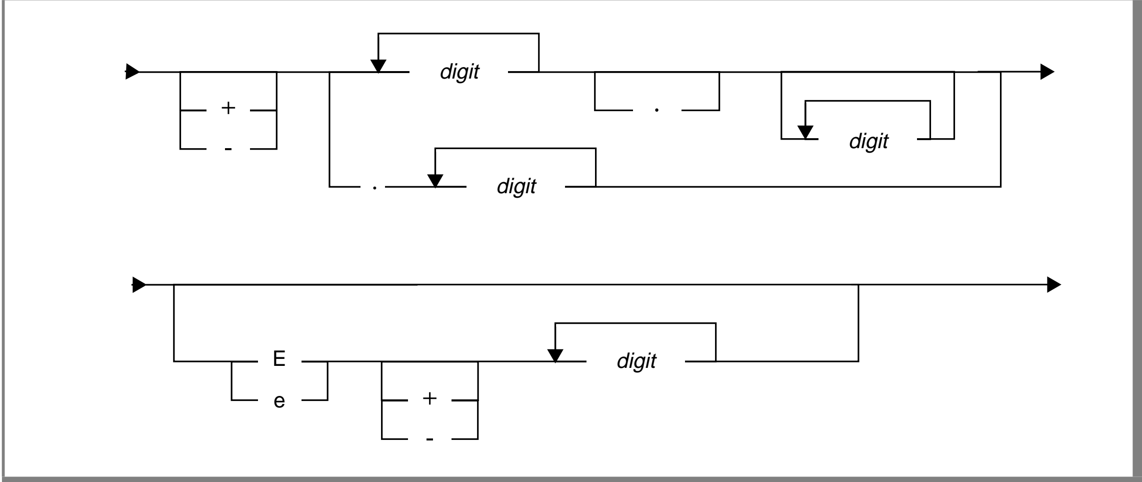
The following numbers are valid decimal constants:

`234`	precision 3, scale 0
`+234.78`	precision 5, scale 2
`+.007`	precision 3, scale 3
`-2.1414`	precision 5, scale 4
`1234567890.123456789`	precision 19, scale 9

Floating-Point Constant

A floating-point constant is a string of more than 38 digits or a string of digits in “exponential” notation. A value is expressed in exponential notation as a decimal number and an exponent separated by a single character (*E* or *e*). The value of the expression is equal to the decimal number multiplied by the power of ten specified by the exponent. This notation is also called “scientific” notation.

The upper half of the following syntax diagram shows how to construct the decimal part of a floating-point number; the lower half shows how to construct the exponential part:



Examples

The following numbers are valid floating-point constants:

```
1.73e+5
-2.93E+9
145.06e-5
.003E+4
+1234.56789e105
```

Usage Notes

- The decimal point character in a floating-point constant must be the period (.), regardless of the server locale.
- Only single-byte encodings for digits are accepted as legal numeric characters.

Datatypes

Each value stored, retrieved, deleted, inserted, or updated by the Red Brick Decision Server has a datatype that characterizes its relevant properties. Datatypes are declared when a table is created with the CREATE TABLE command. A value's datatype can also be changed during a routine computation; for example, when an integer is added to a decimal number, the integer is converted to a decimal.

The Informix Red Brick Structured Query Language (SQL) supports all the ANSI SQL-89 standard datatypes and the tiny integer datatype; it also supports a subset of ANSI SQL-92 date datatypes.

The following table lists the datatypes supported. It also shows the datatype mapping performed by the Informix Red Brick ODBC Driver between server datatypes, as declared in the CREATE TABLE statement, and ODBC datatypes:

Server Datatype	ODBC SQL Datatype	Default ODBC C Datatype
CHAR	SQL_CHAR	SQL_C_CHAR
VARCHAR	SQL_CHAR	SQL_C_CHAR
TINYINT	SQL_TINYINT	SQL_C_STINYINT
SMALLINT	SQL_SMALLINT	SQL_C_SSHORT
INTEGER	SQL_INTEGER	SQL_C_SLONG
NUMERIC, DECIMAL	SQL_DECIMAL	SQL_C_CHAR
REAL	SQL_REAL	SQL_C_FLOAT
DOUBLE, FLOAT	SQL_DOUBLE	SQL_C_DOUBLE
DATE	SQL_DATE	SQL_C_DATE
TIME	SQL_TIME	SQL_C_TIME
TIMESTAMP	SQL_TIMESTAMP	SQL_C_TIMESTAMP

The third column shows the ODBC datatypes to which the Informix Red Brick datatype is logically mapped.

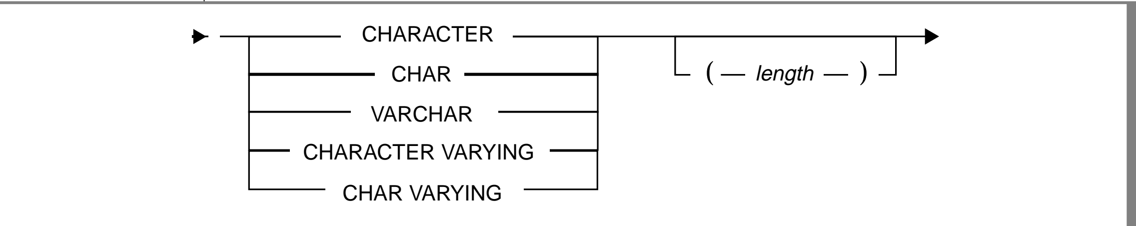
For more information about creating end-user ODBC applications for use with Red Brick Decision Server, refer to the *ODBC Connectivity Guide*.

The maximum and minimum values of floating-point datatypes depend on the host platform.

CHARACTER

You can store character strings as fixed-length or variable-length values. The CHAR data type stores character strings as fixed-length values and VARCHAR stores them as variable-length values.

When a 7- or 8-bit ASCII character set is used, a character is always one byte long, so it makes no difference whether the length argument is interpreted in bytes or characters. However, users of multibyte character sets must account for the possibility that a character column in a database table might not be able to hold as many characters as it would if a single-byte character set were being used.



Usage Notes

- The keyword CHARACTER is synonymous with CHAR.
- The keywords CHARACTER VARYING and CHAR VARYING are synonymous with VARCHAR.
- The value of *length* specifies the maximum length of the column and has a default value of 1. The value of *length* must be less than 1025.
- A character string stored in a CHAR column is left-justified and padded with trailing blanks to the length of the column. All the strings stored in a CHAR column have the same length.

- A character string stored in a VARCHAR column has exactly the same length as the source string or the expression that generated the string (including trailing blanks). Character strings stored in a VARCHAR column can vary in length.
- A character string stored in a VARCHAR column incurs a 2-byte overhead. Do not use this datatype for columns less than 6 bytes long or for columns that store strings of the same length. Use the CHAR datatype instead.

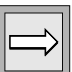
Usage of the VARCHAR versus the CHAR datatype

The VARCHAR datatype is used principally to save storage space in those cases where the length of the source character strings vary widely. When you store a five-character string in a CHAR(20) column, the length of the stored string increases to twenty-characters (fifteen trailing blanks). When you store that same string in a VARCHAR(20) column, however, its length is only five characters.

Use CHAR instead of VARCHAR when:

- The length of the column is less than 6 bytes.
- The length of the source strings *does not* vary widely.
- The column is subject to frequent updates that lengthen the stored row. In these cases, the row might not be stored optimally and subsequent access to the data can be slow.

The presence of trailing blanks in source data can lead to unexpected or indeterminate results if that data is stored in a VARCHAR column.



***Important:** The presence of trailing blanks within a VARCHAR column can lead to unintended or indeterminate results. Informix recommends that you do not store data with trailing blanks in VARCHAR columns.*

Trailing blanks lose significance when stored in a CHAR column. The source strings ‘zebra’ and ‘zebra ‘ (one trailing space) both have length of 20 when stored in a CHAR(20) column but have lengths 5 and 6, respectively, when stored in a VARCHAR(20) column. Trailing blanks within VARCHAR columns are thus significant for scalar functions such as LENGTH, CONCAT, and SUBSTR.

Trailing blanks within VARCHAR columns also have significance in search conditions that use comparison predicates such as LIKE. For example, the following search condition:

```
animal LIKE 'zebra %'
```

is true when the string is 'zebra ' but false when it is 'zebra'.

Trailing blanks within a VARCHAR column can also lead to indeterminate results. For example, the following query can return a 5 or 6:

```
select LENGTH(MAX(animal))
from kingdom_table;
```

The value returned depends on which value the query chooses during a given execution: 'zebra' or 'zebra '. In the context of the MAX set function, the two character strings are equal and thus equally likely to be chosen as the maximum value. The actual value selected depends on the order of the values in the table and circumstances of the query execution such as the query plan chosen, actual degree of parallelism, and so on.

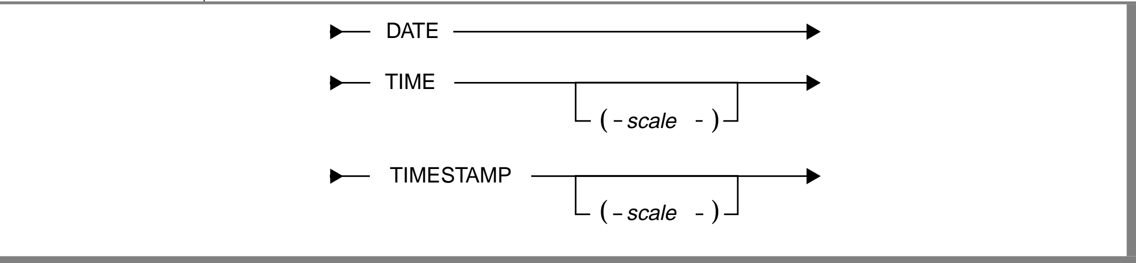
No inconsistency can occur if no trailing blanks occur in VARCHAR data. When trailing blanks are required, use them consistently. For example, if one trailing blank is required, make sure that every value has exactly one trailing blank. If your logic requires a minimum column width, pad values that have less than the minimum with trailing spaces so that they have that minimum width, and strip trailing blanks from all other values.

DATETIME Datatypes: DATE, TIME, and TIMESTAMP

The date, time, and timestamp datatypes (referred to collectively as datetime datatypes) store date and time information in a form that can be operated on by scalar, comparison, and set functions. The DATE, TIME, and TIMESTAMP datatypes comply with ANSI SQL-92 datetime datatype definitions with the following exceptions:

- The concept of time zones is not supported.
- The concept of interval is not directly supported, but is supported indirectly through scalar functions.

These datatypes are declared with the DATE, TIME, and TIMESTAMP keywords.



Usage Notes

- The scale value must be an integer between 0 and 6. This number specifies how many fractional-second digits are available for display and calculations. If *scale* is greater than 0, then 6 digits are stored internally, but only *scale* digits are significant—the remaining digits are zeros. If *scale* = 0, then no fractional-second digits are stored.

  If a scale value is not specified, the following default values for *scale* are used:

  TIME
  TIMESTAMP

- The ranges for the datetime datatypes are as follows:

DATE	January 1, 1 to December 31, 9999
TIME	0:0:0 to 23:59:59.999999
TIMESTAMP	January 1, 1 0:0:0.000000 to December 31, 9999 23:59:59.999999

- For comparison between values of different precisions, the smaller precision value is padded with zeros to the greater precision.
- Chronological order is used when datetime datatypes are sorted or compared.

Examples

The following examples declare datetime datatypes:

DATE	
TIME	No fractional seconds
TIME(6)	Fractional seconds component has 6 significant digits
TIMESTAMP(0)	No fractional seconds
TIMESTAMP	Fractional seconds component has 6 significant digits
TIMESTAMP(2)	Fractional seconds component has 2 significant digits

INTEGER

The integer datatype defines signed integer values that range between -2^{31} and $2^{31}-1$. (2^{31} = 2,147,483,648). This datatype is declared with the INTEGER or INT keywords.

SMALLINT

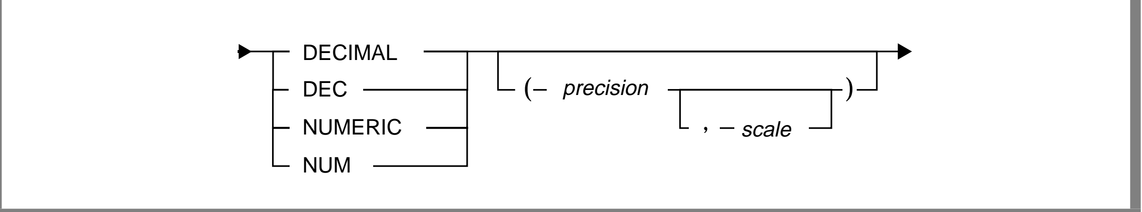
The small integer datatype defines signed integer values that range between -2^{15} and $2^{15}-1$. (2^{15} = 32,768). This datatype is declared with the SMALLINT keyword.

TINYINT

The tiny integer datatype defines signed integer values that range between -2^{7} and $2^{7}-1$. (2^{7} = 128). This datatype is declared with the TINYINT keyword.

DECIMAL and NUMERIC

The decimal and numeric datatypes are exact numeric datatypes that have a precision and a scale. Decimal and numeric datatypes are synonymous. This datatype can be declared with the DECIMAL, DEC, NUMERIC, or NUM keyword.



Usage Notes

- Maximum precision of a decimal datatype is 38 digits.
- Scale of a decimal datatype must be non-negative and less than or equal to its precision.
- Default is 9 for precision and 0 for scale.
- Decimal datatype is the same as the numeric datatype.

Examples

The following examples declare a decimal datatype:

`DECIMAL`	Precision 9, scale 0
`DEC`	Precision 9, scale 0
`DECIMAL(9)`	Precision 9, scale 0
`DECIMAL(38,38)`	Precision 38, scale 38
`NUMERIC`	Precision 9, scale 0
`NUMERIC(9)`	Precision 9, scale 0

REAL

The real datatype is an approximate numeric datatype that defines signed, floating-point numbers. This datatype can be declared with the keyword REAL.

Usage Note

Values of the real datatype are single-precision and range between approximately 1.E–38 and 1.E37 (minimum range; the maximum range depends on the host platform).

DOUBLE PRECISION and FLOAT

The double-precision and float datatypes are approximate numeric datatypes that define signed, floating-point numbers. This datatype can be declared with either the DOUBLE PRECISION or FLOAT keyword.

Usage Notes

- Values of this datatype are double-precision and range between 1.E–308 and 1.E307 (minimum range; the maximum range depends on the host platform).
- DOUBLE PRECISION and FLOAT datatypes are synonymous.

Missing Values and NULLs

The Red Brick Decision Server handles missing values as follows:

- A missing value is not the same as a space or blank. For example:

  ```
  (market is null)
  ```

  is false when the Market column contains spaces but true when entries are missing.

- A missing value is not the same as a zero. For example:

  ```
  (dollars = 0)
  ```

  is true when the Dollars column contains zero but neither true nor false when values are missing. When values are missing, the truth value of the statement is unknown.

- Conditions that reference missing values are evaluated with a three-valued logic (true, false, unknown). For example, the truth value of

  ```
  (dollars > 100)
  ```

  is unknown for each case where values are missing from the Dollars column.

- An arithmetic expression that references a missing value evaluates to NULL. For example:

  ```
  (dollars+5)
  ```

  returns NULL whenever a value is missing from the Dollars column.

- The IS NULL predicate is true and the IS NOT NULL predicate is false when a value is missing from a column. For example:

  ```
  (dollars is null)
  ```

  is true if and only if values are missing from the Dollars column.

- Most display and set functions ignore missing values. For example:

  ```
  rank(dollars)
  ```

  returns the rank of all existing values.

- Most scalar functions return NULL when an expression references a missing value. For example:

  ```
  (float(dollars))
  ```

  returns NULL when the value for the Dollars column is missing.
- The scalar function IFNULL detects missing values and replaces them with a specified value. For example:

  ```
  (ifnull(market, 'No Name'))
  ```

  returns the value *No Name* when values are missing from the Market column.

Assignment and Comparison

The INSERT and UPDATE commands assign values to columns, and various operators and functions compare values. When values are assigned or compared, they must be of compatible datatypes.

Numeric and character datatypes are not compatible. For example, a character string cannot be stored in a column that is declared numeric and cannot be compared with a numeric value.

Assignment

Only character strings can be stored in columns declared CHARACTER; only datetime values can be stored in columns declared as datetime datatypes; and only numeric values can be stored in columns declared as a numeric datatype. The following rules govern special cases.

Character String

When a character string is to be inserted into a column or is to update a column, the following rules apply:

- If the length of the character string is less than the declared length of a CHAR column, the character string is padded on the right with the necessary number of blanks to fill the column.
- If the length of the character string is less than the declared length of a VARCHAR column, the new length of the VARCHAR attribute is the length of the string that is being assigned.
- If the length of a character string is greater than the declared length of the column, the INSERT or UPDATE command fails and the server returns an error message.

Datetime Values

When a datetime literal is inserted into or updates a column, the following rules apply:

- A literal that has a DATE, TIME, or TIMESTAMP prefix is understood to have a datetime value.
- If a timestamp value is missing a time datepart, the time value for midnight is used.
- If a timestamp value is missing a date datepart, the date value for January 1, 1900 (1900-01-01) is used.
- A literal without a DATE, TIME, or TIMESTAMP prefix is interpreted as an alternative datetime literal rather than a character literal *only if the content expects a datetime value*; in this case the literal is converted to a datetime value.

  In INSERT statements, each literal inserted into a DATETIME column is converted to the appropriate datetime type. For example:

  ```
  insert into table1 (date_col) values ('1993-07-04')
  ```

  In UPDATE statements, each literal assigned to a DATETIME column is converted to the appropriate datetime type. For example:

  ```
  update table1 set date_col = current_date
  ```

  If the conversion fails, the statement is terminated and an error is returned.

Numeric Values

When a numeric value is inserted into a column or is used to update a column, the following rule applies.

An integer or the whole part of a decimal is never truncated; the fractional part can be truncated if necessary.

Comparison

Characters can be compared with other character values, datetime values can be compared with other datetime values, and numeric values can be compared with other numeric values. The rules for specific cases are given in the following sections.

Character Strings

For strings to be compared, they first must be of the same length. If two strings are of differing length, the shorter string is padded with space characters on the right to the length of the longer string. The corresponding characters of the two strings are then compared from left to right. The following rules apply:

- If all the characters are the same, the two strings are equal.
- If all the characters are not the same, the first pair of characters that are not equal defines a greater-than relationship between the strings; namely, the string whose character is the highest in the collating sequence of the system has the greater value.

Datetime Values

Datetime values can be compared as follows:

- Date values with date values.
- Time values with time values. In comparing values of different precision, the smaller precision is padded with zeros to the greater precision.

- Timestamp values with timestamp values. In comparing values of different precision, the smaller precision is padded with zeros to the greater precision.
- Date values with timestamp values: The date value is padded with a time value of midnight for the comparison.
- Time values with timestamp values: The time value is padded with a date value of 1900-01-01 for the comparison.
- A literal without a DATE, TIME, or TIMESTAMP prefix is interpreted to be an alternative datetime literal rather than a character literal *only if the content expects a datetime value*, in which case it is converted to a datetime value. For more information, refer to Appendix C, “Alternative Datetime Formats.”

Examples

In WHERE, HAVING, and WHEN clauses, each literal compared to a datetime column is converted to the appropriate datetime type. For example:

```
select *
from table1
where date_col <> '7-4-1993'
```

In select lists, if a literal occurs in a expression where a datetime datatype is expected, the literal is converted to the appropriate datetime type. For example:

```
select datediff (dy, date_col, '7-4-1993')
from table1
```

If the conversion fails, the statement is terminated and an error is returned.

Numeric Values

Numbers are compared according to their sign and magnitude. For example, +1 is greater than –5 although a magnitude of 5 is greater than a magnitude of 1. The mixed cases are compared as follows:

- If an integer is compared with a decimal value, the integer is temporarily converted to a decimal.
- If a decimal number is compared with another decimal number that has a shorter scale, the shorter scale is temporarily extended with trailing zeros.
- If a floating-point number is compared with a decimal or integer, the decimal or integer is temporarily converted to a floating-point number.
- If a single-precision floating point number (REAL) is compared with a double-precision floating point number (DOUBLE or FLOAT), the single-precision number is temporarily converted to a double-precision number.

Chapter

Expressions and Conditions

3

In This Chapter 3-3

Expressions 3-4
Simple Expression 3-4
Compound Expressions 3-5
Simple Expressions 3-5
Display Functions 3-5
Nested Expressions 3-6
Evaluation of Compound Expressions 3-7
Types of Operators 3-7
Datatype of Operation Result 3-8
Missing Values 3-9

Conditions. 3-10
Comparison Predicates 3-11
BETWEEN Predicate 3-13
EXISTS Predicate 3-14
IN Predicate 3-15
IS NULL Predicate. 3-16
LIKE Predicate 3-17
Wildcard Characters. 3-17
Search Condition 3-19
Evaluation 3-19
Order of Evaluation 3-20

In This Chapter

SQL, as implemented by Informix Red Brick Decision Server, contains a full set of arithmetic and logical operators and logical predicates.

This chapter is divided into the following sections:

- Expressions
  - Simple Expression
  - Compound Expressions
  - Evaluation of Compound Expressions
- Conditions
  - Comparison Predicates
  - BETWEEN Predicate
  - EXISTS Predicate
  - IN Predicate
  - IS NULL Predicate
  - LIKE Predicate
  - Search Condition

Expressions

An expression specifies a unique value. For example, the expression 5+12 evaluates to 17 and the expression

```
(sales.dollars/1000)
```

returns a value expressed as thousands of dollars. The value of the column name

```
sales.dollars
```

depends on the context. It has a specific value relative to a given row.

Simple Expression

A simple expression is one of the following values:

- Literal
- Scalar function
- Column name
- Set function
- RISQL display function
- Scalar subquery

For additional information about scalar subqueries, see "Scalar Subqueries and Table Subqueries" on page 7-61.

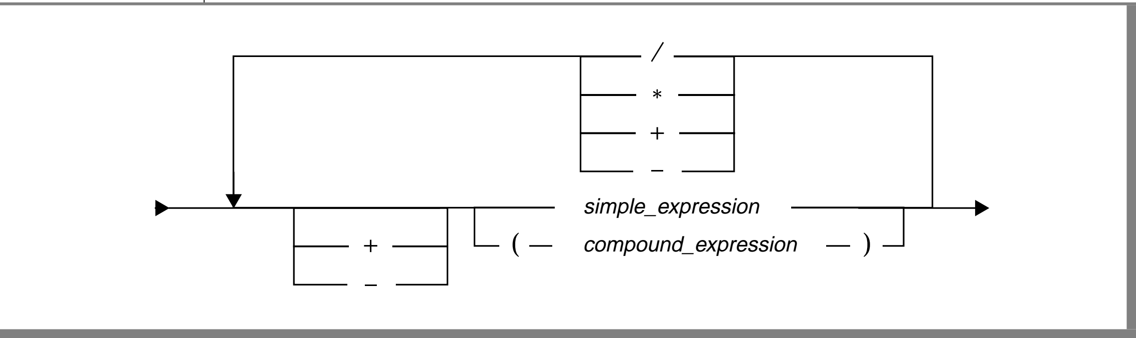
Examples

The following expressions are all simple expressions:

```
'Lotta Latte'
2.1416
sum(dollars)
rank(sales)
rank(sum(dollars))
integer(dollar_cents)
dollars
(select max(bonus) from employee)
```

Compound Expressions

A compound expression is a sequence of simple numeric expressions joined by arithmetic operators. The expressions can be grouped with parentheses. The following syntax diagram shows how to construct a compound expression:



Simple Expressions

A simple expression used in a compound expression must return a numeric value.

Display Functions

One or more RISQL Entry Tool display functions can be used in the same compound expression, as shown in the following examples with RANK. However, a display function cannot be part of an argument for either a set function or another display function.

For more details, refer to Chapter 6, "RISQL Display Functions."

Examples

The following expressions are all valid compound expressions:

```
(125+sales.dollars)/5
avg(sales.dollars/1000)
sum(sales.dollars)/count(*)
rank(sales)/100
rank(price)/rank(earnings)
rank(sum(dollars))/rank(sum(quantity))
ratiotoreport(sales)*100
(select salary from employee
    where emp_no = 227)*1.1
```

Nested Expressions

Some functions can be nested within other functions. You can use the expression

```
string(sum(dollars), 7, 2)
```

to truncate numeric values, for example.

You can also nest any scalar function within another scalar function. For example, the expression

```
string(current_date, 4)
```

returns the year portion of the date.

You *cannot* nest RISQL display functions within set functions or within other display functions; therefore, the following expressions return errors:

```
sum(rank(dollars))
rank(cume(dollars))
```

However, you can nest a set function within a display function. For example, you can use the expression

```
(rank(sum(dollars))
```

to sum sets of values in the Dollars column, then rank them.

Evaluation of Compound Expressions

The value of a compound expression depends on how the server evaluates the expression's components. The evaluation depends on the following:

- Types of operators and their precedence
- Datatypes of the operands
- Presence of missing values (NULL)

Types of Operators

A compound arithmetic expression can be constructed with the following operators:

Operator	Description
( )	Parentheses to control order of evaluation
+, –	Unary positive and negative operator
*, /	Multiplication and division
+, –	Addition and subtraction

Order of Precedence The operators are listed in order of precedence from highest to lowest (from top to bottom and, within a given level, left to right).

Unary Operators The unary operator plus (+) does not change the sign of its operand; the unary operator negative (–) reverses the sign of its operand. The unsigned numeric literals for zero (0, .0, 0.0) are positive values.

Parentheses Parentheses are optional but useful when the order of evaluation must be controlled. The server always evaluates expressions within a set of parentheses first. If parentheses are nested, the server first evaluates expressions within the innermost set of parentheses, then expressions within the next innermost set, and so on.

Examples

The following expressions illustrate the significance of operator precedence:

```
10*5+7 = 57
10*(5+7) = 120
```

Datatype of Operation Result

If both operands of a multiplication, addition, or subtraction operator are integers, the result of the operation is the smallest integer that can represent the result value. If the result value is too large to store as an integer, it is converted to a decimal value. This rule does not apply to the division operator, however. The division of two integers returns a numeric value, not an integer; integer division follows the same rules as decimal division.

If one operand is an integer and one operand is a decimal, the integer is temporarily converted to a decimal of zero scale. The resulting value is a decimal.

If both operands are decimal, the precision and scale of the result follow the rules given in the following table:

Expression	Precision of Result	Scale of Result
$d_1 + d_2$	$\max(p_1 - s_1, p_2 - s_2) + \max(s_1, s_2) + 1$	$\max(s_1, s_2)$
$d_1 - d_2$	$\max(p_1 - s_1, p_2 - s_2) + \max(s_1, s_2) + 1$	$\max(s_1, s_2)$
$d_1 * d_2$	$p_1 + p_2$	$s_1 + s_2$
d_1 / d_2	$\max(6, s_1 + p_2 - s_2 + 1) + p_1 - s_1 + s_2$	$\max(6, s_1 + p_2 - s_2 + 1)$

If either operand is a floating-point number, the operands are temporarily converted to floating-point numbers and the result is a floating-point number.

If the result precision for division operations is greater than 38, the scale is reduced as much as needed to bring the precision down to 38, but never to less than 6. If the scale is 6 and the precision is still greater than 38, the precision is set at 38 and a run-time overflow error is generated whenever results require more than 38 digits.

Missing Values

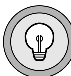
If an operand of an arithmetic operation is NULL, the arithmetic operation returns NULL. The following table defines the result of all arithmetic operations where *x* has missing or unknown information (NULL) and *y* contains a numeric value:

Operation	Result
–x	NULL
+x	NULL
x + y	NULL
x – y	NULL
x * y	NULL
x / y	NULL

Conditions

A condition is a statement about a row or a set of rows that evaluates to true, false, or unknown. Conditions are expressed with SQL comparison and quantified predicates. The following syntax diagram shows how to construct conditions with predicates:

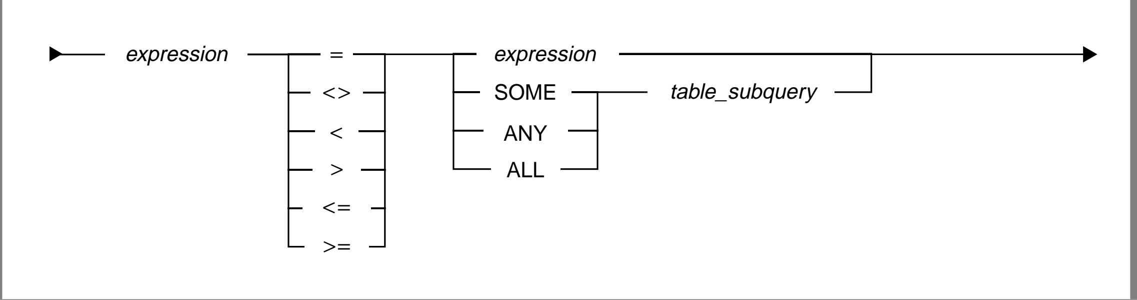
```
►►─┬─ expression ──┬─ =  ─┬──┬─ expression ─────────────────────┬──┬──►
   │               ├─ <> ─┤  ├─ SOME ─┬── table_subquery ───────┘  │
   │               ├─ <  ─┤  ├─ ANY  ─┤                            │
   │               ├─ >  ─┤  └─ ALL  ─┘                            │
   │               ├─ <= ─┤                                        │
   │               └─ >= ─┘                                        │
   ├─ expression ──┬───────┬─ BETWEEN ─ expression ─ AND ─ expression ─┤
   │               └─ NOT ─┘                                        │
   ├─ expression ─── IS ─┬───────┬─ NULL ───────────────────────────┤
   │                     └─ NOT ─┘                                  │
   ├─ expression ─┬───────┬─ LIKE ─ string_expression ─┬──────────────────────────────┬─┤
   │              └─ NOT ─┘                            └─ ESCAPE ─ string_expression ─┘ │
   ├─┬───────┬─ EXISTS ─── table_subquery ──────────────────────────┤
   │ └─ NOT ─┘                                                      │
   └─ expression ─┬───────┬─ IN ─┬─ table_subquery ──────────┬──────┘
                  └─ NOT ─┘      │      ┌──── , ────┐        │
                                 └─ ( ─▼── constant ─┴── ) ──┘
```



Tip: *Expressions can be character, datetime, or numeric, but when multiple expressions occur in a condition, they must have compatible datatypes.*

Comparison Predicates

A comparison predicate states a logical relationship between two values: the comparison is true, false, or unknown with respect to a given row. The following syntax diagram shows how to construct a condition with comparison predicates:



expression Expressions can be character, datetime, or numeric, but they must be compatible datatypes. If the value of the expression is NULL or a subquery returns NULL, the condition evaluates to unknown.

An expression can be a scalar subquery, but not a row subquery or table subquery.

SOME, ANY, ALL ***table_subquery*** A quantified predicate compares an expression with a set of values returned by a table subquery. Although the subquery can have multiple rows, it is restricted to one column in its select list.

The SOME and ANY quantifiers are synonyms. A comparison expressed with SOME or ANY is true if the comparison is true for at least one value returned by the subquery. A comparison expressed with the ALL quantifier is true if the comparison is true for all values returned by the subquery.

If a subquery returns no values and the SOME or ANY quantifier is used, the comparison is false. However, if a subquery returns no values and the ALL quantifier is used, the comparison is true.

Examples

The following condition is true for each value of the Dollars column that equals or exceeds one hundred:

```
dollars >= 100
```

The following condition is true if Brand is equal to any product in the Hot_Products table:

```
brand = any
    (select product from hot_products)
```

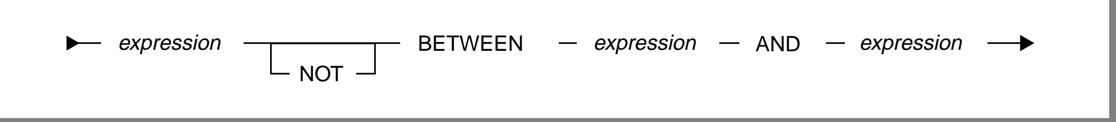
The following condition is true for each value of the Dollars column that equals or exceeds every one of the sales in San Jose. If the subquery returns no values (no sales were made in San Francisco), the condition is true; each value in the Dollars column equals or exceeds nothing.

```
dollars >= all
    (select dollars from store join sales
            on store.storekey = sales.storekey
        where city = 'San Jose')
```

For more detailed examples of the SOME and ALL predicates, refer to the *SQL Self-Study Guide*.

BETWEEN Predicate

The BETWEEN predicate determines whether a value lies within a specified range. The following syntax diagram shows how to construct a condition with the BETWEEN predicate:



expression Expressions can be character, datetime, or numeric, but they must be compatible datatypes.

The first *expression* in a BETWEEN predicate must be the lesser value and the second *expression* the greater. For example, the following condition is always false:

```
between 12 and 1
```

BETWEEN The condition x between y and z is equivalent to

```
(x >=y) and (x<=z)
```

NOT BETWEEN The condition x not between y and z is equivalent to

```
(x<y) OR (x>z)
```

Examples

The following condition is true for those values of the Dollars column that are greater than or equal to 200 and less than or equal to 500:

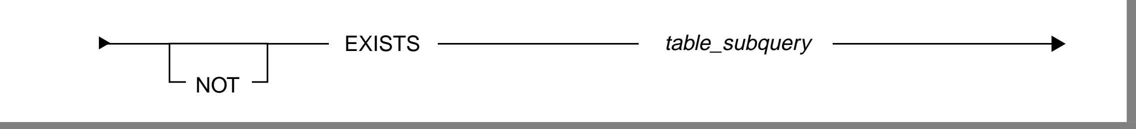
```
dollars between 200 and 500
```

The following condition is true only for those values of the Dollars column that are less than and not equal to 200 or greater than 500:

```
(dollars not between 200 and 500)
```

EXISTS Predicate

The EXISTS predicate evaluates to true if a subquery returns at least one row. If NOT is specified, the predicate evaluates to true if a subquery returns no rows. The following syntax diagram shows how to construct a condition with the EXISTS predicate:



table_subquery A subquery that evaluates to a table with one or more columns and one or more rows.

Examples

The following condition is true if the subquery returns at least one row:

```
exists (select prod_name from store
where population > 5000)
```

The following condition is true if the subquery returns no rows:

```
not exists (select prod_name from store
where population > 5000)
```

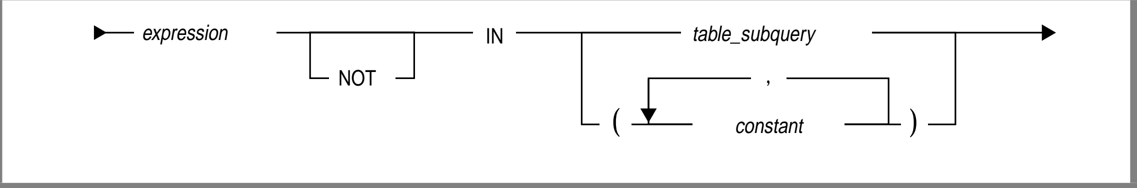
The following query returns all the rows from the Deal table if the subquery returns at least one row:

```
select * from deal
where exists
    (select dealkey, discount from deal
    union
    select promokey, value from promotion);
```

For a more detailed example of the EXISTS predicate, refer to the *SQL Self-Study Guide*.

IN Predicate

The IN predicate compares a value in a column with a set of values. The following syntax diagram shows how to construct a condition with the IN predicate:



expression — The *expression* can be a character, datetime, or numeric datatype but must be a datatype that is compatible with the *constant* (s) or the values returned from the *row_subquery.*

table_subquery — A subquery that evaluates to a table with one or more rows. In this case, however, the subquery is limited to one column in its select list.

Examples

- The condition

  ```
  quantity in (1000, 10000, 100000)
  ```

  is true only for those values equal to 1000, 10000, or 100000.

- The condition

  ```
  quantity not in (1000, 10000, 100000)
  ```

  is true only for those values not equal to 1000, 10000, or 100000.

- The condition

  ```
  month in ('JAN', 'FEB', 'MAR')
  ```

  is true only for months equal to *JAN*, *FEB*, or *MAR*.

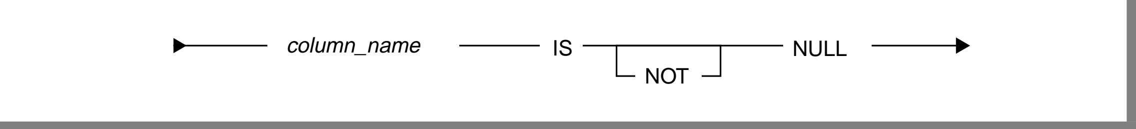
- The condition

  ```
  product in (select product from hot_products)
  ```

  is true only for those products equal to a Product value in the Hot_Products table.

IS NULL Predicate

The IS NULL predicate tests a column for missing values. The following syntax diagram shows how to construct a condition with the IS NULL predicate:



IS NULL The IS NULL predicate is true when the column's value is missing and false when the column has a value.

IS NOT NULL The IS NOT NULL predicate is false when the column's value is missing and true when the column has a value.

Examples

The following condition evaluates to true whenever a value is missing from the Product column:

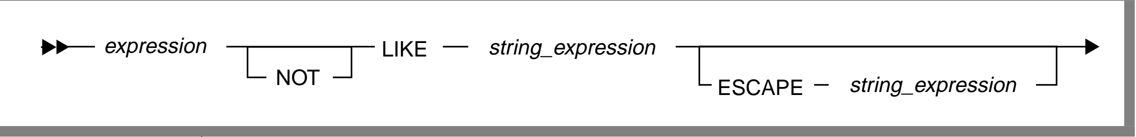
```
product is null
```

The following condition evaluates to false whenever a value is missing from the Product column:

```
product is not null
```

LIKE Predicate

The LIKE predicate compares the values in a column with a completely specified character literal or expression or with a character pattern specified with the wildcard characters percent (%) and underscore (_). The following syntax diagram shows how to construct a condition with the LIKE predicate:



expression — Columns used in expressions with LIKE predicates must contain character datatypes; they cannot contain numeric or datetime datatypes.

string_expression — Specifies the character-string literal or character-string expression with which the values in *expression* will be compared.

ESCAPE — The ESCAPE keyword defines a one-character *string_expression* to serve as an escape character so that wildcards can be treated as character literals or expressions rather than control characters. Use the ESCAPE keyword when the character pattern to be matched contains a percent or underscore character itself (see the last example below).

The ESCAPE *string_expression* must evaluate to a one-character string but that character can be single-byte or multibyte.

Wildcard Characters

The percent (%) wildcard character matches any character string. The underscore (_) wildcard character matches any one character in a fixed position. These wildcards must be specified with single-byte ASCII characters; multibyte versions of these characters are treated as literal values.

Examples

- city like '%ville'

  True for any character string in the City column that ends with *ville*. In this example, if the character string contains trailing blanks, the condition is not true.
- city like '%son%'

  True for any character string in the City column that contains *son*.
- city like 'San%'

  True for any character string in the City column that begins with *San*.
- prod_name like '_EE%'

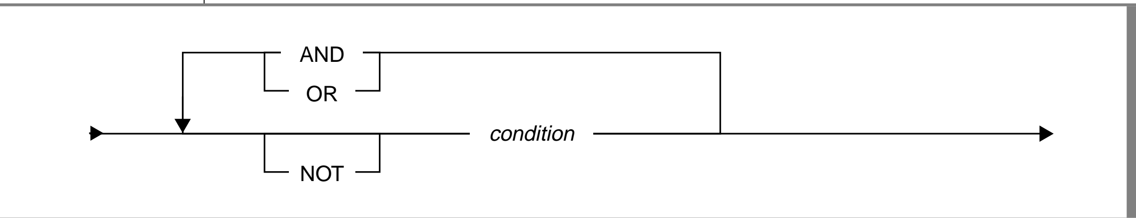
  True for any character string in the Product column whose second and third characters are *EE*.
- prod_name like '%LE_N%'

  True for any character string in the Product column that matches the specified pattern. The strings *CLEAN*, *KLEEN*, and *EXCEPTIONALLY KLEEN* match this pattern.
- sales_pct like '%Monthly \%' escape '\'

  True for any character string that ends with *Monthly %*.

Search Condition

A search condition specifies a logical condition that evaluates to true, false, or unknown. Compound search conditions are constructed from basic conditions using the logical connectives. The conditions can be grouped with parentheses. The following syntax diagram shows how to construct a search condition:



Evaluation

The value of a compound search condition is determined by the values of its components. Compound conditions are evaluated as follows:

C1	C2	C1 AND C2	C1 OR C2	NOT C2
True	True	True	True	False
True	False	False	True	True
True	Unknown	Unknown	True	Unknown
False	True	False	True	
False	False	False	False	
False	Unknown	False	Unknown	
Unknown	True	Unknown	True	
Unknown	False	False	Unknown	
Unknown	Unknown	Unknown	Unknown	

Order of Evaluation

If the order of evaluation is not specified by parentheses, the NOT operator is evaluated before the AND, and the AND is evaluated before the OR.

Examples

The following search condition selects only rows that have *NY* in their State column and *1999* in their Year column:

```
state = 'NY' and year = 1999
```

The following search condition selects only those rows that have *NY* or *GA* in their State column and *2000* in their Year column:

```
(state = 'NY' or state = 'GA') and year = 2000
```

Chapter

4

Set Functions

In This Chapter 4-3
AVG 4-4
COUNT 4-6
MAX 4-8
MIN 4-9
SUM 4-10

In This Chapter

Set functions operate on a value or a set of values and return a single value as the result. For example, the expression

```
sum(dollars)
```

evaluates to the sum of the Dollars column for a set of rows.

Red Brick Decision Server supports the following set functions, which are described in alphabetical order in this chapter:

Function	Description
AVG	Calculates the average of all values.
COUNT	Counts the number of rows.
MAX	Determines the maximum value.
MIN	Determines the minimum value.
SUM	Calculates the sum of all values.

These functions are defined in the ANSI SQL-92 standard. They are sometimes referred to as *aggregation functions* (because they compute aggregates), *group functions* (because they operate on a group of values), and *column functions* (because they operate on values in a column).

Set functions can occur in a select list or a HAVING clause. Each expression can contain only one set function. Nested set functions—that is, a set function within a set function—are not allowed. For example, the following select list returns an error:

```
select max(avg(salary))
```

Set functions cannot occur in the search condition of a WHERE clause.

RISQL display functions, such as CUME and RANK, cannot be used as arguments to SET functions. For example, the following expression returns an error:

```
sum(rank(dollars))
```

However, set functions can be used as arguments to RISQL display functions. For example:

```
rank(sum(dollars))
```

For a discussion of grouping requirements when SET functions are used in queries, refer to "GROUP BY Clause" on page 7-29.

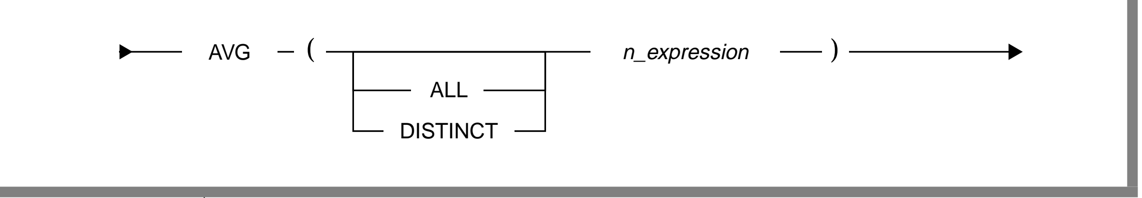
For information about using the Red Brick Vista option to accelerate the performance of queries that contain aggregate functions, refer to the *Informix Vista User's Guide*.

AVG

The AVG function returns the average of a specified set of values.

Syntax

The following syntax diagram shows how to construct an expression with the AVG function:



n_expression The argument *n_expression* must be numeric and must not reference a set or display function.

ALL The function retains all duplicate values from *n_expression* for calculating the average. ALL is the default.

DISTINCT The function eliminates all duplicate values from the specified expression before calculating the average.

Result

If the specified set of values is non-empty, AVG returns their arithmetic average; otherwise, it returns NULL.

If the datatype of *n_expression* is an exact datatype (TINYINT, SMALLINT, INTEGER, or DECIMAL), AVG returns an exact datatype. The precision and scale of the result datatype are such that the number of digits to the left of the decimal point is maintained, while the number of digits to the right of the decimal point is increased by six. This means that even if the resulting value of the AVG function is very small, it will probably still fit into the significant digits of the result datatype.

The following table summarizes the datatypes returned by AVG for different *n_expression* datatypes:

Datatype of *n_expression*	Datatype of Result
TINYINT	DECIMAL(9,6)
SMALLINT	DECIMAL(11,6)
INTEGER	DECIMAL(16,6)
DECIMAL NUMERIC	DECIMAL(p,s) p = min(38, precision of *n_expression* + 6) s = min(6, 38 – precision of *n_expression*)
REAL	REAL
FLOAT DOUBLE PRECISION	FLOAT DOUBLE PRECISION

Example

The following query returns the average daily sales total for Demitasse MS coffee:

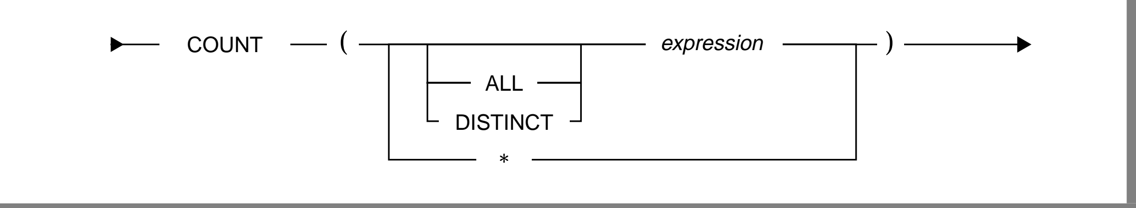
```
select avg(dollars) as sales_avg
from sales join product on sales.classkey = product.classkey
    and sales.prodkey = product.prodkey
where prod_name = 'Demitasse Ms';

SALES_AVG
204.29551820
```

COUNT

The COUNT function returns the number of rows in a specified set of rows.

Syntax

The following syntax diagram shows how to construct an expression with the COUNT function:



ALL If the *expression* is a column name preceded by the ALL keyword (or no keyword), COUNT returns the number of rows that have values in the specified column. Rows with missing values (NULLs) in the column are not counted.

DISTINCT If the *expression* is a column name preceded by the DISTINCT keyword, COUNT eliminates rows with duplicate values in the specified column before counting. Rows with NULLs in the column are not counted.

expression If the *expression* is a column name, COUNT returns the number of rows that have values in the specified column. Rows with missing values (NULLs) in the column are not counted.

The *expression* must not reference a set or display function.

* If the argument is an asterisk (*), COUNT returns the number of rows in the set (zero for an empty set). This function is also referred to as the “count star” function. Rows that include NULLs are counted.

Result

The value returned by the function is always a non-negative integer value: The function never returns NULL.

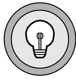
Examples

The following query counts the number of *products* in the Product table:

```
select count(prod_name) as prod_count
from product
PROD_COUNT
59
```

The following query counts the number of *distinct product names* in the Product table:

```
select count(distinct prod_name) as prod_count
from product
PROD_COUNT
38
```



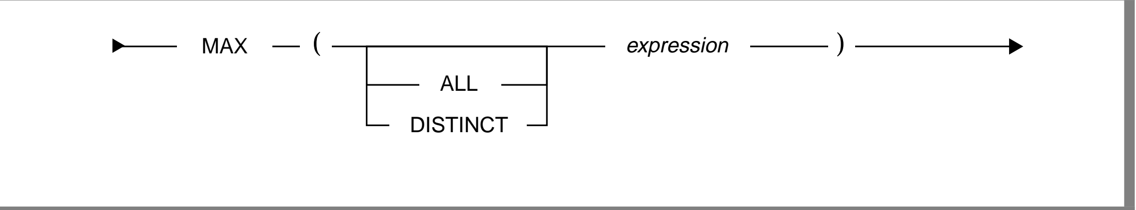
***Tip**: The second example does not count the number of products but the number of different product names. The number of distinct products is smaller because, in the Aroma database, the same name is given to coffee and tea products in the bulk and pre-packed classes.*

MAX

The MAX function returns the maximum value from a specified set of values.

Syntax

The following syntax diagram shows how to construct an expression with the MAX function:



expression The *expression* must not reference a set or display function.

ALL The ALL keyword retains duplicate values in the specified set of values but has no effect on the result.

DISTINCT The DISTINCT keyword eliminates duplicate values from the specified set of values but has no effect on the result.

Result

If the specified set of values is non-empty, MAX returns the maximum value; otherwise, it returns NULL.

Example

The following query returns the maximum Dollars and Quantity values (daily totals) for sales of the Coffee Sampler product:

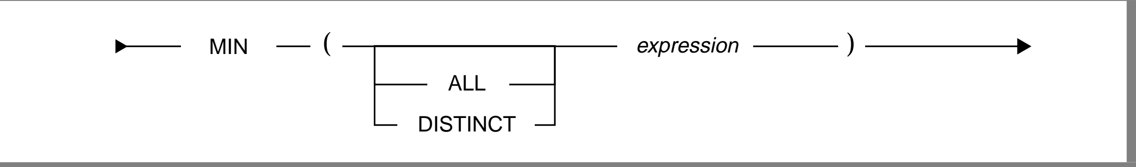
```
select max(dollars) as max_dol, max(quantity) as max_qty
from product natural join sales
where prod_name = 'Coffee Sampler'
MAX_DOL    MAX_QTY
570.00     19
```

MIN

The MIN function returns the minimum value from a specified set of values.

Syntax

The following syntax diagram shows how to construct an expression with the MIN function:



expression The *expression* must not reference a set or display function.

ALL The ALL keyword retains duplicate values in the specified set of values but has no effect on the result.

DISTINCT The DISTINCT keyword eliminates duplicate values from the specified set of values but has no effect on the result.

Result

If the specified set of values is non-empty, MIN returns the minimum value; otherwise, it returns NULL.

Example

The following query returns the minimum Dollars and Quantity values (daily totals) for sales at the East Coast Roast store:

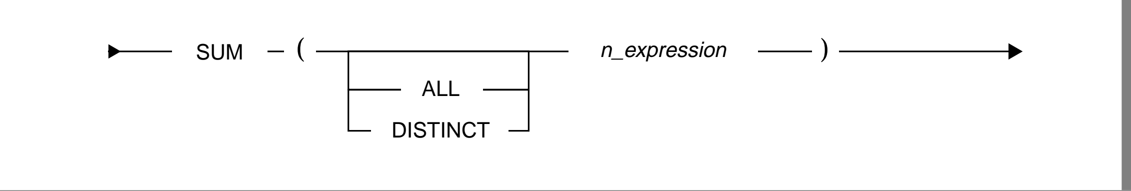
```
select min(dollars) as min_dol, min(quantity) as min_qty
from store natural join sales
where store_name = 'East Coast Roast'
MIN_DOL MIN_QTY
3.751
```

SUM

The SUM function calculates the sum of a specified set of values.

Syntax

The following syntax diagram shows how to construct an expression with the SUM function:



n_expression The argument *n_expression* must be numeric and must not reference a set or display function.

ALL The ALL keyword retains duplicate values before calculating the total. ALL is the default.

DISTINCT If the DISTINCT keyword is included, the function eliminates all duplicate values before calculating the total.

Result

If the specified set of values is non-empty, SUM returns their total; otherwise, it returns NULL.

If the datatype of *n_expression* is an exact datatype (TINYINT, SMALLINT, INTEGER, or DECIMAL), SUM returns an exact datatype. To reduce the chance of the result overflowing its allocated storage, the precision of the result datatype is expanded by six.

The following table summarizes the datatypes returned by SUM for different *n_expression* datatypes:

Datatype of *n_expression*	Datatype of Result
TINYINT	DECIMAL(9,0)
SMALLINT	DECIMAL(11,0)
INTEGER	DECIMAL(16,0)
DECIMAL NUMERIC	DECIMAL(p,s) p = min(38, precision of *n_expression* + 6) s = scale of *n_expression*
REAL	REAL
FLOAT DOUBLE PRECISION	FLOAT DOUBLE PRECISION

Example

The following query returns the total sales dollars for all packaged tea products:

```
select sum(dollars) as tea_dollars
from class natural join product
    natural join sales
where class_type = 'Pkg_tea';
TEA_DOLLARS
510507.25
```

Chapter

Scalar Functions

5

In This Chapter 5-3

Conditional Scalar Functions 5-4
CASE 5-4
COALESCE 5-8
DECODE 5-9
IFNULL 5-11
NULLIF 5-13

Numeric Scalar Functions 5-15
ABS 5-15
CEIL 5-17
DEC 5-19
EXP 5-20
FLOAT 5-22
FLOOR 5-23
INT 5-25
LN 5-27
REAL 5-28
SIGN 5-29
SQRT 5-31
Macros for Statistical Functions 5-32
Power. 5-32
Log 5-33
Log10 5-33
Standard Deviation of a Population 5-33
Sample Estimate of the Population Standard Deviation . . . 5-33
Variance 5-34

String Scalar Functions . . . 5-35
CONCAT . . . 5-36
LENGTH . . . 5-37
LENGTHB . . . 5-39
LOWER . . . 5-40
LTRIM . . . 5-41
RTRIM . . . 5-42
STRING . . . 5-44
SUBSTR . . . 5-47
Handling Trailing Blanks . . . 5-48
SUBSTRB . . . 5-49
TRIM . . . 5-51
UPPER . . . 5-52

Datetime Scalar Functions . . . 5-54
Dateparts for Datetime Scalar Functions . . . 5-54
CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP . . 5-56
DATE . . . 5-57
DATEADD . . . 5-58
DATEDIFF . . . 5-59
DATENAME. . . . 5-61
EXTRACT. . . . 5-63
TIME . . . 5-65
TIMESTAMP. . . . 5-66

CURRENT_USER Function . . . 5-67

In This Chapter

The RISQL extensions include scalar functions—functions that operate on an expression one row at a time. You can construct compound expressions with scalar functions and you can nest a scalar function inside another scalar function. RISQL display functions, discussed in Chapter 6, can also be used as arguments to scalar functions.

Scalar functions fall into the following categories (which represent the main sections of this chapter):

- Conditional Scalar Functions:
  - CASE, COALESCE, DECODE, IFNULL, NULLIF

    (CASE, COALESCE, and NULLIF are defined in the ANSI SQL-92 standard.)
- Numeric Scalar Functions:
  - ABS, CEIL, DEC, EXP, FLOAT, FLOOR, INT, LN, SIGN, SQRT
- String Scalar Functions:
  - CONCAT, LENGTH, LENGTHB, LOWER, LTRIM, RTRIM, STRING, SUBSTR, SUBSTRB, TRIM, UPPER
- Datetime Scalar Functions:
  - CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, DATE, DATEADD, DATEDIFF, DATENAME, EXTRACT, TIME, TIMESTAMP
- CURRENT_USER Function—an informational scalar function

***Tip:** If a divisor of zero is used in a query, the setting for the ARITHIGNORE option determines whether an error or NULL is returned. ARITHIGNORE can be specified globally with OPTION ARITHIGNORE in the rbw.config file or for the current session with the SET ARITHIGNORE command, as described on page 9-7.*

Conditional Scalar Functions

The conditional scalar functions are listed in the table below. The values returned by these functions depend on the conditions that are met by the function arguments:

Conditional Scalar Function	Description
CASE	Returns a value that depends on which of the specified conditions is met.
COALESCE	Returns the first value specified that is not NULL.
DECODE	Replaces a value based on a target value.
IFNULL	Tests for NULL and returns a value.
NULLIF	Compares two values and returns NULL if the two values are equal.

CASE

A CASE expression returns a value that depends on which of the specified set of conditions is met.

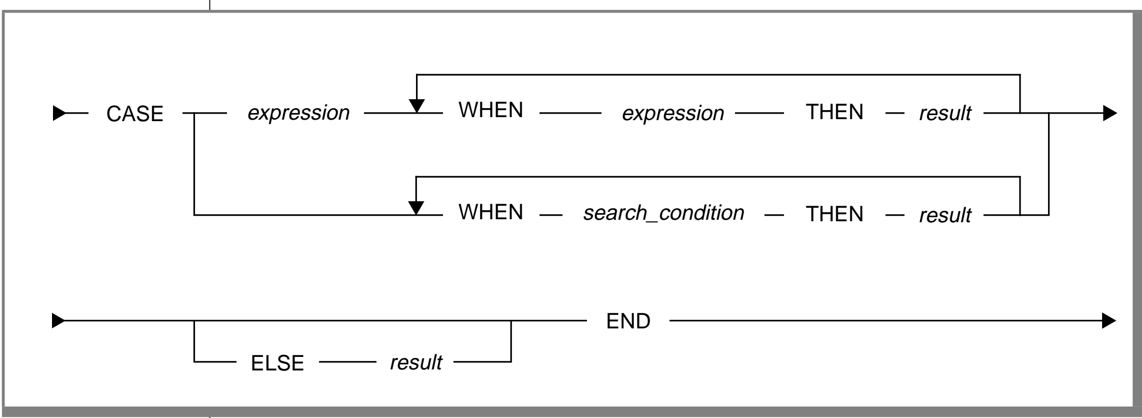
Syntax

There are two forms of CASE expression: *simple* CASE and *searched* CASE.

In the simple CASE expression, an initial *expression* precedes the WHEN clause. The WHEN clause contains an arbitrary number of *expressions* and corresponding *results*.

In the searched CASE expression, the WHEN clause takes an arbitrary number of *search_conditions*, each having a corresponding *result*.

The following syntax diagram shows how to construct both types of CASE expression. Note that a simple CASE expression uses the upper WHEN clause while the searched CASE expression uses the lower WHEN clause.



expression Any type of expression. If two compared *expressions* do not have the same datatype, there must be a legal implicit conversion between the datatypes. The conversion rules for *expression* datatypes are the same as those listed for the UNION, INTERSECT, or EXCEPT operators on page 7-36.

result The *result* can be any type of expression. If the *results* do not all have the same datatype, there must be a legal implicit conversion between the datatypes. The conversion rules for *result* datatypes are the same as those listed for the UNION, INTERSECT, or EXCEPT operators on page 7-36. The *result* datatype need not be the same as the *expression* datatype. A *result* can be NULL; however, at least one of the *results* (in the THEN or ELSE clauses) must be non-NULL so that the CASE expression has a defined return type.

search_condition The *search_condition* specifies a logical condition that evaluates to TRUE, FALSE, or UNKNOWN. For the syntax of search conditions, refer to "Search Condition" on page 3-19.

Result

With a simple CASE expression, the CASE function compares the values of each *expression* in the WHEN clause to the value of the initial *expression*. The *expressions* in the WHEN clause are evaluated in order. If the compared values are equal, the CASE function returns the value of the corresponding *result* and stops processing.

With a searched CASE expression, the CASE function evaluates each *search_condition* in order. If a *search_condition* evaluates to TRUE, the CASE function returns the value of the corresponding *result* and stops processing.

In both simple CASE and searched CASE forms, if none of the WHEN clause conditions are met, the function returns the value of the *result* in the ELSE clause. If no ELSE clause is specified, the default return value is NULL.

Usage Notes

The simple CASE expression has the same functionality as the DECODE function. Informix recommends using CASE expressions instead of DECODE because CASE is consistent with the ANSI SQL-92 standard.

A simple CASE expression can always be expressed as a searched CASE expression; however, the simple CASE expression represents an optimization because the initial *expression* is only evaluated once. Therefore, the simple CASE expression is slightly faster and should be used where applicable.

Examples

The following query uses the simple form of the CASE expression to replace the value for quarter in the query output with a more descriptive string:

```
select case qtr
    when 'Q1_98' then '1st Quarter'
    when 'Q2_98' then '2nd Quarter'
    when 'Q3_98' then '3rd Quarter'
    when 'Q4_98' then '4th Quarter'
    end as Period,
    sum (dollars) as results
from sales natural join period
where year = 1998
group by qtr
PERIOD          RESULTS
1st Quarter     723532.35
2nd Quarter     756282.05
3rd Quarter     778795.20
4th Quarter     782359.05
```

The following query uses the searched form of the CASE expression to return two separate sums on the same column (Dollars):

```
select year,
    sum (case when region = 'West' then dollars else 0 end)
        as West_Region,
    sum (case when ((region = 'Central') or (region = 'North')
        or (region = 'South')) then dollars
        else 0 end) as Other_Regions
from sales, store, market, period
where sales.perkey = period.perkey
    and sales.storekey = store.storekey
    and store.mktkey = market.mktkey
group by year
order by year
YEAR        WEST_REGION      OTHER_REGIONS
       1998      1164414.20       1876554.45
       1999      1195795.10       2084195.95
       2000       296379.45        511010.95
```

For additional examples of CASE expressions, refer to the *SQL Self-Study Guide*.

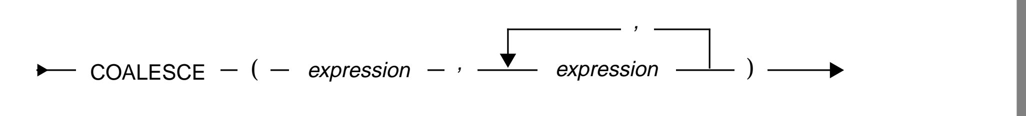
COALESCE

The COALESCE function returns the value of the first argument that does not evaluate to NULL.

COALESCE is an ANSI-defined short-hand for a specific use of the CASE function. Any COALESCE expression can also be expressed as a CASE function.

Syntax

The following syntax diagram shows how to construct an expression with the COALESCE function:



expressions If the *expressions* do not have the same datatype, there must be a legal implicit conversion between the expression datatypes. The conversion rules for *expression* datatypes are the same as those listed for the UNION operation on page 7-36. There must be at least two *expressions*.

Result

The COALESCE function tests the value of each argument in the order they are specified. The function returns the value of the first argument that is not NULL. If all the arguments are NULL, COALESCE returns NULL.

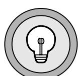
Example

The following COALESCE expression returns one Date column in the result set:

```
coalesce(orders.close_date, line_items.receive_date) as date
```

DECODE

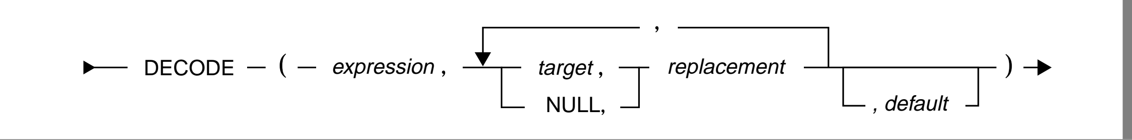
The DECODE function compares and converts an expression to another value.



***Tip:** The DECODE function has the same functionality as the simple CASE function. Informix recommends that you use the CASE function in place of the DECODE function because CASE is included in the ANSI SQL-92 standard.*

Syntax

The following syntax diagram shows how to construct an expression with the DECODE function:



expression	The expression can be any datatype.
target	The *target* arguments must be of the same datatype as the initial argument *expression*.
replacement	The *replacement* arguments can be of any datatype but must all be of the same datatype.
default	If a *default* argument is specified, it must be of the same datatype as the *replacement* arguments.

Result

If *expression* matches *target*, it is replaced by the corresponding *replacement*; otherwise, *expression* is replaced by *default*, or by NULL if no *default* is specified.

***Tip:** The value specified for the initial expression can be specified as the default as well so that no replacement occurs when the expression fails to match the target.*

If *expression* is a character string, the maximum size of the result is the maximum size of *replacement*; if *expression* is numeric, the datatype of the result is the same as the *replacement* of greatest precision.

The null case can be detected by including the character literal NULL (four letters only, no quotes) as *target*. A corresponding *replacement* for the null cases must be provided.

Examples

The following query returns the total quantities of each product sold in Los Angeles:

```
select prod_name,
    sum (decode (city, 'Los Angeles', dollars, 0.0)) as LA
from store sr, product pr, period pd, sales sl
where sl.storekey = sr.storekey
    and sl.classkey = pr.classkey
    and sl.prodkey = pr.prodkey
    and sl.perkey = pd.perkey
and year = 1998
group by prod_name
PROD_NAME           LA
Aroma Roma          1989
Aroma Sheffield Steel Teapot 51
Aroma Sounds CD     177
Aroma Sounds Cassette 180
Aroma baseball cap  440
Aroma t-shirt       529
Assam Gold Blend    791
Assam Grade A       898
Breakfast Blend     832
Cafe Au Lait        1339
Christmas Sampler   4
Coffee Mug          39
...
```

If the City value is Los Angeles, the corresponding Dollar value is returned; otherwise, zero (0.0) is returned. The syntax of this DECODE function translates as follows:

- `city` = *expression*
- `'Los Angeles'` = *target*
- `dollars` = *replacement*
- `0.0` = *default*

The following query replaces the value for the Quarter column in the query output with a more descriptive string:

```
select decode(qtr,
        'Q1_98', '1st Quarter',
        'Q2_98', '2nd Quarter',
        'Q3_98', '3rd Quarter',
        'Q4_98', '4th Quarter') as Period,
    sum(dollars) as results
    from sales, period
    where sales.perkey = period.perkey
        and year = 1998
    group by qtr
PERIODRESULTS
1st Quarter723532.35
2nd Quarter756282.05
3rd Quarter778795.20
4th Quarter782359.05
```

IFNULL

The IFNULL function tests an expression for missing values and replaces each one with a specified value.

The IFNULL function is a special case of the COALESCE function. Informix recommends using COALESCE in place of IFNULL because COALESCE is included in the ANSI SQL-92 standard.

Syntax

The following syntax diagram shows how to construct an expression with the IFNULL function:

```
►── IFNULL ─ ( ─ expression , substitute ─ ) ──────────►
```

expression The *expression* can be any datatype.

substitute The substitute value must be a datatype that is compatible with the expression datatype. If the expression and substitute arguments do not have the same datatype, there must be a legal implicit conversion between the two datatypes. The conversion rules are the same as those listed under "Datatype Conversions" on page 7-40.

Result

If the *expression* is NULL, the function returns the *substitute*; otherwise, it returns the value of the expression.

If the *expression* and the *substitute* are the same datatype, the datatype of the result is also that datatype. If they are different datatypes, the datatype of the result is the datatype of the implicit conversion.

Examples

- `ifnull(market, 'New City')`—replaces each missing value with *New City*.
- `ifnull(dollars, 0.0)`—replaces NULLs in the Dollars column with 0.0.

NULLIF

The NULLIF function compares two expressions. If the expressions have the same value, the function returns NULL; otherwise, the value of the first expression is returned.

Syntax

The following syntax diagram shows how to construct an expression with the NULLIF function:

expression, expression — If the expressions do not have the same datatype, there must be a legal implicit conversion between the expression datatypes. The conversion rules for ***expression*** datatypes are the same as those listed for the UNION operation on page 7-36.

Result

The NULLIF function returns NULL if both expressions have the same value. If the expressions have different values, the value of the first expression is returned.

Usage Notes

You can use the NULLIF function in queries that involve division calculations to replace zero values with NULL, thereby avoiding a possible division by 0. The same goal can be achieved by using the SET ARITHIGNORE command. For details, refer to “SET ARITHIGNORE, ARITHABORT” on page 9-7.

Example

The following query returns the value of the City column, unless the value is *San Jose*, in which case NULL is returned:

```
select prod_name,
    nullif(city, 'San Jose') as not_SJ, sum(dollars) as totals
from sales sl, store st, product pr, class cl
where cl.classkey = pr.classkey
    and sl.classkey = pr.classkey
    and sl.prodkey = pr.prodkey
    and sl.storekey = st.storekey
    and class_type like 'Gifts%'
group by prod_name, city
order by prod_name
PROD_NAME                      NOT_SJ                TOTALS
Aroma Sounds CD                Miami                 2883.00
Aroma Sounds CD                NULL                  4480.00
Aroma Sounds CD                Atlanta               4329.00
Aroma Sounds CD                Los Angeles           4087.00
Aroma Sounds Cassette          Los Angeles           2786.50
Aroma Sounds Cassette          NULL                  2795.50
Aroma Sounds Cassette          Atlanta               2640.00
Aroma Sounds Cassette          Miami                 3420.50
Christmas Sampler              Atlanta               210.00
Christmas Sampler              Los Angeles           270.00
Christmas Sampler              NULL                  1140.00
Christmas Sampler              Miami                 300.00
...
```

Numeric Scalar Functions

The numeric scalar functions operate on numeric expressions or character expressions representing numeric values. The table below lists the numeric scalar functions:

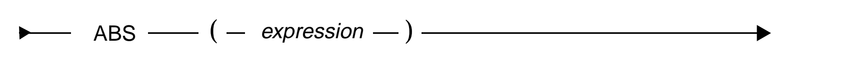
Numeric Scalar Function	Description
ABS	Returns the absolute value of a numeric expression.
CEIL	Returns the closest integral value greater than or equal to a numeric expression.
DEC	Converts character or numeric datatypes to DECIMAL
FLOAT	Converts character or numeric datatypes to FLOAT.
FLOOR	Returns the closest integral value less than or equal to a numeric expression.
INT	Converts character or numeric datatypes to INTEGER.
REAL	Converts a specified value into a REAL datatype.
SIGN	Returns the sign of a numeric expression.

ABS

The ABS function returns the absolute value of a numeric expression.

Syntax

The following syntax diagram shows how to construct an expression with the ABS function:



expression The *expression* can be a numeric or character datatype. If the *expression* is a character string, it must represent a numeric value. For example:

`'19.2'`—valid

`'RAJ'`—not valid

Result

If the argument is a REAL, FLOAT, DOUBLE PRECISION, DECIMAL, or NUMERIC datatype, the ABS function calculates the absolute value, and returns this value, preserving the input datatype.

If the argument is an INTEGER, SMALLINT, or TINYINT, the ABS function calculates the absolute value and returns the absolute value as an integer.

If the argument is a character string representing a numeric value, the ABS function converts this value to a double-precision floating point, calculates the absolute value, and returns the absolute value as a double-precision floating point. If the character string does not represent a numeric value, the ABS function returns an error. If the argument is NULL, the ABS function returns NULL.

Example

The following query takes the absolute value of a column with a decimal datatype (Numvalue) and adds it to the absolute value of a column with a character datatype (Stringnum):

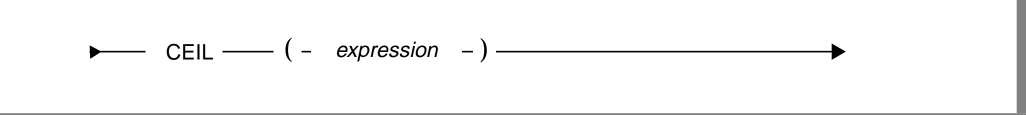
```
select numvalue, stringnum, (abs(numvalue) + abs(stringnum))
    as total from table_1
numvalue        stringnum total
-9.45           8.30     17.75
-2.05           -8.05    10.10
```

CEIL

The CEIL function returns the nearest integral value greater than or equal to the value of the function argument.

Syntax

The following syntax diagram shows how to construct an expression with the CEIL function:



expression The argument can be a numeric or character datatype. If the *expression* is a character string, it must represent a numeric value. For example:

`'19.2'`—valid

`'RAJ'`—not valid

Result

If the datatype of the argument is REAL, FLOAT, DOUBLE PRECISION, DECIMAL, or NUMERIC, the CEIL function calculates and returns the ceiling value, preserving the input datatype.

If the argument is an INTEGER, SMALLINT, or TINYINT value, the CEIL function returns this value immediately, preserving the input datatype.

If the argument is a character string representing a numeric value, the CEIL function converts this value to a double-precision floating point, calculates the ceiling value, and returns the ceiling value as a double precision. If the character string does not represent a numeric value, the CEIL function returns an error.

If the argument is NULL, the CEIL function returns a NULL result.

Example

The following query computes a price-per-item value, which is rounded up to the nearest integral value using the CEIL function:

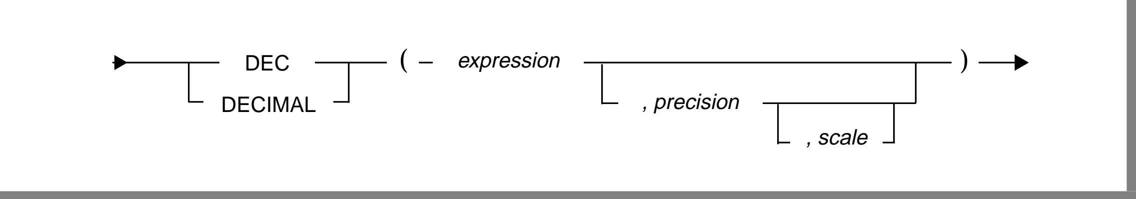
```
select distinct prod_name, store_name,
    ceil(dollars/quantity) as price
from product natural join sales natural join period
    natural join store
where year = 2000
    and month = 'MAR'
    and prod_name like 'Xalapa Lapa%'

prod_namestore_nameprice

Xalapa LapaBeaches Brew9
Xalapa LapaBeans of Boston9
Xalapa LapaCoffee Brewers9
Xalapa LapaCoffee Connection9
Xalapa LapaCupertino Coffee Supply9
Xalapa LapaEast Coast Roast9
Xalapa LapaInstant Coffee7
Xalapa LapaInstant Coffee8
Xalapa LapaInstant Coffee9
Xalapa LapaJava Judy's9
Xalapa LapaMiami Espresso9
Xalapa LapaMinnesota Roaster8
Xalapa LapaMinnesota Roaster9
Xalapa LapaMoon Pennies9
Xalapa LapaMoroccan Moods9
Xalapa LapaMoulin Rouge Roasting9
Xalapa LapaOlympic Coffee Company9
Xalapa LapaRoasters, Los Gatos9
Xalapa LapaSan Jose Roasting Company9
Xalapa LapaTexas Teahouse9
Xalapa LapaThe Coffee Club9
```

DEC

The DEC function converts a specified value to a DECIMAL value.

Syntax

The following syntax diagram shows how to construct an expression with the DEC function:



expression The expression can be a numeric or character datatype but must represent a numeric value. For example:

`'19.2'`—valid

`'RAJ'`—not valid

precision Specifies the precision of the resulting DECIMAL value. This value must be between 1 and 38 inclusive. The default value is 9.

scale Specifies the scale of the resulting DECIMAL value. This argument must be a value between 0 and the value specified in *precision*. The default value is 0.

Result

This function returns a value with the datatype DECIMAL(*precision*, *scale*). If the argument is NULL, the function returns NULL. If the argument is a CHARACTER or VARCHAR expression that represents a number, the argument is converted to a DECIMAL datatype; otherwise, the server returns an error message that the expression must represent a number.

If the value represented by *expression* is too large to express as DECIMAL(*precision*, *scale*) without truncation of significant digits (digits to the left of the decimal point), the server issues an "out of range" error message. If digits to the right of the decimal point must be truncated to fit into the specified DECIMAL(*precision*, *scale*) datatype, no error message is issued.

Example

The expression

```
DEC('40E3',7,2)
```

returns the decimal number 40000.00.

For an example of the DEC function in a complete SELECT statement, refer to the *SQL Self-Study Guide*.

EXP

The EXP function returns the exponential value of the function argument; that is, EXP(x) returns the value y such that $e^x = y$. ($e = 2.71828183$.)

Syntax

The following syntax diagram shows how to construct an expression with the EXP function:

expression The expression can be any datatype but must represent a numeric value. For example:

`'19.2'`—valid

`-8.634E2`—valid

`'RAJ'`—not valid

Result

This function returns a DOUBLE PRECISION value.

If expression is of the datatype TINYINT, SMALLINT, INTEGER, REAL, FLOAT, DECIMAL, or NUMERIC, it is converted to a DOUBLE PRECISION value.

If expression is a CHARACTER or VARCHAR expression representing a numeric value, the EXP function converts this value to a double precision floating point, calculates the exponential value, and returns the exponential value as a double precision. If the character string does not represent a numeric value, the server returns an error message. If the argument is NULL, the function returns NULL.

If the result overflows the DOUBLE PRECISION datatype, the server returns an error message.

Example

The expression

```
EXP(2.0)
```

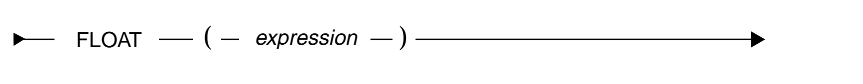
returns *e* raised to the power 2.0, which is 7.389056; *e* = 2.71828183.

FLOAT

The FLOAT function converts a specified value into a double-precision floating-point value.

Syntax

The following syntax diagram shows how to construct an expression with the FLOAT function:



expression The argument can be a numeric or character datatype but must represent a numeric value. For example:

`'19.2'`—valid

`'RAJ'`—not valid

Result

The function returns a DOUBLE PRECISION value. If the argument is NULL, the function returns NULL.

If the argument is of datatype CHARACTER or VARCHAR and represents a number, it is converted to a floating-point number; otherwise, the server returns an error message.

If the argument is of datatype TINYINT, SMALLINT, INTEGER, REAL, FLOAT, DECIMAL, or NUMERIC, it is converted to a DOUBLE PRECISION value.

If the result overflows the DOUBLE PRECISION datatype, the server returns an error message.

Tip: *How floating-point numbers are displayed in result sets depends on the formatting capabilities of the client tool; for example, RISQL Reporter users can use the SET COLUMN column_name FORMAT command to set the display format to EXPONENTIAL.*

Example

The expression

```
float('-19698939.67')
```

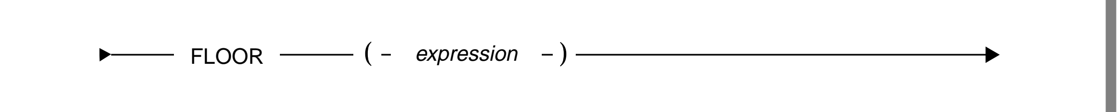
returns the floating-point number –1.969893967E+07 (assuming that an exponential display format is used).

FLOOR

The FLOOR function returns the nearest integral value less than or equal to the value of the function argument.

Syntax

The following syntax diagram shows how to construct an expression with the FLOOR function:



expression The argument can be a numeric or character datatype but must represent a numeric value. For example:

`'19.2'`—valid

`'RAJ'`—not valid

Result

If the argument is a REAL, FLOAT, DOUBLE PRECISION, DECIMAL, or NUMERIC value, the FLOOR function calculates and returns the floor value, preserving the input datatype.

If the argument is an INTEGER, SMALLINT, or TINYINT, the FLOOR function returns this value immediately, preserving the input datatype.

If the argument is a character string representing a numeric value, the FLOOR function converts this value to a double-precision floating point, calculates the floor value, and returns the floor value as a double-precision. If the character string does not represent a numeric value, the FLOOR function returns an error.

If the argument is NULL, the FLOOR function returns NULL.

Example

The following query computes a price-per-item value, rounded down to the nearest integral value:

```
select prod_name, store_name,
    floor(dollars/quantity) as price
from product natural join sales natural join period
    natural join store
where date = '03-31-2000'
    and prod_name like 'Xalapa Lapa%'
PROD_NAME            STORE_NAME                    PRICE
Xalapa Lapa          Instant Coffee                8
Xalapa Lapa          Moroccan Moods                9
```

INT

The INT function converts a specified numeric string into an integer value.

Syntax

The following syntax diagram shows how to construct an expression with the INT function:

expression The argument can be a numeric or character datatype but must represent a numeric value. For example:

`'19.2'`—valid

`'RAJ'`—not valid

Result

The function returns an INTEGER value. If the argument is NULL, the function returns NULL. If the length of the result exceeds the length of the INTEGER datatype, the server returns an "out of range" error message.

If the argument is a character expression that represents a number, the argument is converted to an integer; otherwise, the server returns an error message that the expression must represent a number.

If the argument is a TINYINT or SMALLINT value, the function returns an integer padded with leading blanks.

If the argument is a REAL, FLOAT, DOUBLE PRECISION, DECIMAL, or NUMERIC value, its scale is truncated. If the length of the truncated number exceeds the length of the INTEGER datatype, the server returns an "out of range" error message.

Examples

The expression

```
int('197.665')
```

returns the integer 197.

The following query removes the decimal places from the Dollars values:

```
select int(dollars) as no_cents
from sales join product using (classkey, prodkey)
    join period using (perkey)
where prod_name like 'Vera%'
    and month = 'FEB'
    and year = 2000
NO_CENTS
88
96
72
48
144
48
64
...
```

The following query is an example of an INT function nested inside a RISQL display function (CUME):

```
select cume(int(price)) as price, line_item
from line_items
where order_no = 3600
order by line_item
PRICE                LINE_ITEM
180                  1
480                  2
720                  3
960                  4
1200                 5
```

You can also nest a RISQL display function inside a scalar function:

```
select int(cume(price)) as price, line_item
from line_items
where order_no = 3600
order by line_item
```

LN

The LN function returns the natural logarithm of the function argument; that is, LN(*y*) returns the value of *x* such that $e^x = y$. (e=2.71828183.)

Syntax

The following syntax diagram shows how to construct an expression with the LN function:

expression The expression can be any datatype but must represent a numeric value. For example:

`'19.2'`—valid

`-8.634E2`—valid

`'RAJ'`—not valid

Result

The function returns a DOUBLE PRECISION value. If the argument is NULL, the function returns NULL. If the length of the result exceeds the length of the DOUBLE PRECISION datatype, the server returns an "out of range" error message.

If the argument is a TINYINT, SMALLINT, INTEGER, REAL, FLOAT, DECIMAL, or NUMERIC value, it is converted to a DOUBLE PRECISION value.

If the argument is negative or zero, the behavior is dictated by the ARITHIGNORE or ARITHABORT setting, similar to divide-by-zero processing. If ARITHIGNORE is ON (or ARITHABORT is OFF), the server returns NULL for computing square root on negative arguments. If ARITHIGNORE is OFF (or ARITHABORT is ON), the server produces an error message. Refer to "SET ARITHIGNORE, ARITHABORT" on page 9-7 for more information about the ARITHIGNORE and ARITHABORT settings.

If the argument is a character expression that represents a non-negative number, the argument is converted to a double precision floating-point value; otherwise, the server returns an error message.

Examples

The expression

```
ln(25.0)
```

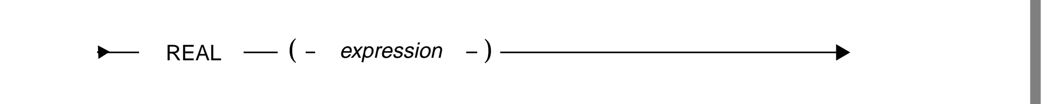
returns 3.22. ($e^{3.22} = 25$; $e = 2.71828183$.)

REAL

The REAL function converts a specified value into a REAL datatype.

Syntax

The following syntax diagram shows how to construct an expression with the REAL function:



expression The argument can be a numeric or character datatype but must represent a numeric value. For example:

`'19.2'`—valid

`'RAJ'`—not valid

Result

The function returns a REAL value. If the argument is NULL, the function returns NULL.

If the argument is of datatype CHARACTER or VARCHAR and represents a number, it is converted to a REAL number; otherwise, the server returns an error message.

If the argument is of datatype TINYINT, SMALLINT, INTEGER, FLOAT, REAL, DECIMAL, or NUMERIC, it is converted to a REAL value.

If the result overflows the REAL datatype, the server returns an error message.

Example

The expression

```
real('-19698939.67')
```

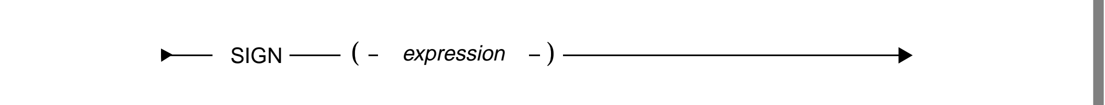
returns the following floating-point number –1.9698940E+07 (assuming that an exponential display format is used, as discussed on page 5-22).

SIGN

The SIGN function returns the sign (+ or –) of an integer value.

Syntax

The following syntax diagram shows how to construct an expression with the SIGN function:



expression The argument can be a numeric or character datatype but must represent a numeric value. For example:

`'19.2'`—valid

`'RAJ'`—not valid

Result

The SIGN function calculates the sign of the expression, and returns 1 for a positive value, –1 for a negative value, and 0 for zero. The return value is always an INT datatype.

If the argument is a CHARACTER or VARCHAR expression that does not represent a number, the SIGN function returns an error.

If the argument is NULL, the SIGN function returns a NULL result.

Example

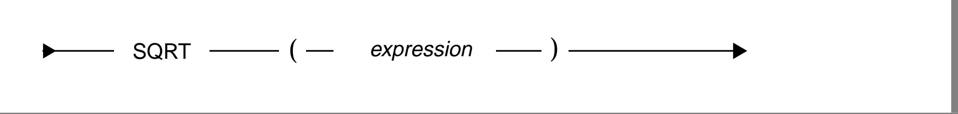
```
select numvalue, sign(numvalue) from numtable;
numvalue        sign(numvalue)
22.89           1
-90.03          -1
0               0
```

SQRT

The SQRT function returns the square root of the function argument.

Syntax

The following syntax diagram shows how to construct an expression with the SQRT function:



expression The expression can be any datatype but must represent a numeric value. For example:

`'19.2'`—valid

`-8.634E2`—valid

`'RAJ'`—not valid

Result

The function returns a DOUBLE PRECISION value. If the argument is NULL, the function returns NULL. If the length of the result exceeds the length of the DOUBLE PRECISION datatype, the server returns an "out of range" error message.

If the argument is a TINYINT, SMALLINT, INTEGER, REAL, FLOAT, DECIMAL, or NUMERIC value, it is converted to a DOUBLE PRECISION value.

If the argument is negative, the behavior is dictated by the ARITHIGNORE or ARITHABORT setting, similar to divide-by-zero processing. If ARITHIGNORE is ON (or ARITHABORT is OFF), the server returns NULL for computing square root on negative arguments. If ARITHIGNORE is OFF (or ARITHABORT is ON), the server produces an error message. Refer to Chapter 9 for more information about the ARITHIGNORE and ARITHABORT settings.

If the argument is a character expression that represents a numeric value, the SQRT function converts the value to a double precision floating-point value, calculates the square-root value, and returns the square-root value as a double-precision value. If the character string does not represent a numeric value, the server returns an error message.

Examples

The expression

```
sqrt(25.0)
```

returns 5.000.

Macros for Statistical Functions

The following macros perform statistical functions. Most of them are built using the EXP, LN, and SQRT scalar functions:

- Power
- Log
- Log10
- Standard deviation
  - A population of values
  - *Sample* estimate of the standard deviation
- Variance
  - A population of values
  - Unbiased *sample* estimate of the population variance

Power

The following macro calculates *x* raised to the *y*th power:

```
create macro power(x, y) as
    (exp( (y) * ln((x)) ));
```

Log

The following macro calculates the logarithm of a number *x* to base *y*:

```
create macro log(x, y) as
    ( ln ((x)) / ln((y)) );
```

Log10

The following macro calculates the logarithm of a number *x* to base 10:

```
create macro log10(x) as
    ( ln((x)) / ln(10) );
```

Standard Deviation of a Population

The standard deviation of a population of values is calculated as:

$$\sigma = [\sum(x_i - \mu)^2/N]^{1/2}$$

where μ is the population mean and N is the population size.

The following macro calculates the standard deviation for the population of values defined by *col_name*:

```
create macro stddev(col_name) as
    sqrt( abs(( sum((col_name) * (col_name)) -
    ((sum((col_name)) * sum((col_name))) /
    count((col_name))) ) / count((col_name))) );
```

Sample Estimate of the Population Standard Deviation

The sample estimate of the population standard deviation is computed as:

$$\sigma = [\sum(x_i - xbar)^2/n\text{-}1]^{1/2}$$

where *xbar* is the sample mean and *n* is the sample size.

The following macro calculates the sample estimate of the population standard deviation for values defined by *col_name*.

```
create macro population_stddev(col_name) as
    sqrt( abs(((count((col_name)) *
    sum((col_name) * (col_name))) -
    (sum((col_name)) * sum((col_name)))) /
    (count((col_name)) * (count((col_name)) - 1))) );
```

Variance

The variance of a population of values is computed as:

$$\sigma^2 = \sum(x_i - \mu)^2/N$$

where μ is the population mean and N is the population size.

The following macro calculates the variance of the population of values defined by *col_name*.

```
create macro variance(col_name) as
    ( (sum((col_name) * (col_name)) -
    ((sum((col_name)) * sum((col_name))) /
    count((col_name)))) / count((col_name)) );
```

Unbiased Sample Estimate of the Population Variance

The unbiased sample estimate of the population variance is computed as:

$$\sigma^2 = [\sum(x_i - xbar)^2/n\text{-}1]^{1/2}$$

where *xbar* is the sample mean and *n* is the sample size.

The following macro calculates the unbiased sample estimate of the population variance for the population defined by *col_name*:

```
create macro population_variance(col_name) as
    ( ((count((col_name)) * sum((col_name) * (col_name))) -
    (sum((col_name)) * sum((col_name)))) /
    (count((col_name)) * (count ((col_name)) - 1)) );
```

Example

The following query uses the *stddev* macro to calculate the standard deviation on the average sales of Earl Grey tea in the Western region during the first quarter of 1999.

```
select store_name, sum(dollars) as Sales,
avg(dollars) as AvgSales,
stddev(dollars) as S_Dev
from period natural join sales
            natural join store
            natural join market
                natural join product
```

```
where region = 'West'
and prod_name = 'Earl Grey'
and qtr='Q1_99'
and year = 1999
group by qtr, store_name
order by qtr, store_name;

STORE_NAME                 SALES      AVGSALES     S_DEV
Beaches Brew              1159.50     39.98275862    15.35
Cupertino Coffee Supply    628.50     28.56818181    15.93
Instant Coffee             717.50     42.20588235    14.92
Java Judy's                595.50     35.02941176    18.34
Roasters, Los Gatos        911.00     41.40909090    18.13
San Jose Roasting Company  395.00     28.21428571    15.64
```

String Scalar Functions

With the exception of the STRING function, which converts values with numeric datatypes into strings, the string scalar functions operate on character strings. Both single- and multibyte character processing is supported.

The following table lists the string scalar functions:

String Function	Description
CONCAT	Concatenates character strings.
LENGTH	Computes the number of characters in a string.
LENGTHB	Computes the number of bytes in a string.
LOWER	Converts character strings to lowercase.
LTRIM	Trims leading blanks.
RTRIM	Trims trailing blanks.
STRING	Converts numeric datatypes to CHARACTER.
SUBSTR	Extracts substrings.

(1 of 2)

String Function	Description
Substrb	Extracts substrings in bytes from a character string.
TRIM	Trims leading and trailing blanks.
UPPER	Converts character strings to uppercase.

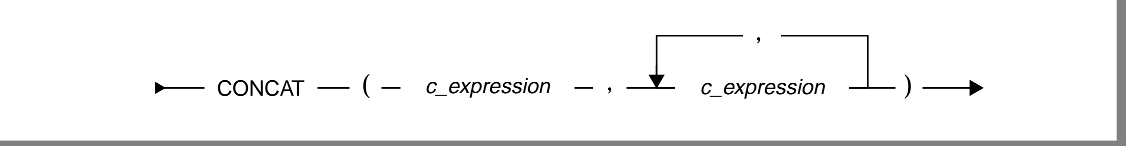
(2 of 2)

CONCAT

The CONCAT function concatenates character strings.

Syntax

The following syntax diagram shows how to construct an expression with the CONCAT function:



c_expressions The *c_expressions* must reference only character datatypes. The function accepts an arbitrary number of single- or multibyte arguments.

Result

If each argument is a non-null character expression, the function concatenates the arguments and returns the concatenated string of characters; otherwise, it returns NULL.

The maximum length of the returned character string is the sum of the maximum byte lengths of its arguments but cannot exceed 1,024 bytes. The actual length of the returned string is equal to the actual lengths of the arguments. The SUBSTR, TRIM, LTRIM, and RTRIM functions can reduce the maximum length of a character string.

Example

The following query concatenates character strings from four different columns:

```
select concat(hq_city, ' ', hq_state, ' ', district, ' ',
    region)
from market
where region = 'West' or region = 'North'
New York NYNew York North
Philadelphia PANew York North
Boston MABoston North
Hartford CTBoston North
San Jose CASan Francisco West
San Francisco CASan Francisco West
...
```

LENGTH

The LENGTH function computes the number of characters in a string.

Syntax

The following syntax diagram shows how to construct an expression with the LENGTH function:

```
►── LENGTH ── ( ─ c_expression ─ ) ──────────►
```

c_expression The *c_expression* must be a character datatype.

Result

If the argument is not NULL, the function returns an integer result specifying the number of characters in the string; otherwise, the result is NULL.

The length of a VARCHAR column includes any trailing blanks.

The length of a CHARACTER column is the length that was defined in the CREATE TABLE statement. Use the TRIM, LTRIM, and RTRIM functions to find out the length of a string within a character column.

The LENGTH function returns the number of characters, not the number of bytes, in a string.

Example

The following query returns the length in characters of the district names in the Market table. Note that because no function such as TRIM or SUBSTR is used to reduce the maximum length of the character string, the query returns the maximum value of the character string.

```
select distinct district, length(district) as char_len
    from market;

DISTRICT             CHAR_LEN
Atlanta                    20
Boston                     20
Chicago                    20
Los Angeles                20
Minneapolis                20
New Orleans                20
New York                   20
San Francisco              20
```

The following query returns the longest district name in the Market table. The TRIM function is used so the actual number of characters in each district name is considered, rather than the maximum length of the character string:

```
select distinct district from market
    where length(trim(district)) =
    (select max(length(trim(district))) from market)
DISTRICT
San Francisco
```

LENGTHB

The LENGTHB function computes the number of bytes in a string.

Syntax

The following syntax diagram shows how to construct an expression with the LENGTH function:

```
►── LENGTHB ── ( ─ c_expression ─ ) ──────────►
```

c_expression The *c_expression* must be a character datatype.

Result

If the argument is not NULL, the function returns an integer result specifying the number of bytes in the c_expression. If the argument is NULL, the result is NULL.

The LENGTHB function returns the number of bytes, not the number of characters, in a string. To return the number of characters in a string, use the LENGTH function.

Example

The following query, performed using a variable-byte-length, multibyte character set, returns the number of bytes in a string where the numbers 0 to 9 are single-byte character length, characters *Xx* and *Yy* are double-byte character length, and character *Zzz* is three bytes long.

```
select lengthb ('12XxYyZzz3') as 'length in bytes' from test;
10
```

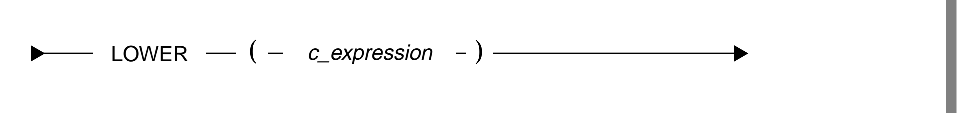
In comparison, the LENGTH function returns the value *6,* the number of characters in the string.

LOWER

The LOWER function converts a character string to lowercase.

Syntax

The following syntax diagram shows how to construct an expression with the LOWER function:



c_expression The *c_expression* must be a character datatype.

Result

If the argument is not NULL, the function converts the character string to lowercase; otherwise, the result is NULL.

Examples

The following query returns the cities in the Market table beginning with the letter *M.* The result table displays the city names with mixed-case letters (as they were loaded into the table).

```
select hq_city
from market
where hq_city like 'M%'
HQ_CITY
-----------
Miami
Minneapolis
Milwaukee
```

The following query uses the LOWER function to display the city names in lowercase letters:

```
select lower(hq_city) as lower_city
from market
where hq_city like 'M%'
LOWER_CITY
-----------
miami
minneapolis
milwaukee
```

LTRIM

The LTRIM function repositions leading blanks in a character string. Multibyte blanks are processed in the same way as single-byte blanks.

Syntax

The following syntax diagram shows how to construct an expression with the LTRIM function:

```
►──── LTRIM ── ( – c_expression – ) ──────────────►
```

c_expression The *c_expression* must be a character datatype.

Result

If the argument is not NULL, the function removes leading blanks from the character string; otherwise, the result is NULL.

The maximum length of the result is the maximum length of its argument.

Usage Note

CHAR columns and output data always have a fixed length and they are padded with blanks to fill out that length. You cannot use LTRIM to reduce the length of a CHAR column or an output column.

Examples

The expression

```
ltrim(market)
```

trims leading blanks from the Market column.

The following query removes leading blanks from the Hq_City and District columns:

```
select concat(ltrim(hq_city), ltrim(district)) as mkt_district
from market
where region in ('South', 'North')
 MKT_DISTRICT
 Atlanta              Atlanta
 Miami                Atlanta
 New Orleans          New Orleans
 Houston              New Orleans
 New York             New York
 Philadelphia         New York
 Boston               Boston
 Hartford             Boston
```

RTRIM

The RTRIM function repositions trailing blanks from a character string. Multibyte blanks are processed in the same way as single-byte blanks.

Syntax

The following syntax diagram shows how to construct an expression with the RTRIM function:

```
▶── RTRIM ── ( ─ c_expression ─ ) ──────▶
```

c_expression The *c_expression* must be a character datatype.

Result

If the argument is not NULL, the function removes trailing blanks from the character string; otherwise, the result is NULL.

The maximum length of the result is the maximum length of its argument.

Usage Note

CHAR columns and output data always have a fixed length and they are padded with blanks to fill out that length. You cannot use RTRIM to reduce the length of a CHAR column or an output column.

Examples

The expression

```
rtrim(market)
```

trims trailing blanks from the Market column.

The following query removes trailing blanks from the Hq_City column. It also concatenates the Hq_City and District columns, using a comma and a space to separate them:

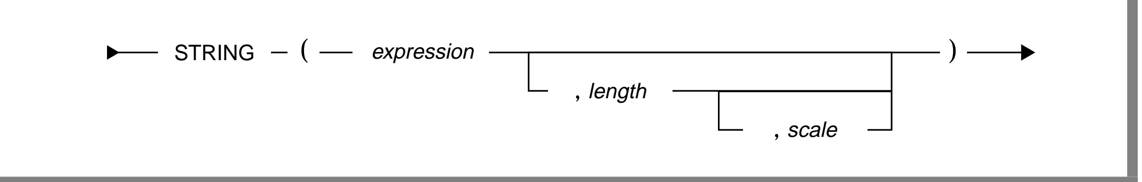
```
select concat(rtrim(hq_city), ', ' , district) as mkt_district
from market
where region in ('South', 'North')
MKT_DISTRICT
Atlanta, Atlanta
Miami, Atlanta
New Orleans, New Orleans
Houston, New Orleans
New York, New York
Philadelphia, New York
Boston, Boston
Hartford, Boston
```

STRING

The STRING function converts numeric or datetime values to character strings.

Syntax

The following syntax diagram shows how to construct an expression with the STRING function:



expression Must be numeric or datetime. If *expression* is not NULL, the function returns a character string; otherwise, it returns NULL.

length Determines the maximum number of characters returned by the function and must not be NULL. The argument can be an integer constant, an integer expression, or omitted. If this argument is omitted, the number of characters returned depends on the datatype of *expression*, as summarized in the following table:

Datatype	Default Number of Characters Returned
TINYINT	4 characters
SMALLINT	6 characters
INTEGER	11 characters
DECIMAL, NUMERIC	(precision + 2) characters *or* (precision + 3) characters when value < 0 and scale = precision

(1 of 2)

Datatype	Default Number of Characters Returned
REAL, FLOAT, DOUBLE PRECISION	23 characters
DATE	10 characters (8 digits, 2 separators)
TIME	15 characters (12 digits, 3 separators)
TIMESTAMP	26 characters (20 digits, 5 separators, 1 space)

(2 of 2)

If the number of digits needed to represent a value is greater than the number of characters returned by the STRING function, the result is truncated (for example, if the default number of characters returned for the given datatype is insufficient, or if the *length* argument specified was too small).

In particular, consider the case of FLOAT expressions. Because the STRING function does not use scientific notation, an expression such as 1E35 will be truncated when no *length* argument is specified, since the default number of characters returned for FLOAT expressions is only 23.

If a numeric expression is constructed from more than one column, the result is usually promoted to a floating-point number. If the *length* argument is an expression, the truncated integral value of the expression is used on a row-by-row basis to format the first argument. If *length* is too short, the result will be truncated without an error or warning message.

Tip: *The STRING function counts the space immediately to the left of a numeric expression, which is a minus sign (–), a plus sign (+), or a blank space, as a character. Therefore, the expression*

```
string(1234, 4)
```

evaluates to 123, not 1234.

scale Determines the number of digits to the right of the decimal point for numeric datatypes and fractional seconds for time and timestamp datatypes. The *scale* value must not be NULL and must be less than or equal to (*length* – 3). This value is ignored for date expressions.

Examples

- The expression

  ```
  string(dollars/quantity, 10,2)
  ```

  returns a string no longer than 10 characters with 2 decimal points.
- The expression

  ```
  string (current_date)
  ```

  returns a string reflecting the current date, similar to 1999-11-07.
- The expression

  ```
  string (current_date, 4)
  ```

  returns only the current year portion of the date—2000, for example.
- The expression

  ```
  string (current_timestamp)
  ```

  returns a string reflecting the current timestamp, similar to

  ```
  '2000-11-07 14:50:40.710474'
  ```
- The expression

```
string (current_timestamp, 26, 4)
```

  returns a string reflecting the current timestamp with four fractional-second digits, similar to

  ```
  '2000-11-07 14:50:40.7104'
  ```
- The following query uses the STRING function to produce Price column values with only two decimal places; otherwise, the server would return long-numeric Price values.

  ```
  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 = 2000
  group by prod_name
  ```

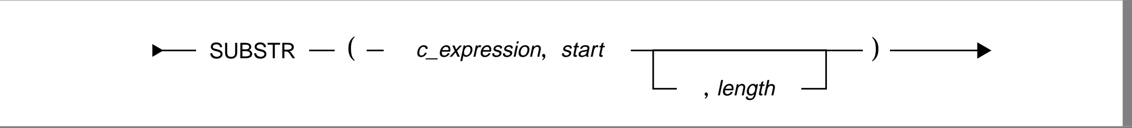
For more examples of the STRING function inside complete SELECT statements, refer to the *SQL Self-Study Guide*.

SUBSTR

The SUBSTR function extracts a substring from a character string.

Syntax

The following syntax diagram shows how to construct an expression with the SUBSTR function:



c_expression	The *c_expression* must be a character datatype.
start	The *start* argument is an integer expression that specifies the starting character position of a substring in the first argument; the first position is 1.
length	The *length* argument specifies the number of characters to be extracted. If specified, *length* can be a non-null integer expression or an integer constant. If *length* is an integer constant, it can be no larger than the length of the first argument.

Result

If the first argument is not NULL, the function returns the substring that begins at position *start* and continues for *length* characters; if *length* is not specified, SUBSTR returns a substring from *start* to the end of *c_expression*.

If the first argument is NULL, the function returns NULL.

If the *start* argument is less than one or is NULL, the server returns an error message. If *start* is greater than the length of the first argument, a string of length zero is returned.

If *length* is greater than (length of *c_expression* –*start*)+1, SUBSTR returns only (length of *c_expression* –*start*)+1 characters .

Handling Trailing Blanks

When SUBSTR is used with a client, output can be inconsistent because of the way in which the system handles trailing blanks. The current native ODBC connectivity software truncates trailing blanks, whereas earlier connectivity software preserved trailing blanks.

The OPTION SERVER_TRIM_TRAILING_SPACES *rbw.config* option corrects this problem. This setting specifies whether the server should trim trailing spaces from the results of a SUBSTR operation before sending data to the client application. The default value is `YES`.

Examples

The expression

```
substr(market, 5, 10)
```

extracts a ten-character substring that begins at character position 5.

The following query extracts substrings that are 8 characters long, starting from position 1:

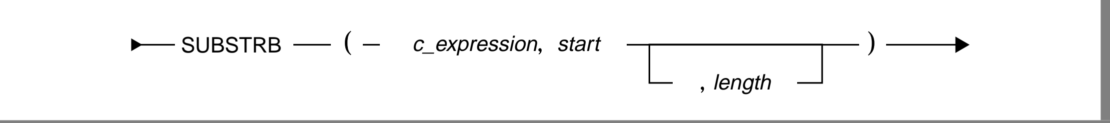
```
select classkey as class_no,
    substr(class_type, 1, 8) as type
from class
CLASS_NO    TYPE
           1 Bulk_bea
           2 Bulk_tea
           3 Bulk_spi
           4 Pkg_coff
           5 Pkg_tea
           6 Pkg_spic
           7 Hardware
           8 Gifts
          12 Clothing
```

SUBSTRB

The SUBSTRB function extracts a substring in bytes from a character string.

Syntax

The following syntax diagram shows how to construct an expression with the SUBSTRB function:



c_expression The *c_expression* must be a character datatype.

start The *start* argument is an integer expression that specifies the starting byte position of a substring in the first argument; the first position is 1.

length The *length* argument specifies the number of bytes to be extracted. If specified, *length* can be a non-null integer expression or an integer constant. If *length* is an integer constant, it can be no larger than the length of the *c_expression*.

Result

If the first argument is not NULL, the function returns the substring that begins at position *start* and continues for *length* bytes; if *length* is not specified, SUBSTRB returns a substring from *start* to the end of *c_expression*.

Warning: *Because SUBSTRB returns bytes and not characters, you can provide a start or length argument that specifies a position in the middle of a multibyte character. If this occurs, SUBSTRB returns results that include a single-byte blank.*

If the first argument is NULL, the function returns NULL.

If the *start* argument is less than one or is NULL, the server returns an error message. If *start* is greater than the length of the first argument, a string of length zero (0) is returned.

If *length* is greater than (length of *c_expression –start*)+1, SUBSTRB returns only (length of *c_expression –start*)+1 bytes .

Example

The following query, performed using a variable-byte-length, multibyte character set, returns the number of bytes in a string where the numbers 0 to 9 are single-byte character length, characters *Xx* and *Yy* are double-byte character length, and character *Zzz* is three bytes long.

```
select substrb ('12XxYyZzz3',3,4) from test;
XxYy
```

For comparison, the SUBSTR function returns the following result:

```
select substr ('12XxYyZzz3',3,4) from test;
XxYyZzz3
```

The following query shows SUBSTRB including a *length* argument that specifies a position in the middle of a multibyte character:

```
select substrb ('12XxYyZzz3',3,3) from test;
Xx<blank>
```

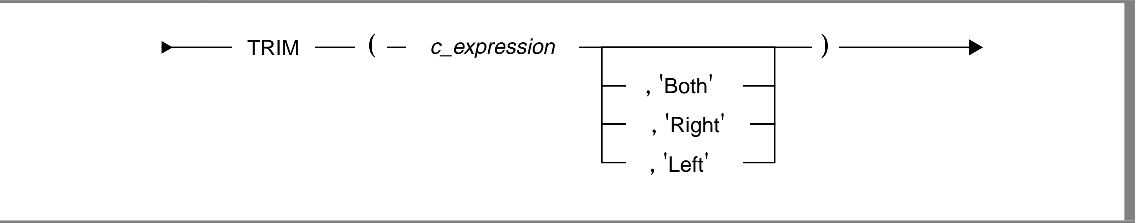
In this case, a single-byte blank is returned in place of the first byte of the multibyte character *Yy*.

TRIM

The TRIM function repositions leading and trailing blanks in a character string. Multibyte blanks are processed in the same way as single-byte blanks.

Syntax

The following syntax diagram shows how to construct an expression with the TRIM function:



c_expression	The first argument must be a character datatype.
Both, Right, Left	The second argument is a character literal that can be written in uppercase, lowercase, or a mixed case: The single quote characters (') must be included. *Both* removes leading and trailing blanks. *Right* removes trailing blanks. *Left* removes leading blanks. The default is *Both*.

Result

If the first argument is not NULL, the function removes leading and trailing blanks from the character string; otherwise, the result is NULL. The maximum length of the result is the maximum length of the first argument.

Usage Note

CHAR columns and output data always have a fixed length and they are padded with blanks to fill out that length. You cannot use TRIM to reduce the length of a CHAR column or an output column.

Examples

The expression

```
trim(market, 'right')
```

repositions trailing blanks in character strings in the Market column. (The TRIM function with the *right* argument is equivalent to the RTRIM function.)

The following query repositions leading and trailing blanks in the values in the City column:

```
select trim(hq_city) as city
from market
```

UPPER

The UPPER function converts a character string to uppercase.

Syntax

The following syntax diagram shows how to construct an expression with the UPPER function:

c_expression The *c_expression* must be a character datatype.

Result

If the argument is not NULL, the function converts the character string to uppercase; otherwise, the result is NULL.

Examples

The following query returns the cities in the Market table beginning with the letter *M*. The result table displays the city names with mixed-case letters, as they were entered into the table:

```
select hq_city as city
from market
where hq_city like 'M%'
CITY
-----------
Miami
Minneapolis
Milwaukee
```

The following query uses the UPPER function to display the city names in uppercase letters:

```
select upper(hq_city) as city
from market
where hq_city like 'M%'
UPPER_CITY
-----------
MIAMI
MINNEAPOLIS
MILWAUKEE
```

Datetime Scalar Functions

The scalar functions in this section operate on datetime expressions. The datetime scalar functions are listed in the table below:

Datetime Function	Description
CURRENT_DATE	Returns current date.
CURRENT_TIME	Returns current time.
CURRENT_TIMESTAMP	Returns current date and time.
DATE	Converts TIMESTAMP or character string to DATE.
DATEADD	Adds interval to datetime value.
DATEDIFF	Computes difference between two datetime values.
DATENAME	Extracts datepart component from datetime value as character string.
EXTRACT	Extracts datepart component from datetime value as INTEGER.
TIME	Converts TIMESTAMP or character string to TIME.
TIMESTAMP	Converts DATE or TIME or character string to TIMESTAMP.

For additional examples of the DATEADD, DATEDIFF, and EXTRACT functions, see the *SQL Self-Study Guide*.

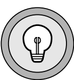
Dateparts for Datetime Scalar Functions

Some scalar functions operate on the individual subfields, or dateparts, that comprise the datetime datatypes. These dateparts can be extracted from a datetime value with the EXTRACT function.

The datepart arguments are not translated; they are the same for all locales. However, the output of the DATENAME and EXTRACT functions is localized.

The following table defines these dateparts:

Datepart	Abbreviation	Range of Return Values
YEAR	yy	1 to 9999
QUARTER	qq	1 to 4 (determined from the month value)
MONTH	mm	1 to 12 (or a localized month name when used with the DATENAME function)
DAYOFYEAR	dy	1 to 366
DAY	dd	1 to 31
WEEK	wk	1 to 53. All weeks except the first week begin on the first day of the week (as specified by the formatting rules for the territory defined in the server locale). The first (and last) week might be a fractional week, depending on the day of the week for January 1.
WEEKDAY	dw	1 to 7, where 1 is the first day of the week (depending on the territory defined in the server locale). When used with the DATENAME function, this datepart returns a localized day name.
HOUR	hh	0 to 23
MINUTE	mi	0 to 59
SECOND	ss	0 to 59
MILLISECOND	ms	0 to 999
MICROSECOND	us	0 to 999999



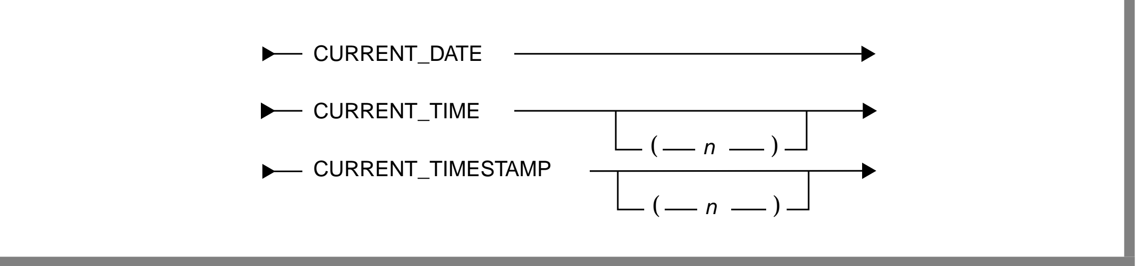
Tip: *For those client tools that do not use the ANSI SQL-92 EXTRACT function, the nonstandard DATEPART scalar function is supported. For more information about DATEPART, refer to Appendix C.*

CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP

The CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP functions return the current date, current time, and current timestamp values.

Syntax

The following syntax diagram shows how to construct a datetime expression with these functions:



n — The *n* argument is an integer that specifies the precision of the fractional second component returned in the TIME and TIMESTAMP values.

If *n* is not specified, the default precision is as follows:

- CURRENT_TIME(0)
- CURRENT_TIMESTAMP(6)

Result

The results of the CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP functions are constant for the duration of the query.

Example

The following example inserts the current date into the Date_Col column:

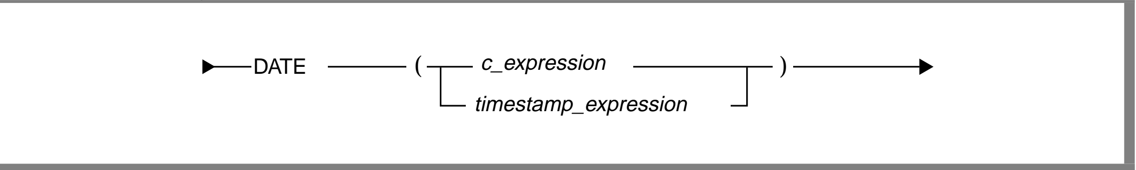
```
insert into table_1 (date_col)
values (CURRENT_DATE)
```

DATE

The DATE function creates a date value from a character string or a timestamp expression.

Syntax

The following syntax diagram shows how to construct a datetime expression with the DATE function:



c_expression, timestamp_expression	The argument can be either a character or a timestamp expression. (A date expression is also accepted, although clearly no conversion needs to be performed in this case.) The character expression must form a valid date value, as defined in "Datetime Literals" on page 2-12. Alternative date or timestamp formats are not allowed.

Result

The result is a date datatype.

Examples

The following example uses DATE to convert values from a character column in one table and insert them into the Date_Col column in another table:

```
insert into table_1 (date_col)
    select date (char_col) from table_2
```

The following example uses DATE to convert a timestamp value to a date value:

```
insert into table_1 (date_col)
    select date (timestamp_col) from table_2
```

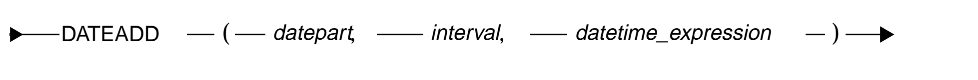
The default for the DATE format is YYYY-MM-DD. To display the date in a different format, use the EXTRACT function. For a specific example, refer to EXTRACT on page 5-63.

DATEADD

The DATEADD function adds an interval to a datetime value.

Syntax

The following syntax diagram shows how to construct a datetime expression with the DATEADD function:



datepart	The datepart argument specifies the datepart to which to add the interval; it must be a datepart or abbreviation defined in "Dateparts for Datetime Scalar Functions" on page 5-54.
interval	The *interval* argument must be an integer.
datetime_expression	The expression must be a DATE, TIME, or TIMESTAMP datatype.

Result

The function returns a result that is the same datetime datatype as that of *datetime_expression.*

Calculations that involve *day, dayofyear,* and *dayofweek* use units of days.

If month or quarter is the datepart being added and the resulting month does not have enough days, the date is set to the last day of the month.

If year is the datepart being added, the year is a leap year, and the result is February 29 in a non-leap year, the date is set to March 1 of that year.

Examples

The DATEADD function increments the month by 1 for each value in the Date_Col column of Table_1:

```
select dateadd(month, 1, date_col) as month_increment
from table_1
```

The date 1999-11-25 is incremented to 1999-12-25. The date 1999-12-25 is incremented to 2000-1-25.

In the next example, the DATEADD function decrements the month by 1 for each value in the Date_Col column of Table_1:

```
select dateadd(month, -1, date_col) as month_decrement
from table_1
```

DATEDIFF

The DATEDIFF function finds the difference between two datetime expressions.

Syntax

The following syntax diagram shows how to construct a datetime expression with the DATEDIFF function:

```
►─DATEDIFF ── ( ─ datepart, ─ datetime_expression, ── datetime_expression ── ) ─►
```

datepart	The datepart argument specifies the datepart on which to calculate the difference; it must be a datepart or abbreviation defined in “Dateparts for Datetime Scalar Functions” on page 5-54.
datetime_expression	The expressions must be DATE, TIME, or TIMESTAMP datatypes. The expressions do not need to be the same datatype, but they should both contain the specified datepart.

If either expression does not contain the specified datepart, a default time of midnight is used for missing time parts and a default date of 1900-01-01 is used for missing dateparts.

Result

The function returns an integer result in datepart units. Calculations that involve *day*, *dayofyear*, and *dayofweek* use units of days.

Example

For the following example, assume the Date_Col column in Table_1 has a date value of 1996-12-03:

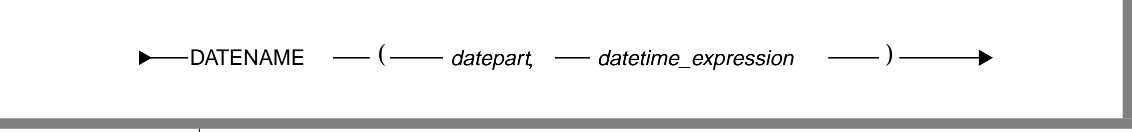
```
select current_date as today,
datediff(year, '01-01-2001', current_date) as till_next_mil
from period where date = '12-03-1998';
TODAY      TILL_NEXT
1999-09-0          2
```

DATENAME

The DATENAME function extracts the specified datepart component and returns its value as a character string.

Syntax

The following syntax diagram shows how to construct a datetime expression with the DATENAME function:



datepart	Specifies the datepart from which to extract the date component; it must be a datepart or abbreviation defined in "Dateparts for Datetime Scalar Functions" on page 5-54.
datetime_expression	Must be a DATE, TIME, or TIMESTAMP datatype.

Result

If a datetime expression does not contain the specified datepart, missing time parts default to midnight and missing dateparts default to 1900-01-01. These default values are also returned as character strings.

The output of the DATENAME function is localized; month and day names are displayed according to the language and territory specified by the server locale.

Examples

If the language defined by the server locale is English, for a date value of 1999-12-25, the DATENAME function returns *December* for the month:

```
select datename(mm, date)
from period
where date = '12-25-1999'
December
```

In a German database, the DATENAME function returns German month names and day names:

```
select distinct datename(mm, date)
from period
where qtr like 'Q4%'
Dezember
November
Oktober

select datename(dw, date)
from period
order by perkey
Samstag
Sonntag
Montag
...
```

For 1999-12-25, the DATENAME function returns 1999 for the year:

```
select datename(yy, date '1999-12-25')
from period
1999
...
```

For 1997-12-01, the DATENAME function returns 1 for the day:

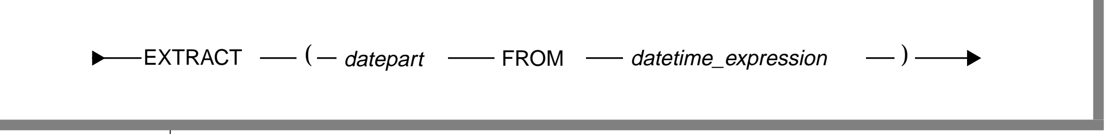
```
select datename(dd, date '1997-12-01')
from period
1
...
```

EXTRACT

The EXTRACT function extracts a specified datepart component from a datetime value as an integer value.

Syntax

The following syntax diagram shows how to construct a datetime expression with the EXTRACT function:



datepart	Specifies the datepart from which to extract the date component; it must be a datepart or abbreviation defined in "Dateparts for Datetime Scalar Functions" on page 5-54.
datetime_expression	Must be a DATE, TIME, or TIMESTAMP datatype.

Result

If the datetime expression does not contain the specified datepart, a default time of midnight is used for missing time elements and a default date of 1900-01-01 is used for missing date elements. These default values are also returned as integers.

The output of the EXTRACT function is localized, based on the formatting rules for the territory specified in the server locale. For example, extracting the weekday from a date (a number from 1 to 7) yields a different answer depending on the day on which the week starts.

If *datepart* is week, EXTRACT takes into account the day of the week on which the first day of the specified year falls. For example, if the first day of January is a Saturday, and the week starts on Sunday, then the second day is in week 2. If the week starts on Monday, the second day is in week 1, and so forth.

Examples

The EXTRACT function extracts the year from the date:

```
select extract (year from date_col) as year_1999
from table_1

1999
...
```

The EXTRACT function can be used to display the date in a different format:

```
select date,
concat (substr (string (extract(month from date_col)), 10,
2),
'/',
substr (string (extract (day,date_col)), 10, 2),
'/',
substr (string (extract (year from date_col)), 10, 2)) as
new_date
from period;

DATE                    NEW_DATE
1998-01-01              1/1/98
1998-01-02              1/2/98
1998-01-03              1/3/98
```

The slash (/) can be omitted or changed to a dash (-) as needed.

The EXTRACT function returns localized values for weekdays. For example, run against an English_UnitedStates database, the following query returns a value of 2:

```
select extract(dw from date)
from period
where date = '01-03-2000'
2
```

This result is obtained because the week begins on Sunday and January 3, 2000, is a Monday. For languages and locations whose weeks begin on Monday, such as German_Germany, the same query would return a value of1.

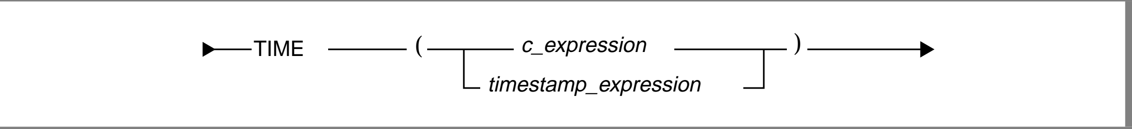
Tip: *For those client tools that do not use the ANSI SQL-92 EXTRACT function, the nonstandard DATEPART scalar function is supported. For more information about DATEPART, refer to Appendix C.*

TIME

The TIME function creates a time value from a character string or a timestamp datatype expression.

Syntax

The following syntax diagram shows how to construct a datetime expression with the TIME function:



c_expression, timestamp_expression	Specifies either a character expression or a timestamp expression. (A time expression is also accepted, although clearly no conversion needs to be performed in this case.) The character expression must form a valid time value as defined in "Datetime Literals" on page 2-12. Alternative time or timestamp formats are not allowed.

Result

The result is a time datatype.

Examples

The following example uses TIME to convert values from a character column in one table and then insert them into a TIME column in another table:

```
insert into table_1 (time_col)
    select time (char_col) from table_2
```

The following example uses TIME to convert a timestamp value to a time value:

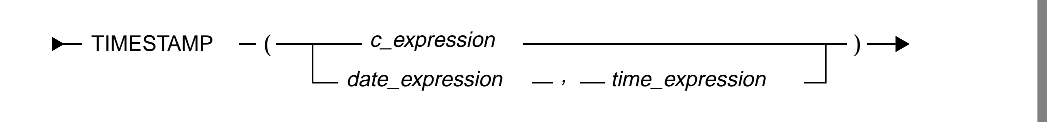
```
insert into table_1 (time_col)
    select time (timestamp_col) from table_2
```

TIMESTAMP

The TIMESTAMP function creates a timestamp value from a character string or from time and data values.

Syntax

The following syntax diagram shows how to construct a datetime expression with the TIMESTAMP function.



c_expression, date_expression, time_expression — A single argument must be a character expression. The character expression must form a valid timestamp value, as defined in "Datetime Literals" on page 2-12. Alternative timestamp formats are not allowed.

If there are two arguments, the first must be a date expression and the second must be a time expression separated by a comma (,). If either the date expression or the time expression is NULL, the resulting timestamp expression is also NULL.

Result

The result is a timestamp datatype.

Examples

The following example uses TIMESTAMP to convert values from a character column in one table and then insert them into a TIMESTAMP column in another table:

```
insert into table_1 (timestamp_col)
    select timestamp (char_col) from table_2
```

The following example uses TIMESTAMP to convert date and time values to a timestamp value:

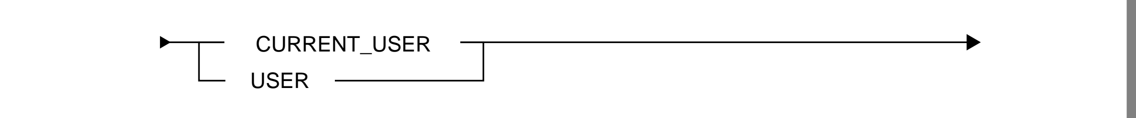
```
insert into table_1 (timestamp_col)
    select timestamp (date_col, time_col) from table_2
```

CURRENT_USER Function

There is one scalar function that is provided for informational and administrative purposes—the CURRENT_USER function. This function returns the database username (authorization ID) of the current user of the database. Database usernames are created with the GRANT CONNECT command, as described on page 8-198.

Syntax

The following syntax diagram shows how to construct an expression with the USER function:



Result

The system returns the name of the current user of the database. This function can be used to restrict or provide access based on a user's name.

Examples

The following query displays the tables created by the user currently connected to the database:

```
select *
from rbw_tables
where creator = CURRENT_USER
```

The following view displays only the tables created by individual users:

```
create view table_list
    as select * from rbw_tables
    where creator = CURRENT_USER
```

The creator of the view must then grant select privileges for public access on the view:

```
grant select on table_list to public
```

Users can then query the Table_List view to see the tables they have created. Only the tables created by the current user are displayed; system tables and tables to which the user has access but has not created are not displayed.

```
select * from table_list
```

Chapter

RISQL Display Functions

6

In This Chapter 6-3
CUME 6-5
MOVINGAVG 6-9
MOVINGSUM 6-12
NTILE 6-15
RANK 6-19
RATIOTOREPORT 6-22
TERTILE 6-24

In This Chapter

Red Brick Decision Server includes several RISQL Entry Tool display functions that are unique to Red Brick Decision Server. These functions operate on sets of rows and perform sequential calculations frequently used in business queries. For example, the function

```
CUME(dollars)
```

returns a running total for each row in a set of rows.

Although they are not defined by the ANSI SQL-92 standard, RISQL display functions are valuable because:

- They simplify the expression of commonly asked business questions.
- They are efficient.
- They are fast compared to other methods of calculating data.

The following table briefly describes each display function:

Function	Description
CUME	Calculates a cumulative sum (a running total).
MOVINGAVG	Calculates a moving average (an average computed over an interval of *n* rows).
MOVINGSUM	Calculates a moving sum (a sum computed over an interval of *n* rows).
NTILE	Determines a rank of *n* tiers.
RANK	Determines rank.
RATIOTOREPORT	Calculates the fractional part of a total.
TERTILE	Determines a three-tiered rank.

This chapter describes these functions in alphabetical order, and includes some examples. Several more detailed examples of queries that contain RISQL display functions are presented in the *SQL Self-Study Guide*.

Some Ground Rules

Within SQL queries, RISQL display functions can be used:

- In the select list
- In expressions
- As arguments of scalar functions
- Within a condition in a WHEN clause
- In subqueries

Display functions cannot be used in:

- The search condition of a WHERE clause
- Arguments for set (aggregate) functions or for other display functions. For example, the expression

  ```
  cume(rank(sales))
  ```

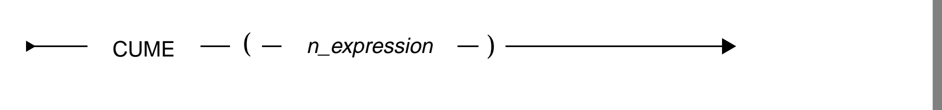
  is not valid. (However, set functions can be used as arguments for display functions.)

CUME

The CUME function calculates a running total by row (including the current row) for a set of values. The running total can be set back to zero in the result by using the RESET BY subclause of the ORDER BY clause.

Syntax

The following syntax diagram shows how to construct an expression with the CUME function:



n_expression The argument *n_expression* must be numeric. It must not reference another display function.

Result

For each row of a set, the function returns the sum of the specified argument for the current and preceding rows. The function ignores unknown and missing values (NULL). Note that the order of the rows in the set affects the return values; refer to "Ordered Result Sets" on page -7.

If the datatype of *n_expression* is an exact datatype (TINYINT, SMALLINT, INTEGER, or DECIMAL), CUME returns an exact datatype. To reduce the chance of the result overflowing its allocated storage, the precision of the result datatype is expanded by six.

The following table summarizes the datatypes returned by CUME for different *n_expression* datatypes:

Datatype of *n_expression*	Datatype of Result
TINYINT	DECIMAL(9,0)
SMALLINT	DECIMAL(11,0)
INTEGER	DECIMAL(16,0)
DECIMAL NUMERIC	DECIMAL(p,s) p = min(38, precision of *n_expression* + 6) s = scale of *n_expression*
REAL	REAL
FLOAT DOUBLE PRECISION	FLOAT DOUBLE PRECISION

RESET BY

The running total can be re-initialized to zero for specified sets of values with the RESET BY subclause of the ORDER BY clause, as described on page 7-47.

Examples

The expression

```
cume(dollars/1000)
```

returns the sum of values in the Dollars column (expressed in thousands) for all the preceding rows (including the Dollars value in the current row).

The following query returns a running total of quantities of Aroma baseball caps sold on Fridays in the first quarter of 2000. An ORDER BY clause is used to ensure that the result set is sorted in chronological order (by values in the Date column).

```
select prod_name, store_name, date,
    cume(quantity) as run_total
from sales natural join period natural join product
    natural join store
where day = 'FR'
    and year = 2000
    and qtr = 'Q1_00'
    and prod_name = 'Aroma baseball cap'
order by date
```

```
PROD_NAME           STORE_NAME                    DATE         RUN_TOTAL
Aroma baseball cap  San Jose Roasting Company     2000-02-11   2
Aroma baseball cap  Miami Espresso                2000-02-18   10
Aroma baseball cap  Olympic Coffee Company        2000-03-03   24
Aroma baseball cap  San Jose Roasting Company     2000-03-17   30
Aroma baseball cap  San Jose Roasting Company     2000-03-24   44
Aroma baseball cap  Beaches Brew                  2000-03-24   48
```

Ordered Result Sets

The results of the following functions might not be computed accurately or consistently unless the query in which they are used contains an ORDER BY clause:

- CUME
- MOVINGAVG
- MOVINGSUM

The ORDER BY clause guarantees that the values computed by the display function are based on a consistently sorted set of values for the numeric expression that serves as the argument of the display function.

When the results of a query expression that contains a CUME, MOVINGSUM, or MOVINGAVG function are combined with the results of another query expression (as in a UNION query), the ORDER BY clause applies to the results of the entire query. Therefore, the values computed by the display function in the first query expression are not based on an ordered set of rows; the ORDER BY clause is applied only *after* the results of the two query expressions have been combined. Because of this behavior, it is often impractical to use the CUME, MOVINGSUM, and MOVINGAVG functions in queries that contain multiple query expressions.

For example, the following query might return incorrect or inconsistent results for the Cume_Sales column, depending on how the results of the first query expression happen to be sorted:

```
select qtr, sum(dollars) as total_sales,
    cume(sum(dollars)) as cume_sales
from sales_west natural join period
group by qtr
union
select qtr, sum(dollars) as total_sales,
    cume(sum(dollars)) as cume_sales
from sales_east natural join period
group by qtr
order by total_sales asc;
```

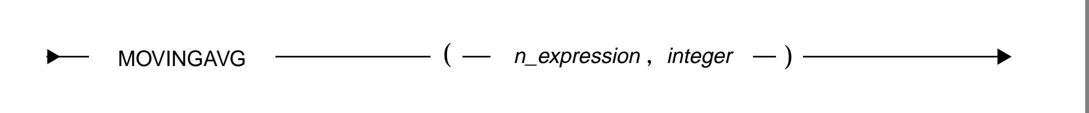
Similarly, although display functions can be included in query expressions defined in views, ORDER BY clauses are not allowed in such query expressions; therefore, it is often impractical to include the CUME, MOVINGSUM, and MOVINGAVG functions in view definitions. For more information about views, see "CREATE VIEW" on page 8-157.

MOVINGAVG

The MOVINGAVG function calculates a moving average by row for a specified set of values over a specified number of rows.

Syntax

The following syntax diagram shows how to construct an expression with the MOVINGAVG function:



n_expression, integer	The first argument, *n_expression*, must be numeric, and the second argument must be a positive integer; *n_expression* must not reference another display function.

Result

For each row of a set, the function returns a moving average calculated as the average of *n_expression* for the current row and the preceding *integer*–1 rows.

If there are *integer* consecutive missing values, the function returns NULL. The first *integer*–1 rows display NULL until enough rows have been processed to calculate the first moving average. Note that the order of the rows in the set affects the return values; refer to “Ordered Result Sets” on page -7.

If the datatype of *n_expression* is an exact datatype (TINYINT, SMALLINT, INTEGER, or DECIMAL), MOVINGAVG returns an exact datatype. The precision and scale of the result datatype are such that the number of digits to the left of the decimal point is maintained, while the number of digits to the right of the decimal point is increased by six. This means that even if the resulting value of the MOVINGAVG function is very small, it will probably still fit into the significant digits of the result datatype.

The following table summarizes the datatypes returned by MOVINGAVG for different *n_expression* datatypes:

Datatype of *n_expression*	Datatype of Result
TINYINT	DECIMAL(9,6)
SMALLINT	DECIMAL(11,6)
INTEGER	DECIMAL(16,6)
DECIMAL NUMERIC	DECIMAL(p,s) p = min(38, precision of *n_expression* + 6) s = min(6, 38- precision of *n_expression*)
REAL	REAL
FLOAT DOUBLE PRECISION	FLOAT DOUBLE PRECISION

RESET BY

The moving average can be re-initialized to zero for groups of values with the RESET BY subclause of the ORDER BY clause, as described on page 7-47.

Examples

The expression

```
movingavg(dollars,6)
```

returns the moving average of Dollars values, calculated as the average dollars for the current and five preceding rows.

The following query returns a moving average of values for the Quantity column. (The DEC scalar function is used to truncate the long-numeric results of the MOVINGAVG calculation.)

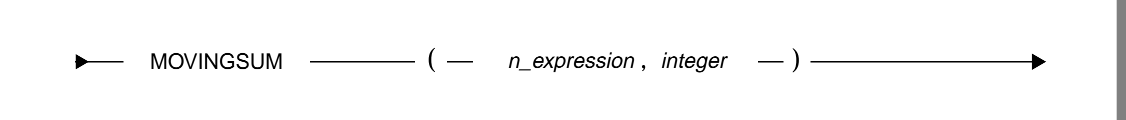
```
select month, quantity, dec(movingavg(quantity, 3),7,2)
from sales natural join period natural join promotion
where month in ('DEC', 'APR')
    and sales.promokey in (1001, 1002, 2001, 2002)
order by sales.perkey
MONTH QUANTITY    MV_AVG
APR           16      NULL
APR           13      NULL
APR           17     15.33
APR            7     12.33
APR           14     12.66
APR            7      9.33
APR           13     11.33
APR           18     12.66
APR            4     11.66
APR           17     13.00
DEC            4      8.33
DEC           12     11.00
DEC            7      7.66
APR            1      6.66
APR            8      5.33
...
```

MOVINGSUM

The MOVINGSUM function calculates a moving sum by row for a specified set of values over a specified number of rows.

Syntax

The following syntax diagram shows how to construct an expression with the MOVINGSUM function:



n_expression, integer — The first argument, *n_expression*, must be numeric, and the second must be an integer; *n_expression* must not reference another display function.

Result

For each row of a set, the function returns a moving sum calculated as the sum of the *n_expression* for the current row and the preceding *integer*–1 rows. If there are *integer* consecutive missing values, the function returns NULL. Note that the order of the rows in the set affects the return values; refer to "Ordered Result Sets" on page -7.

If the datatype of *n_expression* is an exact datatype (TINYINT, SMALLINT, INTEGER, or DECIMAL), MOVINGSUM returns an exact datatype. To reduce the chance of the result overflowing its allocated storage, the precision of the result datatype is expanded by six.

The following table summarizes the datatypes returned by MOVINGSUM for different *n_expression* datatypes:

Datatype of *n_expression*	Datatype of Result
TINYINT	DECIMAL(9,0)
SMALLINT	DECIMAL(11,0)
INTEGER	DECIMAL(16,0)
DECIMAL NUMERIC	DECIMAL(p,s) p = min(38, precision of *n_expression* + 6) s = scale of *n_expression*
REAL	REAL
FLOAT DOUBLE PRECISION	FLOAT DOUBLE PRECISION

RESET BY

The moving sum can be re-initialized to zero for groups of values with the RESET BY subclause of the ORDER BY clause, as described on page 7-47.

Examples

The expression

```
movingsum(dollars,6)
```

returns the moving sum of Dollars values, calculated as the sum of dollars for the current and five preceding rows.

The following query returns a moving sum for the Quantity column:

```
select promo_desc, quantity,
    movingsum(quantity, 3) as mvg_sum
from sales natural join promotion natural join period
    where year = 2000
    and month = 'FEB'
    and promo_desc like '%coupon%'
    order by sales.perkey;

PROMO_DESC                          QUANTITY   MVG_SUM
Aroma catalog coupon                       3          NULL
Aroma catalog coupon                       9          NULL
Aroma catalog coupon                      12            24
Aroma catalog coupon                      11            32
Aroma catalog coupon                       6            29
Aroma catalog coupon                       7            24
Aroma catalog coupon                      28            41
Aroma catalog coupon                      16            51
Aroma catalog coupon                      16            60
Aroma catalog coupon                      14            46
Aroma catalog coupon                       9            39
Aroma catalog coupon                       1            24
Aroma catalog coupon                      39            49
Aroma catalog coupon                      19            59
Aroma catalog coupon                       9            67
Aroma catalog coupon                      24            52
Aroma catalog coupon                      27            60
...
```

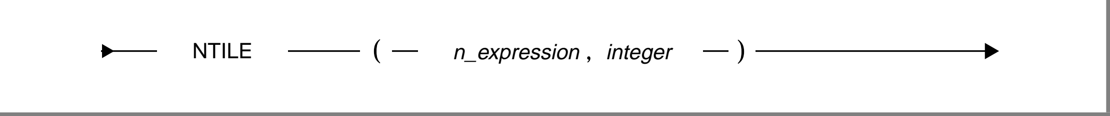
NTILE

The NTILE function determines the rank of a value in terms of a range that you specify. Unlike the TERTILE function, which restricts the range to thirds and returns a character value to represent them (H = high, M = medium, and L = low), the NTILE function returns integers to represent any range of ranks, such as 1 (highest) to 100 (lowest).

The difference between NTILE and RANK is that NTILE divides the result set into fifths, tenths, hundredths, and so on, according to the integer value you specify, whereas RANK simply arranges the whole result set hierarchically.

Syntax

The following syntax diagram shows how to construct an expression with the NTILE function:



n_expression, integer	The argument *n_expression* must be numeric, and *integer* must be a positive, non-zero integer; *n_expression* must not reference another display function.

Result

If the *n_expression* argument is not NULL, the function returns an integer that represents a rank within the requested range. For example, if you set the integer argument to 5, 1 is returned if a given value falls into the highest rank, 5 if a value falls into the lowest, and so on.

RESET BY

The ranking can be re-initialized to zero for specified groups with the RESET BY subclause of the ORDER BY clause, as described on page 7-47.

Usage Notes

When a set of values is not divisible by the specified integer, the NTILE function puts any leftover rows in the higher-level groups. In those cases where equal values span a boundary, they are distributed between adjacent groups; results might vary from query to query.

For example, the expression

```
ntile(col_name, 3)
```

will return the values 1, 1, 2, and 3 if there are four rows in the result set; and 1, 1, 2, 2, and 3 if there are five rows.

The NTILE function can be used to localize the output of the TERTILE function, which is described on page 6-24.

Examples

The following query ranks sales of coffee and tea products by the sum of the values in the Dollars column. The ranking is in sixths, so each product name receives a value from 1 to 6.

```
select prod_name, ntile(sum(dollars), 6) as sales_rank
from sales natural join product
where product.classkey in (1, 2, 4, 5)
group by prod_name
PROD_NAME                   SALES_RANK
Demitasse Ms                         1
Xalapa Lapa                          1
NA Lite                              1
Lotta Latte                          1
Cafe Au Lait                         2
Expresso XO                          2
Aroma Roma                           2
Veracruzano                          3
La Antigua                           3
Colombiano                           3
Darjeeling Special                   4
Assam Gold Blend                     4
Darjeeling Number 1                  4
Irish Breakfast                      4
English Breakfast                    5
Breakfast Blend                      5
Earl Grey                            5
Assam Grade A                        6
Special Tips                         6
Gold Tips                            6
```

The following example uses the NTILE function inside a CASE expression to spread the tiled values (fifths) into three unequal groups: the top 20% are represented as *top_20*, the middle 60% as *mid_60*, and the bottom 20% as *low_20*.

```
select prod_name, date, dollars,
    case ntile(dollars, 5)
        when 1 then 'top_20'
        when 2 then 'mid_60'
        when 3 then 'mid_60'
        when 4 then 'mid_60'
        when 5 then 'low_20'
        end as n_rank
from sales natural join product
    natural join period
    natural join store
where year = 2000
    and day = 'TH'
    and store_name like 'Minnesota Roaster%'
order by prod_name
PROD_NAME            DATE          DOLLARS   N_RANK
Aroma Roma           2000-03-23     166.75 mid_60
Aroma Roma           2000-02-17     224.75 mid_60
Cafe Au Lait         2000-02-24     119.00 mid_60
Colombiano           2000-03-02     135.00 mid_60
Colombiano           2000-02-24     175.50 mid_60
Colombiano           2000-01-06     148.50 mid_60
Colombiano           2000-01-20     162.00 mid_60
Colombiano           2000-03-23     128.25 mid_60
Colombiano           2000-02-03     276.75 top_20
Demitasse Ms         2000-02-24      61.50 low_20
Demitasse Ms         2000-03-30     292.50 top_20
Demitasse Ms         2000-02-24     185.25 mid_60
Expresso XO          2000-02-17      74.25 low_20
La Antigua           2000-01-13     100.75 low_20
La Antigua           2000-03-23     210.25 mid_60
La Antigua           2000-03-16     181.25 mid_60
La Antigua           2000-01-13     159.50 mid_60
Lotta Latte          2000-01-13      85.00 low_20
Lotta Latte          2000-02-17     161.50 mid_60
Lotta Latte          2000-03-09     240.00 mid_60
Lotta Latte          2000-01-27     127.50 mid_60
NA Lite              2000-01-06     396.00 top_20
NA Lite              2000-03-02     297.00 top_20
NA Lite              2000-01-06     126.00 mid_60
NA Lite              2000-02-17     108.00 low_20
Veracruzano          2000-01-13     360.00 top_20
Veracruzano          2000-02-10     322.50 top_20
Veracruzano          2000-02-03     142.50 mid_60
Veracruzano          2000-01-27     232.50 mid_60
```

By adding a WHEN clause to the previous query, you can eliminate all but the top 20% from the result set:

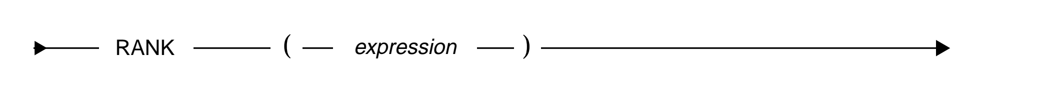
```
select prod_name, date, dollars,
    case ntile(dollars, 5)
        when 1 then 'top_20'
        when 2 then 'mid_60'
        when 3 then 'mid_60'
        when 4 then 'mid_60'
        when 5 then 'low_20'
        end as n_rank
from sales natural join product
    natural join period
    natural join store
where year = 2000
    and day = 'TH'
    and store_name like 'Minnesota Roaster%'
when n_rank = 'top_20'
order by prod_name
PROD_NAME            DATE          DOLLARS   N_RANK
Colombiano           2000-02-03     276.75 top_20
Demitasse Ms         2000-03-30     292.50 top_20
NA Lite              2000-01-06     396.00 top_20
NA Lite              2000-03-02     297.00 top_20
Veracruzano          2000-01-13     360.00 top_20
Veracruzano          2000-02-10     322.50 top_20
```

RANK

The RANK function determines the rank of a specified value relative to a group of values.

Syntax

The following syntax diagram shows how to construct an expression with the RANK function:



expression The argument *expression* can be of any datatype; it must not reference another display function. If the expression is preceded by a minus (–) sign, a reverse (low-to-high) ranking results. For non-numeric datatypes, the ranking is based on the collation sequence defined in the server locale specification.

Result

If the argument is not NULL, the function returns an integer rank for the argument relative to other values in the set; otherwise, the function returns NULL. You can rank values from low-to-high or high-to-low (the default).

When the values to be ranked are equal, they are assigned the same ranking value. For example, two rows might receive the rank of 3, in which case 4 would be skipped.

Top Ten

The final result table can be restricted to the top five, top ten, or an arbitrary number of rankings with the WHEN clause. For more information about this clause, see "WHEN Clause" on page 7-34.

RESET BY

The rank can be re-initialized to zero for groups of values with the RESET BY subclause of the ORDER BY clause, as described on page 7-47.

Examples

The expression

```
rank(-dollars)
```

will return the integer rank of a row according to the Dollars value for each row in a group. The rankings will be in reverse order (lowest to highest).

The following query ranks products in terms of their total sales over the life of the database (27 months):

```
select prod_name, sum(dollars) as prod_sales,
    rank(sum(dollars)) as prod_rank
from product join sales on sales.classkey = product.classkey
    and sales.prodkey = product.prodkey
group by prod_name
PROD_NAME                          PROD_SALES      PROD_RANK
Demitasse Ms                        656401.50              1
Xalapa Lapa                         577450.00              2
NA Lite                             557655.00              3
Lotta Latte                         533454.50              4
Cafe Au Lait                        526793.50              5
Expresso XO                         514094.50              6
Aroma Roma                          479330.25              7
Veracruzano                         479015.50              8
La Antigua                          473434.50              9
Colombiano                          462003.50             10
Ruby's Allspice                     299977.00             11
Darjeeling Special                  292751.00             12
Assam Gold Blend                    156962.00             13
Darjeeling Number 1                 136768.25             14
Irish Breakfast                     109281.00             15
English Breakfast                   100459.50             16
Breakfast Blend                      93790.75             17
Earl Grey                            90798.00             18
Assam Grade A                        88651.00             19
Special Tips                         87413.25             20
Gold Tips                            86315.75             21
Espresso Machine Italiano            48057.95             22
Aroma t-shirt                        45632.80             23
Espresso Machine Royale              35754.00             24
Tea Sampler                          32411.00             25
Aroma baseball cap                   32249.00             26
Coffee Sampler                       32220.00             27
```

```
Spice Sampler                               16219.00           28
Aroma Sheffield Steel Teapot                15797.00           29
Aroma Sounds CD                             15779.00           30
Aroma Sounds Cassette                       11642.50           31
French Press, 4-Cup                          9727.55           32
Spice Jar                                    9694.00           33
French Press, 2-Cup                          7060.25           34
Easter Sampler Basket                        5280.00           35
Travel Mug                                   3312.80           36
Coffee Mug                                   2793.00           37
Christmas Sampler                            1920.00           38
```

To eliminate all but the top five, add a WHEN clause to the previous query:

```
select prod_name, sum(dollars) as prod_sales,
    rank(sum(dollars)) as prod_rank
from product join sales on sales.classkey = product.classkey
    and sales.prodkey = product.prodkey
group by prod_name
when prod_rank <=5
PROD_NAME                         PROD_SALES      PROD_RANK
Demitasse Ms                           656401.50            1
Xalapa Lapa                            577450.00            2
NA Lite                                557655.00            3
Lotta Latte                            533454.50            4
Cafe Au Lait                           526793.50            5
```

The following query ranks Price values in reverse order (low to high), where the lowest value is assigned the rank of 1:

```
select price, rank(-price)
from orders
where order_no > 3616
PRICE
  3995.95           1
  4325.25           2
  4325.25           2
  4425.00           4
  4425.00           4
  5400.00           6
  5400.00           6
 10234.50           8
 10234.50           8
 16500.00          10
```

Depending on how a query is written, the order in which the ranked rows are returned might vary, but this order does not affect the rank of the values:

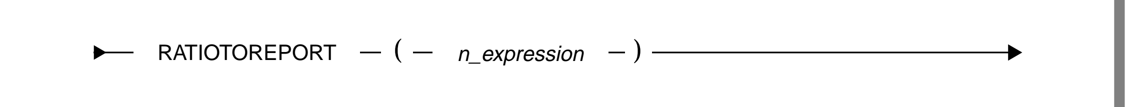
```
select price, rank(-price)
from orders
where order_no > 3616
order by price desc
PRICE
 16500.00          10
 10234.50           8
 10234.50           8
  5400.00           6
  5400.00           6
  4425.00           4
  4425.00           4
  4325.25           2
  4325.25           2
  3995.95           1
```

RATIOTOREPORT

The RATIOTOREPORT function calculates the ratio of a value to the sum of a group of values.

Syntax

The following syntax diagram shows how to construct an expression with the RATIOTOREPORT function:



n_expression The *n_expression* argument must be numeric; it must not reference another display function.

Result

If the argument is not NULL, the function returns the ratio of a value to the sum of a group of values; otherwise, the function returns NULL.

RESET BY

The RATIOTOREPORT function can be reset for groups of values with the RESET BY subclause of the ORDER BY clause, as described on page 7-47.

Examples

The expression

```
ratiotoreport(quantity)
```

returns the ratio of the Quantity value for a given row to the sum of all quantities in a set of values.

The following query returns the ratio of sales to total sales (multiplied by 100 to return a percentage for the ratio):

```
select city, sum(dollars) as sales,
    ratiotoreport(sum(dollars))*100 as ratio_dollars
from sales natural join store natural join period
group by city
order by sales desc
CITY                  SALES           RATIO_DOLLARS
San Jose                    896931.15           12.58
Atlanta                     514830.00            7.22
Miami                       507022.35            7.11
Los Angeles                 503493.10            7.06
Phoenix                     437863.00            6.14
New Orleans                 429637.75            6.03
Cupertino                   424215.00            5.95
Boston                      421205.75            5.91
Houston                     417261.00            5.85
New York                    397102.50            5.57
Los Gatos                   394086.50            5.53
Philadelphia                392377.75            5.50
Milwaukee                   389378.25            5.46
Detroit                     305859.75            4.29
Chicago                     294982.75            4.14
Hartford                    236772.75            3.32
Minneapolis                 165330.75            2.32
```

TERTILE

The TERTILE function determines the rank of a value as High, Middle, or Low relative to a group of values.

Syntax

The following syntax diagram shows how to construct an expression with the TERTILE function:

n_expression The *n_expression* argument must be numeric; it must not reference another display function.

Result

If the argument is not NULL, the function returns the character *H*, *M*, or *L* (High, Middle, or Low); otherwise, the function returns NULL.

RESET BY

The ranking can be re-initialized to zero for specified groups with the RESET BY subclause of the ORDER BY clause, as described on page 7-47.

Usage Notes

When a set of values is not divisible by three, the TERTILE function puts any leftover rows in the higher-level groups. In those cases where equal values span a boundary, they are distributed between adjacent groups; results might vary from query to query. An example of this kind of variation is presented in the description of the NTILE function, earlier in this chapter.

You can use the NTILE function to reproduce the behavior of the TERTILE function while localizing its output—that is, to change the values *H*, M, and L to three alternative character strings that are more meaningful in the language of the database. For more information about the NTILE function, refer to page 6-15.

Examples

The expression

```
tertile(dollars)
```

ranks a group of rows by Dollars values and returns a High, Middle, or Low rank for each row in the group.

The following query returns a High, Middle, or Low rank for the Dollars column and the Quantity column:

```
select prod_name, dollars, quantity,
    tertile(dollars) as sales,
    tertile(quantity) as qty
from store natural join sales
    natural join period
    natural join product
where year = 2000
and week = 2
and store_name like 'Instant%';
PROD_NAME                      DOLLARS   QUANTITY    SALE QTY
Lotta Latte                     328.00          41 H    H
Colombiano                      216.00          32 H    H
Lotta Latte                     216.00          27 H    H
La Antigua                      210.25          29 H    H
Cafe Au Lait                    136.00          16 H    H
Darjeeling Special              126.50          11 H    L
Darjeeling Special              115.00          10 H    L
Lotta Latte                     110.50          13 H    M
Lotta Latte                     110.50          13 H    H
Colombiano                      108.00          16 H    H
Xalapa Lapa                      99.00          11 M    M
Darjeeling Number 1              94.50          18 M    H
Aroma Roma                       94.25          13 M    M
...
```

The following macro uses the NTILE function to reproduce the behavior of the TERTILE function. When this macro is used in a query, the values *A*, *B*, and *C* are returned for tiled sums of dollars. This kind of CASE expression can return any three character strings, as appropriate to the user's language.

```
create macro nls_tertile as
    case ntile((sum(dollars)), 3)
        when 1 then 'A'
        when 2 then 'B'
        when 3 then 'C'
        end as tile;
```

Chapter

Query Expressions

7

In This Chapter 7-3

Join and Non-Join Query Expressions 7-3

Joined Tables 7-6
- Syntax 7-7
- Qualified Joins 7-7
  - Inner Joins 7-7
  - Outer JoinsS 7-8
  - Qualified-Join Syntax 7-9
- Cross Joins 7-12
- Table References 7-13

Query Specifications 7-16
- Select List 7-17
  - Set Functions 7-20
- FROM Clause 7-21
  - Joins in the FROM Clause 7-22
  - Qualified Column Names 7-23
- WHERE Clause 7-26
  - Join Predicates 7-27
  - Non-Standard Outer Join Syntax 7-28
- GROUP BY Clause 7-29
- HAVING Clause 7-32
- WHEN Clause 7-34

UNION, EXCEPT, and INTERSECT Expressions 7-36
  - Datatype Conversions 7-40

SELECT Statements 7-46
ORDER BY Clause 7-47
Ordering Sequence 7-49
RESET BY Subclause. 7-52
BREAK BY Subclause 7-55
SUPPRESS BY Clause. 7-58
How SELECT Statements Are Processed 7-59

Subqueries 7-60
Scalar Subqueries and Table Subqueries 7-61
Correlated Subqueries 7-64
Qualified Column Names 7-65
Column Name Resolution 7-66

In This Chapter

This chapter defines the rules for writing SQL queries, starting with a discussion of the query expression and proceeding with explanations and examples of its components. The chapter ends with descriptions of the complete SELECT statement and subqueries.

This chapter is divided into six main sections:

- Join and Non-Join Query Expressions
- Joined Tables
- Query Specifications
- UNION, EXCEPT, and INTERSECT Expressions
- SELECT Statements
- Subqueries

In addition to the examples in this chapter, there are several more detailed examples of different kinds of query expressions in the *SQL Self-Study Guide*.

Join and Non-Join Query Expressions

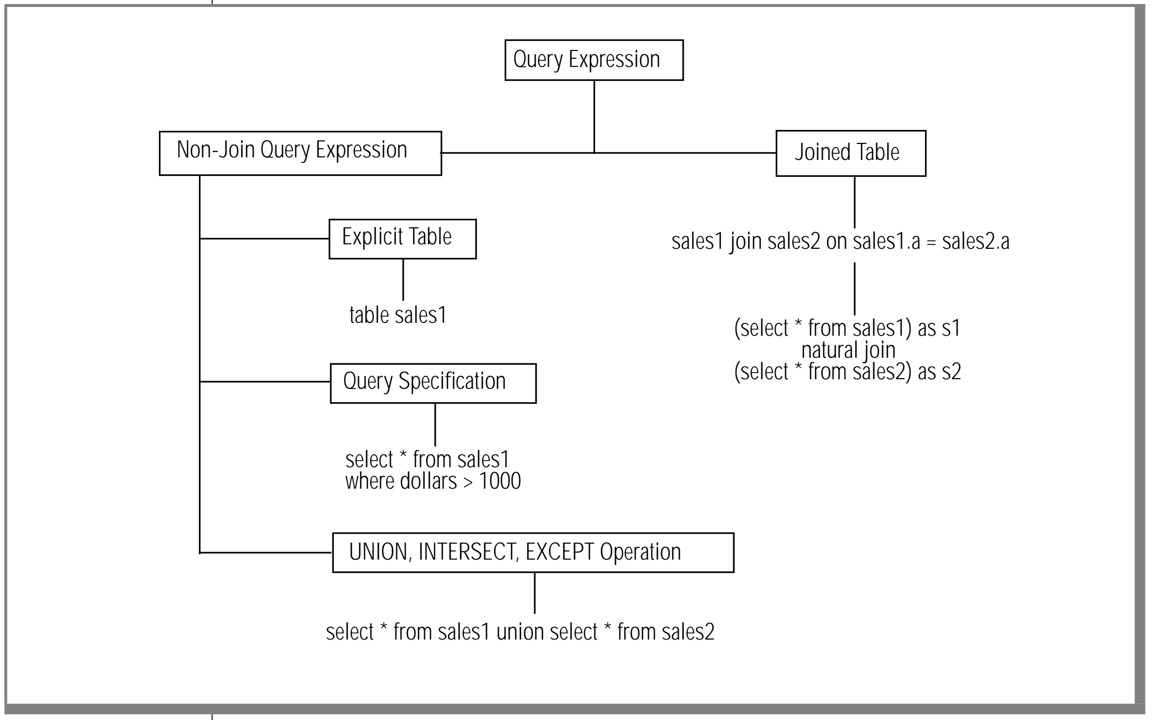
A query expression evaluates selected data into a table of results. In simple queries, this table is returned to the user as a final result set; in more complex queries, it is used as an intermediate table that is combined with the results of other query expressions to produce the final result.

The query expression is at the top of the hierarchy of SQL query-writing constructs. In many commercial texts, it is referred to as a *table expression*, but the term *query expression* is used throughout this document to be consistent with the ANSI SQL-92 standard.

Each query expression takes one of two forms—join or non-join. A non-join query expression derives its results from a *query specification* (otherwise known as a select expression), from an *explicit table*, or from a more complex expression that includes UNION, INTERSECT, or EXCEPT operators.

A join query expression, or *joined table*, derives its results from the explicit joining of two or more table references. A table reference can be a simple table name, a joined table in its own right, or another query expression.

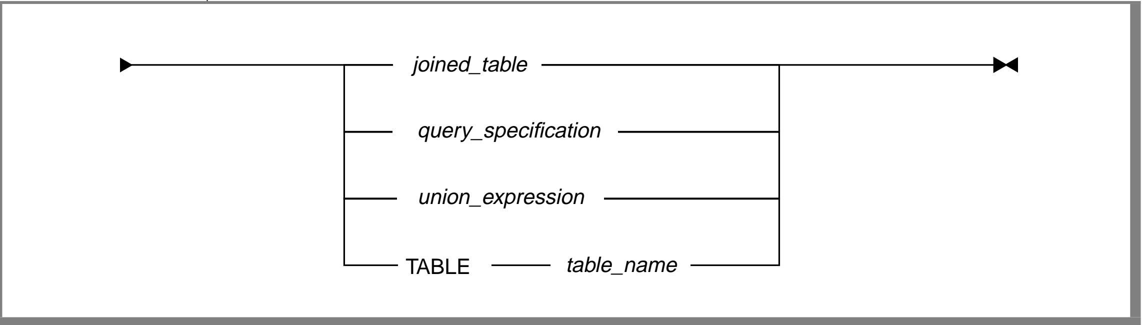
The following diagram identifies the different types of query expression, and presents some simple examples:



In its simplest form, a query expression is a single query specification. In its more complex forms, a query expression derives its results from the evaluation of multiple intermediate tables derived from subqueries, joined tables, and expressions that contain UNION, INTERSECT, and EXCEPT operators. This flexibility enables the query writer to combine several data-selection methods in a single query, and to write queries that do not require the presence of the SELECT keyword. Simply by adding a semicolon (or some other entry-tool terminator) to the expression, you could issue each of the above examples as an SQL SELECT statement.

Syntax

The following syntax diagram shows how to construct a query expression:



joined_table Specifies a query expression that explicitly joins two table references.

For a complete syntax diagram and description, see "Joined Tables" on page 7-6.

query_specification Specifies a query expression that must begin with the SELECT keyword and a FROM clause. It may also contain a number of optional clauses that, if used, must be listed in the correct sequence.

For a complete syntax diagram and description, see "Query Specifications" on page 7-16.

union_expression Specifies a query expression that consists of two or more query expressions connected with UNION, INTERSECT, or EXCEPT operators. (The term *union_expression* refers to the concept of set theory, which these operators emulate.)

For a complete syntax diagram and description, see "UNION, EXCEPT, and INTERSECT Expressions" on page 7-36.

TABLE *table_name* Specifies an explicit table that represents all the rows in the named table. For example:

```
table store
```

is equivalent to

```
select * from store
```

Joined Tables

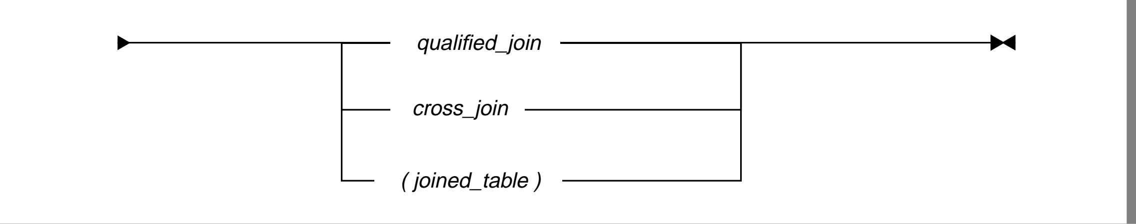
A joined table is a query expression that explicitly joins two table references. There are three types of joined table:

- Qualified join—an inner or outer join that names the tables to be joined and the joining columns
- Cross join, or Cartesian product join
- A joined table enclosed in parentheses

Any two tables can be joined, as long as the joining columns have comparable datatypes and those columns are clearly specified in the query.

Syntax

The following syntax diagram shows how to construct a *joined_table*:



Qualified Joins

A qualified join is an inner or outer join that can be specified in one of three different ways: as a join over columns with identical names (known as a *natural join*), as a join over named columns, or as a join over a predicate. This section contains definitions of the terms *inner join* and *outer join*, followed by syntax descriptions of the three specifications.

Inner Joins

An inner join concatenates rows from two or more tables based on matching column values. If the column values of rows from the joined tables match, the server concatenates the rows and places them in an intermediate result table.

If there are any tables in a query that are not explicitly joined (using standard qualified-join syntax), the server will compute the Cartesian product (the set of all possible combinations of rows) of those tables.

For example, consider the two tables State and Region:

```
STATE   REGION
statecity   city    area
------------------------------------
FL  JacksonvilleJacksonvilleSouth
FL  Miami   Miami   South
GA  Atlanta Atlanta South
TN  NashvilleNew OrleansSouth
```

An inner join of these two tables (defined on the City column) produces the following result set:

```
statestate.cityregion.cityarea
-----------------------------------
FL  JacksonvilleJacksonvilleSouth
FL  MiamiMiami  South
GA  AtlantaAtlantaSouth
```

The inner join condition defined on the City column is:

```
state.city = region.city
```

Outer JoinsS

There are three kinds of outer join—left, right, and full.

Each type of outer join generates a different intermediate result set. A left outer join returns all the rows that would be returned by an inner join, plus all the rows from the left (or first-listed) table that do not match any row from the right table. Conversely, a right outer join returns all the inner-join rows, plus all the rows from the right (or second-listed) table that do not match any row from the left table. A full outer join retains all the rows returned from both tables.

For all three types of outer join, NULLs represent columns in rows that do not match.

For example, a left outer join of the State and Region tables contains all the rows of the State table and a NULL in each column of any row that does not match a corresponding row in the Region table:

```
statestate.cityregion.cityarea
-----------------------------------
FL  JacksonvilleJacksonvilleSouth
FL  MiamiMiami  South
GA  AtlantaAtlantaSouth
TN  NashvilleNULLNULL
```

A right outer join of the State and Region tables is similar except that it contains all the rows of the Region table.

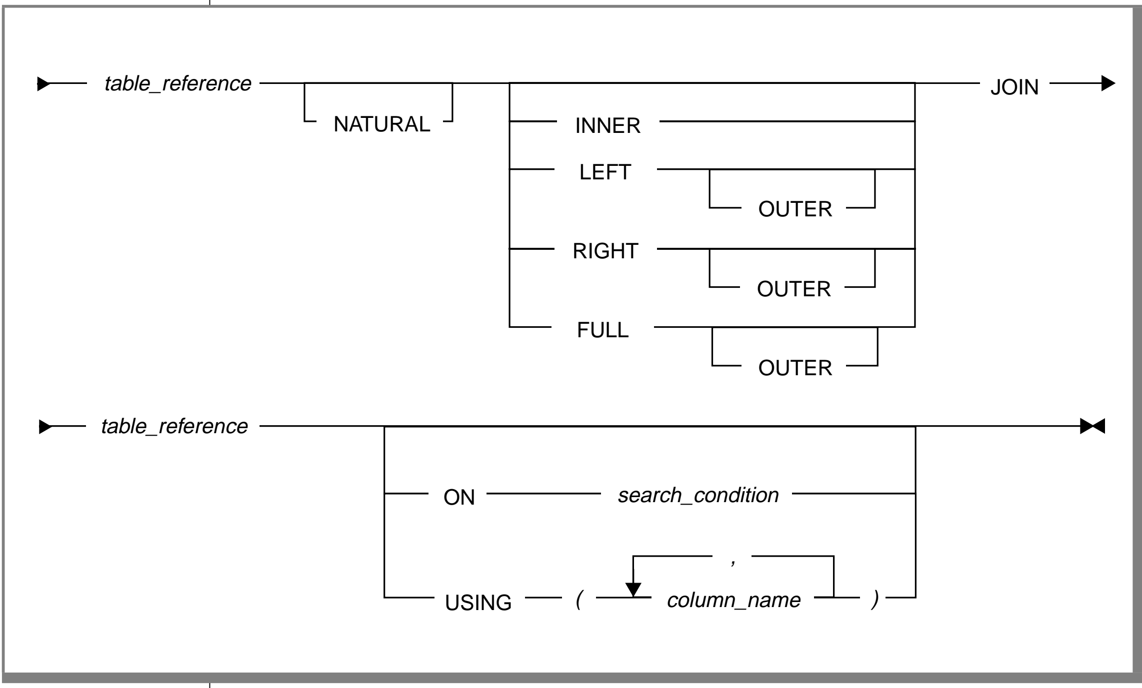
```
statestate.cityregion.cityarea
-----------------------------------
FL  JacksonvilleJacksonvilleSouth
FL  MiamiMiami  South
GA  AtlantaAtlantaSouth
NULL NULL New OrleansSouth
```

A full outer join contains all the rows of both tables:

```
statestate.cityregion.cityarea
------------------------------------
FL  JacksonvilleJacksonvilleSouth
FL  MiamiMiami  South
GA  AtlantaAtlantaSouth
TN  NashvilleNULLNULL
NULL NULL New OrleansSouth
```

Qualified-Join Syntax

The following syntax diagram shows how to construct a *qualified_join*:



table_reference	Specifies a named table, view, or synonym or a query expression that evaluates to a table. For the complete syntax of the *table_reference*, see "Table References" on page 7-13.
NATURAL	Joins tables over their identically named columns; the column names are not explicitly stated. The natural join operates on all pairs of identically named columns. For example: `sales natural join product` joins the Sales and Product tables over their identically named Classkey and Prodkey columns. If the NATURAL keyword is used, neither the ON clause nor the USING clause may be used.
INNER	Specifies an inner join of the table references. The INNER keyword is optional.
LEFT, RIGHT, FULL	Specifies a left, right, or full outer join of the table references.
OUTER	Specifies an outer join of the table references. The OUTER keyword is optional; specifying LEFT, RIGHT, or FULL implies an outer join.
JOIN	Specifies a qualified join of the table references. This keyword is required for inner and outer joins. If you use JOIN without the INNER and OUTER keywords, an inner join is implied.

ON search_condition Specifies a join over a predicate. An ON clause cannot be specified if a USING clause or a NATURAL join is specified.

The *search_condition* defines the predicate that must be satisfied in order for two rows to join. Any valid search condition that references columns from the two tables being joined can be used. In most cases, the predicate expresses a simple equijoin; however, it may be a complex expression such as

```
table1.col1 = table2.col2 + table2.col3
```

or a non-equijoin such as

```
table1.col1 > table2.col2
```

In all cases, the search condition must be preceded by the ON keyword. For example:

```
on sales.classkey = product.classkey
and sales.prodkey = product.prodkey
```

USING (column_name) Specifies a join over one or more named columns. For example:

```
using (classkey, prodkey)
```

The listed columns must exist in both of the tables participating in the join.

A USING clause cannot be specified if an ON clause or a NATURAL join is specified.

Usage Notes

When you join tables with the NATURAL JOIN or USING syntax, the result is a table that consists of all of the common column names in the order they were found in the first table, followed by all the columns in the first table not participating in the join, followed by all the columns in the second table not participating in the join. The datatype of a column that results from a join might be different from the datatype of either of the joining columns that produced it; however, the resulting datatype will be large enough to hold any value from either column.

When you join tables with the ON syntax, both joining columns (and their datatypes) are preserved in the result. For example, if you join the Sales and Period tables on their respective Perkey columns, the intermediate result set will contain two Perkey columns, not one.

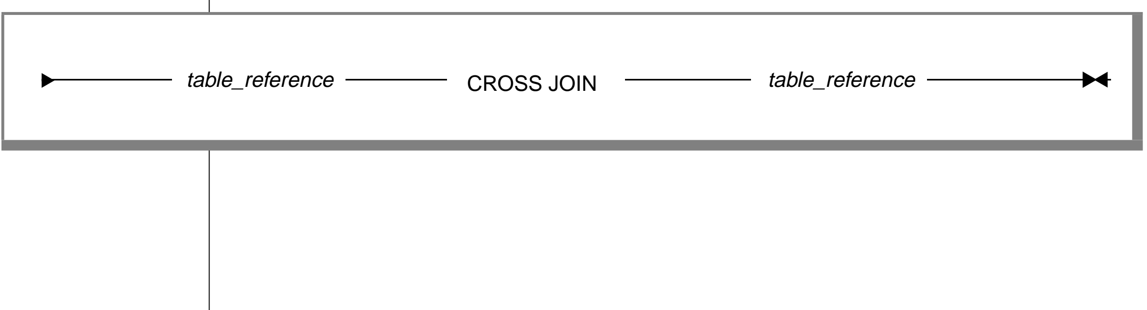
Cross Joins

A *cross join* computes the Cartesian product of all the rows from the joined tables—that is, the result represents every possible combination of those rows.

By default, the server returns an error message when queries that include cross joins are issued. To enable cross-join queries, issue a SET CROSS JOIN ON command or edit the OPTION CROSS_JOIN parameter in the *rbw.config* file. For more information about this parameter, refer to page 9-10.

Syntax

The following syntax diagram shows how to construct a *cross_join*:

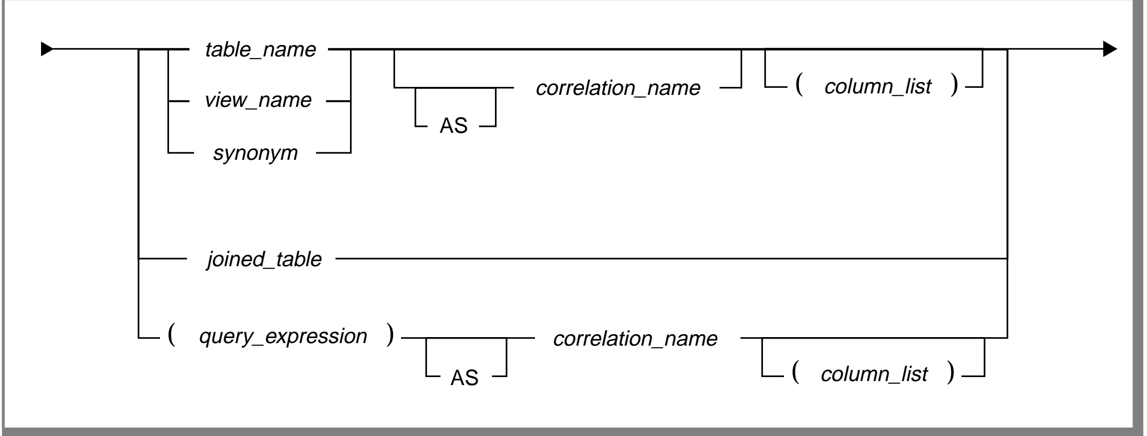


table_reference Specifies a named table, view, or synonym or a query expression that evaluates to a table. For the complete syntax of the *table_reference*, see "Table References" on page 7-13.

CROSS JOIN Concatenates all the rows from the joined tables and returns a result set that represents every possible combination of those rows.

Table References

The following syntax diagram shows how to construct a table reference:



table_name, view_name, synonym Specifies table names, view names, or names of synonyms, all of which can be qualified with correlation names. For information about synonyms, refer to "CREATE SYNONYM" on page 8-131.

AS correlation_name Assigns a name to a base table or a table derived from a query expression for the duration of the query. The correlation name, once defined, must be used in all references to the table in the query. The AS keyword is optional.

Correlation names must immediately follow their table references. For example, the following list of table references assigns the correlation names *sr*, *pr*, *pd*, and *sl* to the Store, Product, Period, and Sales tables, respectively.

```
store sr, product pr, period pd, sales sl
```

Correlation names should be assigned to prevent ambiguous references that might occur when a query and a subquery both reference the same table(s). For more information about subqueries, refer to page 7-60.

If a correlation name is assigned to a table, qualified column names can be expressed only with the correlation name and not with the table name. For example, if the Period table is assigned the correlation name *pd*, the Month column of the Period table can be specified as either *month* or *pd.month* —provided there are no other Month columns in the scope of the column reference that could cause ambiguity. Specifying the column as *period.month* would result in an error message.

Correlation names are usually assigned whenever a subquery must correlate its references with those of its parent. For example, the following condition correlates references to Month made by a subquery and its parent:

```
pp.month = pd.month
```

The condition forces any references made by the subquery (*pp.month*) to be the same as references made by its parent (*pd.month*).

Another use for correlation names is to give distinct names to tables that are being joined to themselves (or *self-joined*). For more information about self-joins, refer to the *SQL Self-Study Guide*.

column_list

Assigns names to the columns in the table reference for the duration of the query. If present, the list must include a distinct name for each column in the table.

If the table reference is a query expression that participates in either a natural join or a join over named columns, the column names must be unique.

joined_table

Specifies any valid join query expression, as defined under "Joined Tables" on page 7-6.

(query_expression)

Specifies any non-join query expression or joined table, as defined on page 7-3. A query expression used as a table reference must be enclosed in parentheses and given a correlation name; then it can be joined with other tables or the results of other query expressions. For example, the following query includes a query expression in its FROM clause.

```
select *
from (class natural join product)
as cp(col1, col2, col3, col4, col5, col6)
```

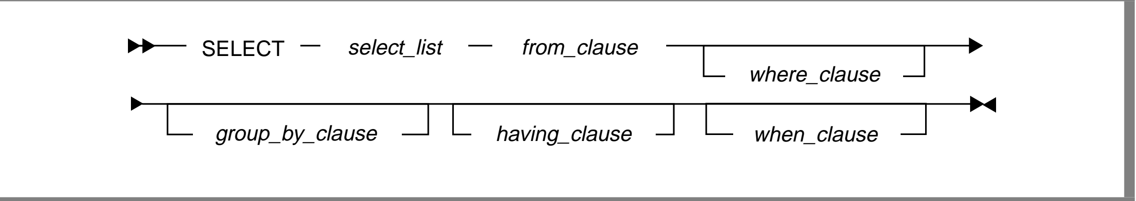
The column list is optional.

Query Specifications

The query specification—sometimes referred to as a *select expression*—contains standard SQL clauses that must be used in the correct sequence; these are the SELECT, FROM, WHERE, GROUP BY, HAVING, and WHEN clauses.

Syntax

The following syntax diagram shows how to construct a query specification:



A query specification must contain:

- The SELECT keyword
- A select list
- A FROM clause

The other clauses are optional. If one or more optional clauses are included, however, they must occur in the order specified in the syntax diagram.

The user must have SELECT privilege on all base tables and views referenced in the query specification; otherwise, the server will reject the query.

Example

The following query returns the name of each Class_Type in the Class table:

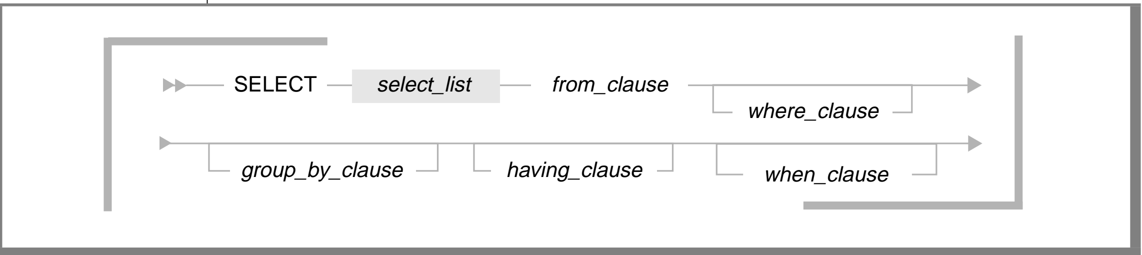
```
select class_type
from class
CLASS_TYPE
Bulk_beans
Bulk_tea
Bulk_spice
Pkg_coffee
Pkg_tea
Pkg_spice
Hardware
Gifts
Clothing
```

Select List

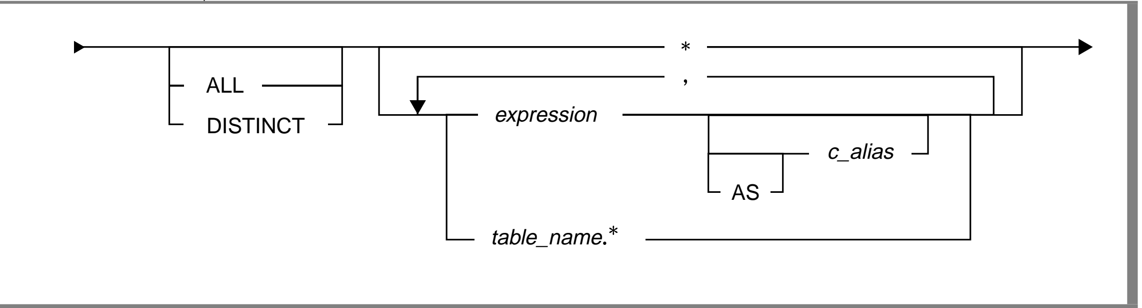
A select list specifies which columns are returned in the final result table and whether duplicate rows are eliminated. Alternative names (or “aliases”) for columns in the result table can also be defined in the select list.

Syntax

The syntax of the query specification is repeated here to provide a point of reference for the *select_list* syntax:



The following syntax diagram shows how to construct a *select_list*:

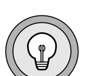


ALL Directs the server to return all rows of the result table.

DISTINCT Directs the server to eliminate duplicate rows from the result table.

A row in the result table is a duplicate of another row if each column value of the first row is equal to the corresponding value of the second row (NULLs are considered equal). Note that a result table contains only those columns in the select list. The ALL or DISTINCT keyword can be used only once in a select list.

asterisk (*) The asterisk (*) is an abbreviation for a list of all the column names defined in the FROM clause.



Tip: *Red Brick Decision Server also supports explicit tables, whereby the query:*

```
select * from market
```

can be stated simply as

```
table market
```

expression Defines a specific column in the result table. Typically, an expression is a column name, a qualified column name, or a function (set, scalar, datetime, or RISQL display) associated with a column name.

An expression can also be a scalar subquery—a subquery that returns exactly one scalar value (character, datetime, numeric, or NULL). For additional information about subqueries, refer to page 7-60.

AS c_alias Specifies a column alias, which defines the name of the column in the table that results from the evaluation of the expression. In turn, the alias is returned as a column heading in the final result table. Aliases can be referenced in other clauses of the query (WHERE, GROUP BY, and so on), as well as in subsequent columns in the select list.

Column names in a select list are processed from left to right. A column alias must be defined in the AS clause before it is used in the select list. For example, the following select list would result in an error because *rank_dollars* is specified before it is defined in the AS clause:

```
select rank_dollars, rank(sum(dollars)) as
rank_dollars
```

The above select list can be rewritten as follows:

```
select rank(sum(dollars)) as rank_dollars,
rank_dollars
```

The AS keyword is optional. For example, the following select list defines *rank_dollars* as an alias for *rank(sum(dollars))*:

```
select prod_name, rank(sum(dollars)) rank_dollars
```

The column alias is a database identifier and must conform to the rules for names and identifiers, as defined on page 2-3.

*table_name.** Specifies all the columns from the named table. For example:

```
product.*
```

Set Functions

If the query specification does not contain a GROUP BY clause, the select list must contain only set functions, literal values, or uncorrelated subqueries. If the select list contains only set functions, the result table contains a single row; otherwise, it can contain multiple rows depending on the search condition specified in the WHERE clause.

If the query specification contains a GROUP BY clause, an expression in the select list must be one of the following:

- A character or datetime literal or a numeric constant.
- A column name specified in the GROUP BY clause or an expression constructed from column names specified in the GROUP BY clause.
- A set function whose argument is a constant or an expression that references only column names of tables defined by the FROM clause.
- A RISQL display function whose argument is a constant, an expression that references only column names specified in the GROUP BY clause, or a set function that is legal for the query.

The result table contains one row for each group.

Examples

The following query returns the name of each distinct region in the Market table:

```
select distinct region
from market
REGION
Central
North
South
West
```

The following query returns the names of each region, and counts the number of stores in each one:

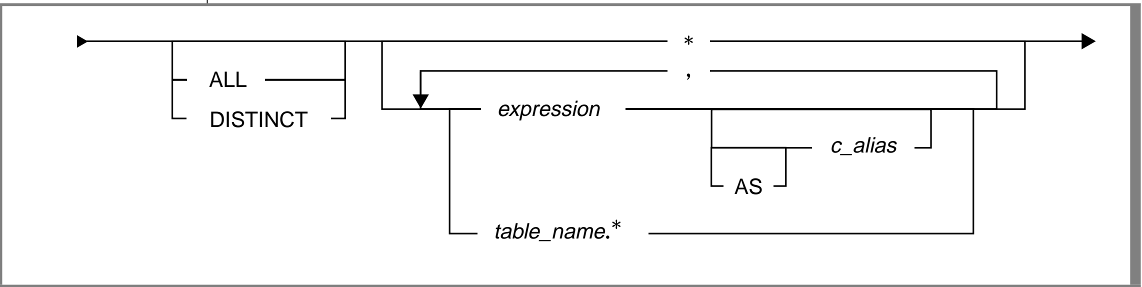
```
select region, count(store_name) as nbr_stores
from market natural join store
group by region
REGION               NBR_STORES
South                         4
North                         4
Central                       4
West                          6
```

FROM Clause

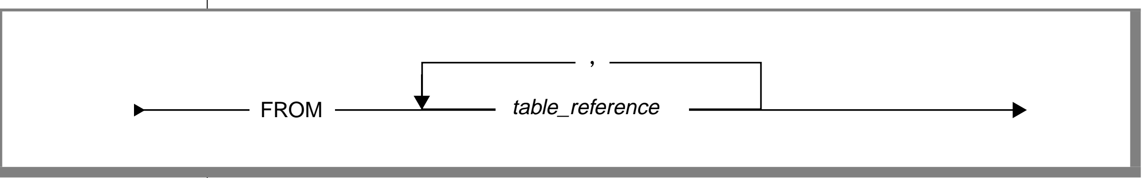
The FROM clause lists the sources of the data retrieved and evaluated by the query specification. Each source is a table reference.

Syntax

The syntax of the query specification is repeated here to provide a point of reference for the FROM clause syntax:



The following syntax diagram shows how to construct a FROM clause:



table_reference Specifies any named table (base, system, view, or synonym), a joined table (a query expression that joins two or more tables), or a query expression inside parentheses. For a complete syntax diagram and description, refer to "Table References" on page -13.

If a table reference in the FROM clause is a query expression enclosed in parentheses, it must be qualified with a correlation name. Correlation names can also be used for tables and views, either to provide abbreviated alternative names or to distinguish references to the same table by a query and its subqueries.

If a single table is referenced, it is the only source of data for the query specification. If there are multiple table references, the logical result of the specification is the Cartesian product (or cross join) of the table references. To prevent the Cartesian product from being computed, join specifications must be stated explicitly in either the FROM clause or the WHERE clause.

A cross join is implicit in the following table references:

```
product, sales
```

and explicit in the following joined table:

```
product cross join sales
```

Joins in the FROM Clause

A single join operation retrieves and concatenates rows from two table references. Note the following rules and restrictions regarding joins:

- A specific join operation can be an inner or an outer (left, right, or full) join, but the columns to be joined must be explicitly specified in all cases.
- The tables to be joined must be listed in the FROM clause, but the joining columns can be specified in either the FROM clause or the WHERE clause, regardless of the type of join (inner or outer).
- Joining columns must have comparable datatypes.

- You can join a table to itself (that is, reference the same table twice in the same FROM clause), but you must distinguish the table references by using a correlation name for at least one of the tables. For an example of a self-join query, refer to the *SQL Self-Study Guide*.
- Cross joins (or Cartesian product joins) are computed only if the OPTION CROSS_JOIN parameter in the *rbw.config* file is set to ON or the user has issued a SET CROSS JOIN ON statement. This restriction is a safeguard against the execution of unintended cross-join queries—resulting from an incorrect qualified-join specification, for example.
- Union joins are not supported.
- There are three ways to specify an inner or outer join in the FROM clause:
  - ON subclause (join over a predicate)
  - USING subclause (join over named columns)
  - NATURAL JOIN syntax (join over all pairs of identically named columns)
- If an outer join condition is specified in the FROM clause, the first table is considered the "left" table and the second is the "right." If the join is specified in the WHERE clause, the outer join is taken from the specified operator, as shown in the syntax diagram on page 7-28.

For the full syntax of each type of join, see "Qualified Joins" on page 7-7.

Qualified Column Names

To prevent ambiguity when tables referenced in a FROM clause have identical column names, references to those columns must be qualified with the name of the table to which they belong. For example, the Storekey column of the Store table can be specified as either of the following:

- storekey
- store.storekey

However, if the FROM clause includes references to the Store and Sales tables, references to the Storekey column must be qualified:

- store.storekey
- sales.storekey

A table derived from a query expression can have columns with duplicate names. If these columns are referenced elsewhere in the query, they must be named distinctly—either by using column aliases in the appropriate query specification or by providing a derived column-name list after the correlation name.

For more information about qualified column names, refer to page 2-8.

Examples

The following query returns the average price of all orders and line items in their respective tables. Qualified column names must be assigned in the select list and the FROM clause to distinguish the Price and Order_No columns from the two tables.

```
select avg(orders.price) as orders_avg,
    avg(line_items.price) as line_avg
from orders join line_items
    on orders.order_no = line_items.order_no
ORDERS_AVGLINE_AVG
12340.647582411380.16296703
```

The Promotion table contains the following columns: Promokey, Promo_Type, Promo_Desc, Value, Start_Date, and End_Date. The following example uses a correlation name *p* for the table Promotion, then uses a column list to temporarily rename the Promokey column to Promo_Code and the Value column to Amount:

```
select promo_code, amount
from promotion p(promo_code, promo_type, promo_desc, amount,
    start_date, end_date)
```

If you were to add a natural join specification to this query, between the Promotion and Sales tables, no common column names would be found and the query would require a cross join. For the duration of the query, the Promokey column from the Promotion table does not exist; its name is Promo_Code.

The following queries use three different join specifications in the FROM clause but produce identical result sets:

```
select prod_name, dollars
from sales natural join product
where dollars < 1000
select prod_name, dollars
from sales join product
    on sales.classkey = product.classkey
        and sales.prodkey = product.prodkey
where dollars < 1000
select prod_name, dollars from sales join product
    using (classkey, prodkey)
where dollars < 1000
```

The following example contains a natural join of three tables:

```
select class_type, prod_name, dollars
from class natural join product natural join sales
CLASS_TYPE   PROD_NAME                        DOLLARS
Clothing     Aroma t-shirt                      54.75
Clothing     Aroma t-shirt                     109.50
Clothing     Aroma t-shirt                     107.40
Clothing     Aroma t-shirt                     197.10
Clothing     Aroma t-shirt                      54.75
Clothing     Aroma t-shirt                      43.80
...
```

In this example, a subquery is used in the FROM clause to constrain the Value column of the Promotion table before joining the derived table to the Sales table. A correlation name (*p*) must be supplied for the derived table.

```
select distinct promo_desc, sum(dollars) as dollars
from (select * from promotion where value > 2.00) as p
    join sales on sales.promokey = p.promokey
group by promo_desc
PROMO_DESCDOLLARS
Christmas special1920.00
Easter special5280.00
```

System tables can be joined in the same way as base tables and views. The following example joins the RBW_SEGMENTS and RBW_STORAGE tables to list the segment name (Storage), the filename of the physical storage unit (PSU_Location), and the associated base table (Table_Name).

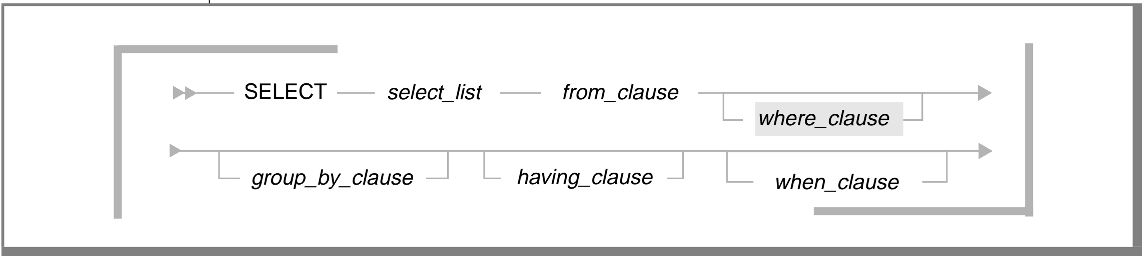
```
select segname as storage, location as psu_location,
    tname as table_name
from rbw_storage join rbw_segments
    on rbw_storage.segname = rbw_segments.name;
STORAGE                PSU_LOCATION    TABLE_NAME
RBW_SYSTEM             RB_DEFAULT_SEGM NULL
RBW_SYSTEM             RB_DEFAULT_INDE NULL
RBW_SYSTEM             RB_DEFAULT_TABL NULL
RBW_SYSTEM             RB_DEFAULT_LOCK NULL
RBW_SYSTEM             RB_DEFAULT_IDX  NULL
DEFAULT_SEGMENT_1      dfltseg1_psu1   MARKET
DEFAULT_SEGMENT_2      dfltseg2_psu1   MARKET
DEFAULT_SEGMENT_3      dfltseg3_psu1   STORE
DEFAULT_SEGMENT_5      dfltseg5_psu1   CLASS
DEFAULT_SEGMENT_7      dfltseg7_psu1   PRODUCT
...
```

WHERE Clause

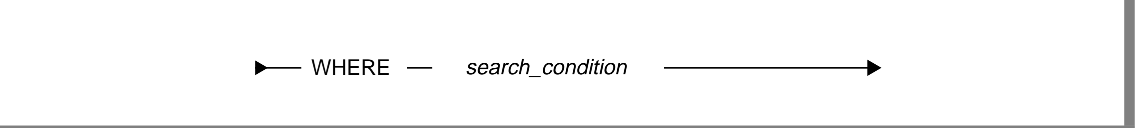
The WHERE clause specifies a search condition that is applied to each row of the intermediate result table generated by the FROM clause.

Syntax

The query specification is repeated here to provide a point of reference for the *where_clause* syntax:



The following syntax diagram shows how to construct a *where_clause*:



search_condition Evaluates to true, false, or unknown for each row of the intermediate result table generated by the FROM clause. If the condition is true, the row is retained in the result table; otherwise, it is discarded. If the WHERE clause is omitted, all the rows of the intermediate result table are retained.

The search condition must meet the following requirements:

- It must be constructed according to the rules described under "Conditions" on page 3-10.
- Each column it references must uniquely specify a column of the intermediate result table or be a correlated column reference (possible only in a subquery).

Join Predicates

Join specifications may be specified in the WHERE clause, given that the tables to be joined are listed in the FROM clause. The predicate for the join is subject to the rules described on page 7-11. For example, the query

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

is an inner equijoin of the State and Region tables over their City columns.

Non-Standard Outer Join Syntax

The syntax strongly preferred for outer joins is the FROM clause syntax described on page 7-9, which is consistent with the ANSI SQL-92 standard.

For migration purposes only, Red Brick Decision Server also has limited support for the syntax shown in the following diagram. This syntax can be used to construct left, right, and full outer join specifications, respectively, in a WHERE clause:

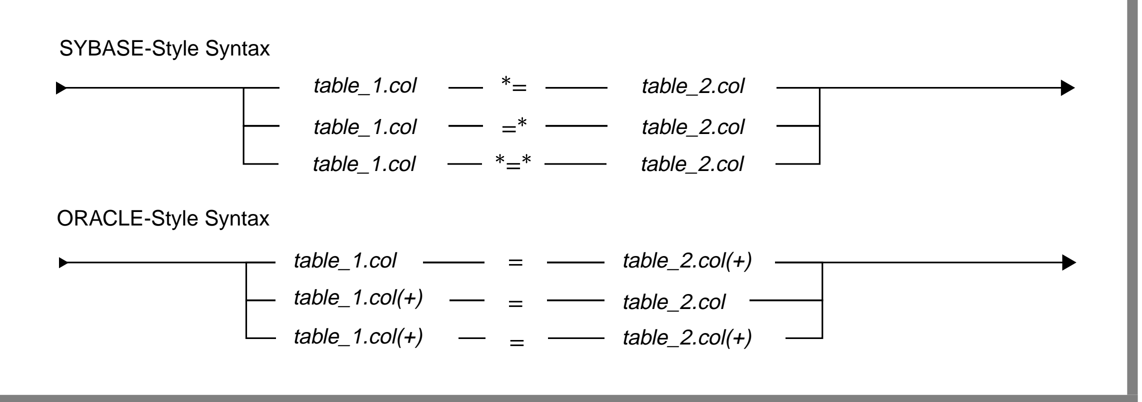


table_1.col, table_2.col

Specifies any two columns from any two tables that have comparable datatypes. For example, a left outer join condition on the Sales and Orders tables can be specified as follows:

```
where sales.perkey *= orders.perkey
```

All the rows returned by the inner equijoin defined on the Perkey columns are returned, plus those rows of the Sales table that fail to match any row of the Orders table on the shared column.

The same left outer join can alternatively be specified as follows:

```
where sales.perkey = orders.perkey(+)
```

Example

The following query returns the dollar amounts for products whose sales exceeded $500 on any given day during December 1998. Notice that the column alias for the Dollars column, Top_Prods, is used in the WHERE clause. The WHERE clause is also used to specify the joining columns for the tables listed in the FROM clause.

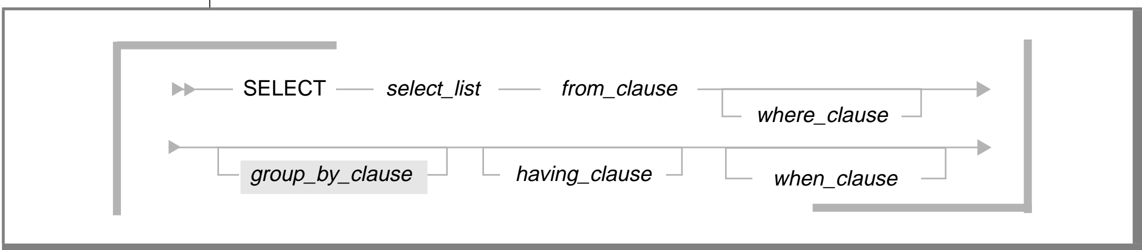
```
select prod_name, dollars as top_prods
from sales s, product p, period d
where s.classkey = p.classkey
    and s.prodkey = p.prodkey
    and s.perkey = d.perkey
    and year = 1998
    and month = 'DEC'
    and top_prods > 500;
PROD_NAME                          TOP_PRODS
Espresso Machine Italiano             699.65
Coffee Sampler                        570.00
```

GROUP BY Clause

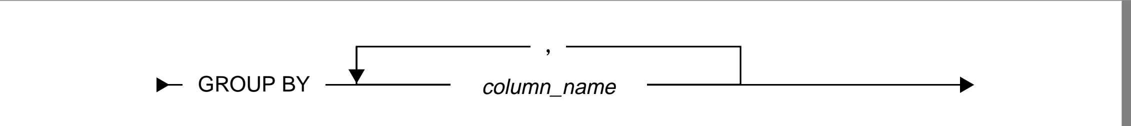
The GROUP BY clause divides the result table into groups according to the columns specified.

Syntax

The query specification is repeated here to provide a point of reference for the *group_by_clause* syntax:



The following syntax diagram shows how to construct a *group_by_clause*:



column_name Uniquely specifies a column in the result table. The columns referenced in the GROUP BY clause are called "grouping columns."

The GROUP BY clause divides the result table into groups of rows defined by the grouping columns. A group can contain one or more rows. If a group contains more than one row, each row has the same value in its grouping columns. NULLS are treated as distinct values in a column. The result table contains one row that summarizes each group.

The column used to group the results can be displayed in the result table. If results are grouped by a column that is not in the select list, the grouping column is not displayed in the result table.

If a query contains a GROUP BY clause, all columns in the select list that reference columns in the query's FROM clause must either be arguments of set functions or be listed in the GROUP BY clause. In other words, an expression in the select list must be one of the following:

- A character or datetime literal or a numeric constant.
- A column name or column alias specified in the GROUP BY clause or an expression constructed from column names specified in the GROUP BY clause.
- A set function whose argument is a constant or an expression that references only column names of tables defined by the FROM clause.
- A RISQL display function whose argument is a constant, an expression that references only column names specified in the GROUP BY clause, or a set function that is legal for the query.

The names of grouping columns can occur in the search condition of a HAVING clause.

If a query that contains a select-list subquery requires a GROUP BY clause, the correlation columns, if any, must be identified in the GROUP BY clause of the outer query. For information about correlated subqueries, refer to page 7-64 of this document and to the *SQL Self-Study Guide*.

Example

The following query returns the total sales figures for January 2000, grouped by promotion description:

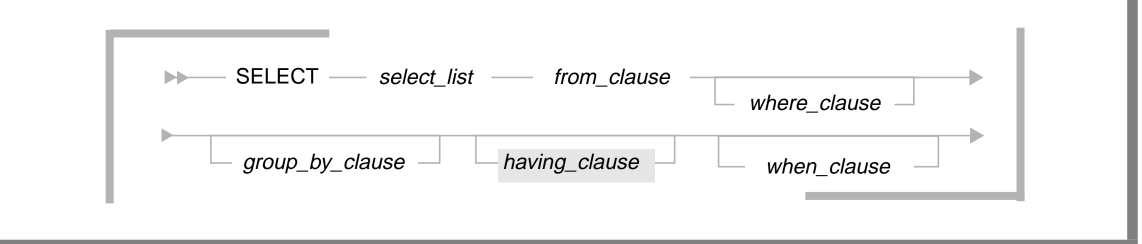
```
select promo_desc, sum(dollars) as promo_totals
from promotion natural join sales natural join period
where year = 2000
    and month = 'JAN'
group by promo_desc

PROMO_DESCPROMO_TOTALS
No promotion250668.75
Temporary price reduction4046.25
Aroma catalog coupon5216.80
Monthly coffee special1083.00
Store display1896.75
```

HAVING Clause

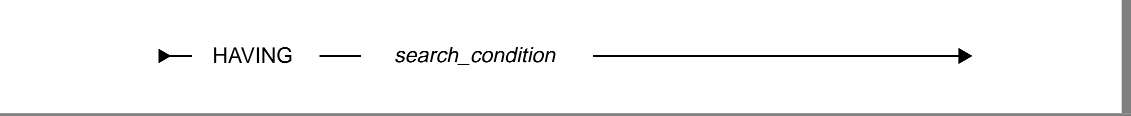
The HAVING clause specifies a search condition that is applied to the result table generated by the GROUP BY clause. This clause can be used only if the query specification contains set functions in the select list or a GROUP BY clause.

Syntax

The query specification is repeated here to provide a point of reference for the *having_clause* syntax:



The following syntax diagram shows how to construct a *having_clause*:



search_condition Evaluates to true, false, or unknown for each group of the result table. If the condition is true for a group, it is retained; otherwise it is discarded. If the HAVING clause is omitted, all the groups of the result table are retained.

The search condition must meet the following requirements:

- The condition must be constructed in accordance with the rules described under "Conditions" on page 3-10.
- Each column referenced in the condition must specify a grouping column, be a correlated-column reference, or be specified within a set function.

When a correlated subquery occurs in a search condition, it is evaluated for each group of the result table.

Example

The following query returns the 1998 totals for products whose sales exceeded $50,000:

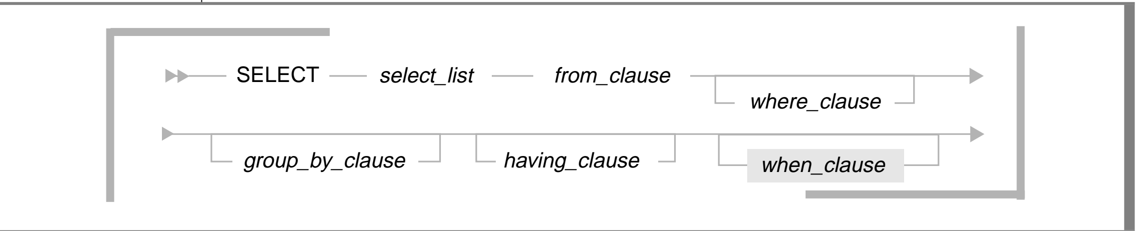
```
select prod_name, sum(dollars) as total_1998
from product natural join sales natural join period
where year = 1998
group by prod_name
having total_1998 > 50000

PROD_NAME TOTAL_199
Darjeelin  62283.25
La Antigu 197069.50
Expresso  224020.00
Aroma Rom 203544.00
Colombian 188474.50
Veracruza 201230.00
NA Lite   231845.00
Ruby's Al 133188.50
Darjeelin 127207.00
Xalapa La 251590.00
Lotta Lat 217994.50
Demitasse 282385.25
Assam Gol  71419.00
Cafe Au L 213510.00
```

WHEN Clause

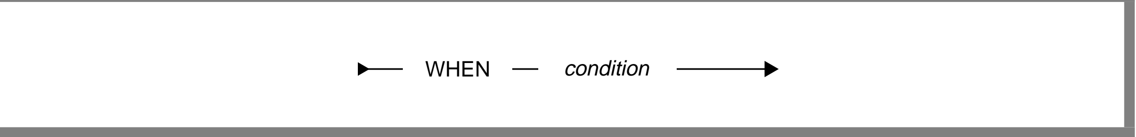
The WHEN clause specifies a condition on the result table after the computation of set functions (AVG, MIN, MAX, SUM, COUNT), the evaluation of any HAVING clause, and any processing for RISQL display functions. Rows that satisfy the condition are retained in the final result table; otherwise they are discarded.

Syntax

The query specification is repeated here to provide a point of reference for the *when_clause* syntax:



The following syntax diagram shows how to construct a *when_clause*:



condition Specifies a condition on the result table after the computation of aggregate functions and the evaluation of any HAVING clause. Compound conditions constructed with the AND, OR, and NOT logical connectives are allowed.

For detailed information about conditions, refer to Chapter 3, "Expressions and Conditions."

Usage Note

RISQL display functions in the WHEN clause are only affected by simple RESET BY clauses. In order to filter rows based on complex RESET BY specifications, a column alias in the select list must be used and the WHEN clause must reference that column alias; for an example of this scenario, refer to "BREAK BY Subclause" on page 7-55.

Example

The following query ranks products by their total dollar sales in 1999, then returns the figures for the top 10 products only:

```
select prod_name, sum(dollars) as sales,
    rank(sum(dollars)) as top_ten_99
from sales natural join product
    natural join period
where year = 1999
group by prod_name
when top_ten_99 <= 10
```

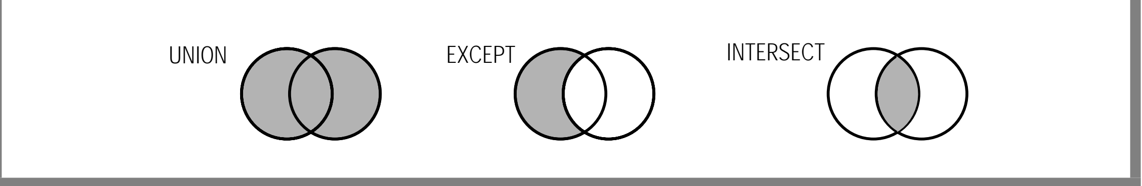
UNION, EXCEPT, and INTERSECT Expressions

The UNION, EXCEPT, and INTERSECT operators all operate on multiple result sets to return a single result set.

- The UNION operator combines the output of two query expressions into a single result set. Query expressions are executed independently, and their output is combined into a single result table.
- The EXCEPT operator evaluates the output of two query expressions and returns the difference between the results. The result set contains all rows returned from the first query expression except those rows that are also returned from the second query expression.
- The INTERSECT operator evaluates the output of two query expressions and returns only the rows common to each result.

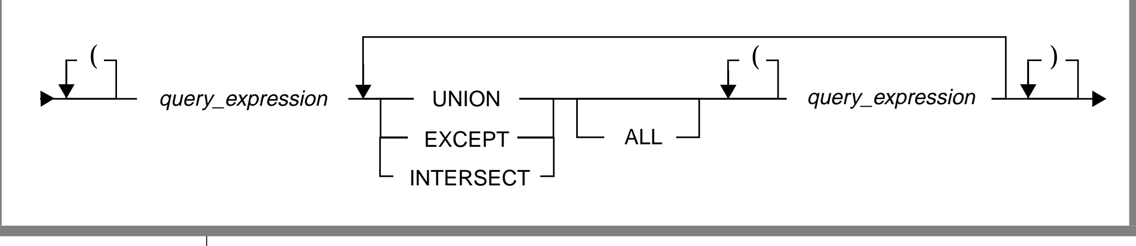
The following figure illustrates these set theory concepts with Venn diagrams, where the shaded portion indicates the result set:

Figure 7-1
caption text here



Syntax

The following syntax diagram shows how to construct a *union_expression*:



query_expression Specifies any valid join or non-join query expression, as defined on page 7-5.

Each query expression in the statement must have the same number of columns in its select list. These corresponding columns must be listed in the same order, but they do not have to have the same names. The names of the columns in the result table are taken from the first query expression; parentheses do not change this rule. To define a different column heading for the result table, use the AS clause in the select list of the first query expression; alternatively, if the *union_expression* occurs in the FROM clause, provide headings by using a correlation name and a column list (see page 7-13).

Corresponding columns in each query expression must be of the same datatype, or there must be a legal implicit conversion between the datatypes. Smaller datatypes are converted to larger datatypes, and less precise datatypes are converted to more precise datatypes. For example, if a column is of datatype REAL and the corresponding column is of datatype FLOAT, the column in the result table is FLOAT because FLOAT is more precise than REAL.

The Datatype Conversion Results table on page 7-40 defines conversions between datatypes of corresponding columns.

UNION | EXCEPT | INTERSECT

UNION combines the rows from two or more result sets into a single result set.

EXCEPT evaluates two result sets and returns all rows from the first set that are not also contained in the second set.

INTERSECT computes a result set that contains the common rows from two result sets.

UNION, INTERSECT, and EXCEPT operators can be combined in a single *union_expression*.

In statements that include multiple operators, the default order of evaluation (precedence) for these operators is left to right; however, the INTERSECT operator is evaluated before UNION and EXCEPT. The order of evaluation can be modified with parentheses.

ALL

If ALL is specified, duplicate rows returned by the *union_expression* are retained. If two query expressions return the same row, two copies of the row are returned in the final result. If ALL is not specified, duplicate rows are eliminated from the result set.

In statements with multiple UNION, EXCEPT, and INTERSECT operators in which the ALL keyword is used, the order of evaluation can affect the results. The placement of the ALL keyword in relation to the query evaluation determines the duplicates that are retained and eliminated. If the last operation performed does not contain the ALL keyword, any duplicates retained from previous evaluations are eliminated.

Usage Notes

The UNION and EXCEPT operators have the same precedence. A query with multiple operators is evaluated from left to right. For example:

```
query_expression1 union query_expression2 except query_expression3
```

is evaluated as:

```
(query_expression1 union query_expression2) except
query_expression3
```

The INTERSECT operator has higher precedence than the UNION and EXCEPT operators. Therefore, in the absence of parentheses, INTERSECT operations are always evaluated first. For example:

```
query_expression1 union query_expression2 intersect
query_expression3
```

is evaluated as:

```
query_expression1 union (query_expression2 intersect
query_expression3)
```

If parentheses override the default precedence, they affect the processing order and therefore which duplicate rows are retained and eliminated.

Joins specified in a *union_expression* take higher precedence than all three operators unless the default order of precedence is changed with parentheses.

An INSERT INTO...SELECT statement can contain UNION, INTERSECT, and EXCEPT operators. For example:

```
insert into orders.all
    select * from orders.new
    union
    select * from orders.old
```

When queries use search conditions that contain OR connectives, you can sometimes improve performance by rewriting those queries with UNION operators. For an example, refer to Chapter 5 of the *SQL Self-Study Guide*.

Datatype Conversions

The following tables show the datatype conversion results when corresponding columns with different datatypes are compared in a UNION, INTERSECT, or EXCEPT operation.

The following table shows the conversion results when the database server compares any of the supported data types with the CHAR, VARCHAR, DEC, TINYINT, or SMALLINT data types:

	CHAR(n)	VARCHAR(n)	DEC(p2, s2)	TINYINT	SMALLINT
CHAR(m)	CHAR(MAX(m,n))	VARCHAR(MAX(m,n))	error	error	error
VARCHAR(m)	VARCHAR(MAX(m,n))	VARCHAR(MAX(m,n))	error	error	error
DEC(p1,s1)	error	error	†	†	†
TINYINT	error	error	†	TINYINT	SMALLINT
SMALLINT	error	error	†	SMALLINT	SMALLINT
INTEGER	error	error	†	INTEGER	INTEGER
REAL	error	error	FLOAT	REAL	REAL
FLOAT**	error	error	FLOAT	FLOAT	FLOAT
DATE	error	error	error	error	error
TIME(m)	error	error	error	error	error
TIMESTAMP(m)	error	error	error	error	error

The following table shows the conversion results when the database server compares any of the supported data types with the INTEGER, REAL, FLOAT, DATE, TIME, or TIMESTAMP data types:.

	INTEGER	REAL§	FLOAT**	DATE	TIME(n)	TIMESTAMP(n)
CHAR(m)	error	error	error	error	error	error
VARCHAR(m)	error	error	error	error	error	error
DEC(p1,s1)	†	FLOAT	FLOAT	error	error	error
TINYINT	INTEGER	REAL	FLOAT	error	error	error
SMALLINT	INTEGER	REAL	FLOAT	error	error	error
INTEGER	INTEGER	FLOAT	FLOAT	error	error	error
REAL	FLOAT	REAL	FLOAT	error	error	error
FLOAT**	FLOAT	FLOAT	FLOAT	error	error	error
DATE	error	error	error	DATE	error	error
TIME(m)	error	error	error	error	TIME(MAX(m,n))	error
TIMESTAMP(m)	error	error	error	error	error	TIMESTAMP(MAX(m,n))

†The conversion varies depending on the precision and scale of the datatypes. The system will use the smallest possible datatype to hold the value.

§REAL is the same as FLOAT(4).

**FLOAT is the same as DOUBLE PRECISION.

Examples

UNION

The first example contains a simple UNION operation that combines data from the City and Hq_City columns in the Store and Market tables:

```
select hq_city as ca_cities
from market
where hq_state like 'CA%'
union
select city
from store
where state like 'CA%'

CA_CITIES
Cupertino
Los Angeles
Los Gatos
Oakland
San Francisco
San Jose
```

UNION ALL

The second example adds the ALL keyword, so duplicate Hq_City and City names are retained:

```
select hq_city as ca_cities
from market
where hq_state like 'CA%'
union all
select city
from store
where state like 'CA%'

CA_CITIES
San Jose
San Francisco
Oakland
Los Angeles
Los Gatos
San Jose
Cupertino
Los Angeles
San Jose
```

EXCEPT

The following example replaces the UNION operator with EXCEPT; consequently, only those California cities that are in the Market table but not the Store table are returned.

```
select hq_city as ca_cities
from market
where hq_state like 'CA%'
except
select city
from store
where state like 'CA%'

CA_CITIES
Oakland
San Francisco
```

INTERSECT

The following example replaces the UNION operator with INTERSECT; consequently, only those California cities that are in both the Market table and the Store table are returned.

```
select hq_city as ca_cities
from market
where hq_state like 'CA%'
intersect
select city
from store
where state like 'CA%'

CA_CITIES
Los Angeles
San Jose
```

Multiple INTERSECTs

The following query uses multiple INTERSECT operations to return a list of common key values in five different tables:

```
select prodkey as common_keys from product
    intersect select classkey from class
    intersect select promokey from promotion
    intersect select perkey from period
    intersect select storekey from store

COMMON_KEYS
1
3
4
5
12
```

Order of Evaluation

The following example uses parentheses to force the order of evaluation in a query that contains a UNION operation and an INTERSECT operation. The parentheses force the UNION operator to be evaluated first; without them, the INTERSECT operator would take precedence and the result set might differ.

```
(select prod_name from product
    natural join sales_canadian
union
select prod_name from product
    natural join sales_mexican)
intersect
select prod_name from product
    natural join sales
```

Because parentheses override the default order of evaluation for UNION, EXCEPT, and INTERSECT operations, they sometimes determine whether duplicate rows are retained or eliminated from the final result set. The following two queries illustrate this point.

In the first query, the default left-to-right order of evaluation dictates that the first and second query expressions are evaluated first, retaining duplicates because of the UNION ALL. The subsequent UNION of the result of the first and second query expressions with the third query expression discards all duplicates.

```
select prod_name from product
    natural join sales_canadian
union all
select prod_name from product
    natural join sales_mexican
union
select prod_name from product
    natural join sales
```

In the second query, the UNION of the second and third query expressions is evaluated first and discards duplicates. Then the UNION ALL of the first query expression and the result of the second and third query expressions retains duplicates.

```
select prod_name from product
    natural join sales_canadian
union all
(select prod_name from product
    natural join sales_mexican
union
select prod_name from product
    natural join sales)
```

For additional examples of queries that contain UNION, INTERSECT, and EXCEPT operators, refer to the *SQL Self-Study Guide*.

SELECT Statements

A SELECT statement retrieves multiple rows of data from the database. It consists of any query expression followed by optional SUPPRESS BY and ORDER BY clauses. To turn a query expression into a SELECT statement that can be executed by the server, simply append a semicolon (or some other terminator, depending on the entry tool you are using).

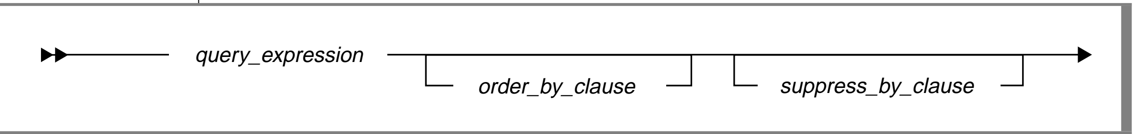
Here are some simple examples:

- Joined table:
  ```
  sales_east natural join sales_west;
  ```
- Query specification:
  ```
  select * from sales_east natural join sales_west;
  ```
- UNION expression:
  ```
  select * from sales_east union select * from sales_west;
  ```
- Explicit table:
  ```
  table sales_east;
  ```

A SELECT statement does not always require the SELECT keyword; in some cases, the select list and FROM clause are implied by the query expression.

Syntax

The following syntax diagram shows how to construct a SELECT statement:



query_expression Specifies any valid join or non-join query expression, as defined on page 7-5.

order_by_clause Sorts the rows of the result table. For details, "ORDER BY Clause" on page 7-47. This clause may also contain two RISQL subclauses—RESET BY and BREAK BY—as defined on page 7-52 and page 7-55, respectively.

A SELECT statement that contains a BREAK BY clause cannot be used inside an INSERT statement.

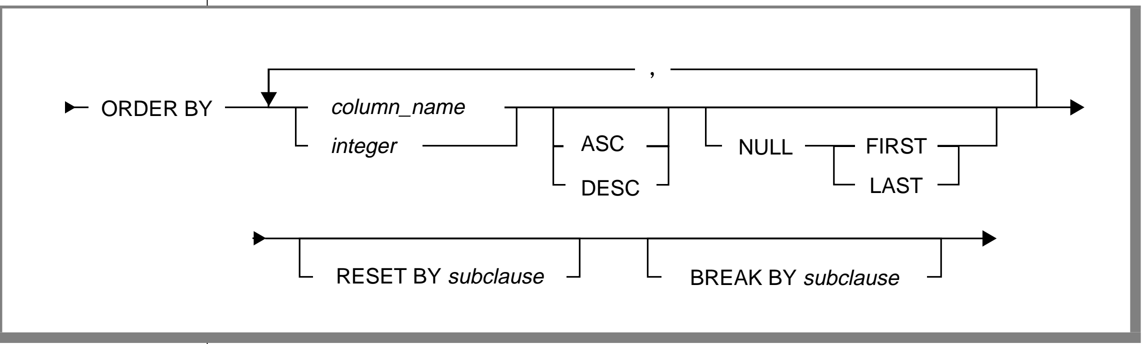
suppress_by_clause Eliminates rows from the final result set when the specified columns contain NULLs, spaces, or zeroes. For details, see "SUPPRESS BY Clause" on page 7-58.

ORDER BY Clause

The ORDER BY clause sorts the rows of the result table into ascending or descending order according to the values in specified columns.

Syntax

The following syntax diagram shows how to construct an *order_by_clause*:



column_name Specifies a column of the table used to sort the rows of the result table.

The column used to sort the results does not have to be displayed in the result table. If results are sorted by a column not in the select list, the sorting column is not displayed. Assuming that the *query_expression* that precedes the ORDER BY clause is a *query_specification*, you cannot sort by a column that is not in the select list if there are any columns in the GROUP BY clause that are not in the select list.

If the DISTINCT keyword is used in the SELECT clause to eliminate duplicate results, all of the columns in the ORDER BY list must occur in the select list.

If multiple columns are specified, they are placed in a nested sort order from left to right.

integer References a column in the select list. The integer value must be greater than zero and less than or equal to the number of columns in the result table. The integer specifies the kth column of the select list.

ASC Orders the values in ascending order. ASC is the default.

DESC Orders the values in descending order.

NULL By default, NULLs are usually evaluated as higher than the highest value in the collating sequence of the host operating system. Consequently, they occur at the end of an ascending sequence of values and at the beginning of a descending sequence of values. This default can be modified:

When the server is initialized from one of the server initialization files.

Before execution of the query with a SET ORDER BY command.

During execution of the query with the NULL subclause.

The server initialization files (*.rbwrc* files) that determine the original default placement of NULLs are described in the *Informix Red Brick Decision Server Administrator's Guide*. The SET ORDER BY command, which dynamically modifies default placement, is described on page 9-36.

Ordering Sequence

Rows are ordered as expected for the given column datatype—that is, numerically for numeric datatypes, alphabetically for character datatypes, and chronologically for datetime datatypes. For information about where NULLs occur in the ordering sequence, refer to the preceding discussion.

The server compares and sorts data according to the collation sequence or sort method specified by the server locale. If a different sort method is specified by the client, it has no effect. For information about defining the server locale, refer to the *Informix Red Brick Decision Server Administrator's Guide*.

Usage Notes

An ORDER BY clause that follows a *union_expression* must sort the final result set by a column named in the table that results from the first query expression. (This means that a column name assigned an alias in the first query expression must be referenced by that alias, and that qualified column names must not be used in ORDER BY clauses for *union_expressions*.)

An ORDER BY clause that follows a *union_expression* can contain a BREAK BY subclause but not a RESET BY subclause.

Examples

The following SELECT statement contains a joined table as its query expression to join the class and product tables and sort the resulting six-column table by package type.

```
class natural join product
order by pkg_type
```

```
CLASSKEY CLASS_TYPE CLASS_DESC PRODKEY PROD_NAME       PKG_TYPE
8        Gifts      Samplers,  31      Aroma Sounds Ca Aroma designer box
8        Gifts      Samplers,  30      Aroma Sounds CD Aroma designer box
8        Gifts      Samplers,  20      Easter Sampler  Gift box
8        Gifts      Samplers,  4       Coffee Sampler  Gift box
8        Gifts      Samplers,  10      Christmas Sampl Gift box
8        Gifts      Samplers,  5       Spice Sampler   Gift box
8        Gifts      Samplers,  3       Tea Sampler     Gift box
...
```

The following SELECT statement uses a union expression as its query expression. The results are ordered by Price. The column alias Order_Amounts (as defined in the first query expression's select list) is the name of the column and must be referenced in the ORDER BY clause. The ORDER BY clause could alternatively use the positional number 2, but it could not use Price.

```
select prod_name, price as order_amounts
from period natural join product natural join line_items
where year = 1998
    and qtr = 'Q1_98'
union
select prod_name, price
from product, line_items
where prod_name like 'Lotta%'
order by order_amounts
```

The following query returns the 1999 dollar sales in California stores of products packaged in bags. The result table is sorted by product and total sales for each city in descending order:

```
select prod_name, city, sum(dollars) as city_total
from sales natural join product
    natural join store
    natural join period
where year = 1999
    and state = 'CA'
    and pkg_type like '%bag%'
group by prod_name, city
order by prod_name, city_total desc
```

The following query returns the October 1999 dollars for products sold in Minneapolis. The query sorts the result table by Quantity in descending order. Note that the Quantity column is not displayed.

```
select prod_name, dollars
from sales natural join product
    natural join period
    natural join store
where city like 'Minn%'
    and year = 1999
    and month = 'OCT'
order by quantity desc

PROD_NAME                      DOLLARS
Colombiano                       330.75
Aroma Roma                       355.25
Colombiano                       330.75
Colombiano                       317.25
Aroma Roma                       333.50
La Antigua                       333.50
Veracruzano                      337.50
...
```

The following query returns the total sales and cumulative sales for Lotta Latte in each city and sorts the result table by region and city:

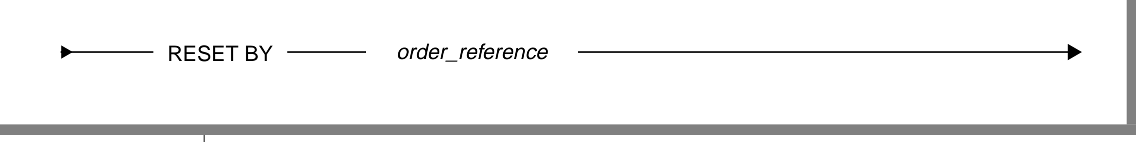
```
select city, region, sum(dollars) as total_dollars,
    cume(sum(dollars)) as running_total
from sales natural join product
    natural join store
    natural join market
where prod_name like 'Lotta%'
group by region, city
order by region, city
CITY            REGION     TOTAL_DOLLARS   RUNNING_TOTAL
Chicago           Central    32432.00        32432.00
Detroit           Central    28033.00        60465.00
Milwaukee         Central    31133.00        91598.00
Minneapolis       Central    18475.50        110073.50
Boston            North      26849.00        136922.50
Hartford          North      23896.50        160819.00
New York          North      35420.50        196239.50
Philadelphia      North      26848.50        223088.00
Atlanta           South      30390.00        253478.00
Houston           South      29766.00        283244.00
Miami             South      28056.00        311300.00
New Orleans       South      30878.50        342178.50
Cupertino         West       29908.00        372086.50
Los Angeles       West       27887.00        399973.50
Los Gatos         West       32369.50        432343.00
Phoenix           West       32813.50        465156.50
San Jose          West       68298.00        533454.50
```

RESET BY Subclause

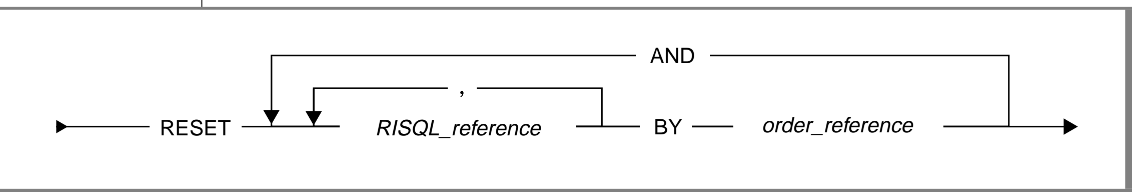
The RESET BY subclause re-initializes the computed value of a RISQL display function to zero according to control breaks specified on columns referenced in an ORDER BY clause.

Syntax

The following syntax diagram shows how to construct a simple RESET BY subclause:



The following syntax diagram shows how to construct a complex RESET BY subclause for a display function in the select list:



RESET BY Introduces a subclause that may only appear immediately after an ORDER BY clause.

order_reference Defines the control breaks by referencing a column in the ORDER BY clause. A control break occurs whenever the value of *order_reference* or any column preceding it in the ORDER BY clause changes.

The reference can be:

A column name in the ORDER BY clause.

An integer value that specifies the *k*th column in the select list. The integer must be greater than zero and less than or equal to the number of columns in the result table. This same column must be referenced in the ORDER BY clause.

RISQL_reference Refers to at least one display function in the select list of the query. The reference can be:

An integer value greater than zero and less than or equal to the number of columns in the intermediate result table. The integer specifies the *k*th column of the query; that column must contain a RISQL display function.

A column alias that references an expression in the select list that contains at least one RISQL display function.

Example

The following query returns a running total of 1999 sales for products sold in the Chicago district. The Prod_Total column is reset each time the product name changes, but the Run_Total column is not.

```
select prod_name, city, sum(dollars) as prod_dol,
    cume(sum(dollars)) as prod_total,
    cume(sum(dollars)) as run_total
from sales natural join product
    natural join period
    natural join store
    natural join market
where year = 1999
    and district like 'Chicago%'
group by prod_name, city
order by prod_name, city
    reset 4 by prod_name;
PROD_NAME        CITY           PROD_DOL   PROD_TOTAL RUN_TOTAL
Aroma Roma       Chicago         13188.50   13188.50   13188.50
Aroma Roma       Detroit         12820.75   26009.25   26009.25
Cafe Au Lait     Chicago         17394.50   17394.50   43403.75
Cafe Au Lait     Detroit         15737.50   33132.00   59141.25
Colombiano       Chicago         10544.25   10544.25   69685.50
Colombiano       Detroit         11104.75   21649.00   80790.25
Demitasse Ms     Chicago         16960.50   16960.50   97750.75
Demitasse Ms     Detroit         17264.00   34224.50  115014.75
...
```

Within product groups, the row order might vary.

RISQL display functions referenced in the WHEN clause are separate and distinct from those in the select list. RESET BY clauses operate only on the display functions referenced in the select list unless a column alias is used and the WHEN clause references that alias. The following example illustrates this scenario:

```
select city, prod_name, sum(dollars) as sales_00,
    rank(sum(-dollars)) as rank_00
from sales s, product p, store r, period d
where s.prodkey = p.prodkey
    and s.classkey = p.classkey
    and s.storekey = r.storekey
    and s.perkey = d.perkey
    and year = 2000
```

```
    and city in ('Atlanta', 'Boston', 'Phoenix')
group by city, prod_name
when rank_00 <= 2
order by city
reset rank_00 by city;

CITY PROD_NAME SALES_00 RANK_00
Atlanta Coffee Mug 55.00 1
Atlanta Aroma Sounds Cassette 136.00 2
Boston Special Tips 538.25 1
Boston Earl Grey 556.00 2
Phoenix Earl Grey 436.00 1
Phoenix Special Tips 714.50 2
```

If the expression

```
rank(sum(-dollars))
```

is specified in the WHEN clause, instead of the alias *rank_00*, the query returns only the two bottom-selling products in 2000 for all cities and shows their ranking within their city:

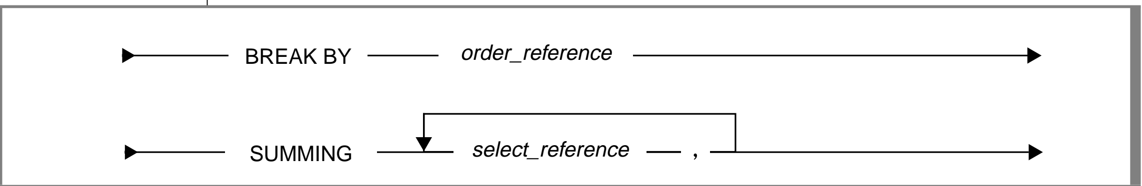
```
CITY          PROD_NAME             SALES_00      RANK_00
Atlanta        Coffee Mug                 55.00         1
Atlanta       Aroma Sounds Cassette      136.00         2
```

BREAK BY Subclause

The BREAK BY subclause computes a subtotal on a specified column each time a control break occurs.

Syntax

The following syntax diagram shows how to construct a *break_by* subclause:



The BREAK BY subclause inserts subtotal rows into the result table whenever the value of *order_reference* or any column preceding it in the ORDER BY clause changes (control break). The subtotal is the sum of values that precede a control break. This subclause also inserts a grand total row as the last row of the result table.

order_reference Specifies a column that is referenced in the ORDER BY clause. This reference can be:

A column name or alias.

An integer value greater than zero and less than or equal to the number of columns in the result table; the integer specifies the *k*th column of the select list.

SUMMING select_reference Specifies a numeric expression that occurs in the select list. This reference can be:

A column name or alias.

An integer value greater than zero and less than the number of columns in the result table. The integer specifies the *k*th column of the select list.

The *select_reference* column must have a numeric datatype so its values can be calculated into a subtotal.

Usage Notes

A SELECT statement that contains a BREAK BY clause cannot be used inside an INSERT statement.

When processing a BREAK BY subclause, the server:

- Returns a subtotal row whenever the value in the *order_reference* or any column preceding it in the ORDER BY clause changes.
- Returns a grand total row as the last row of the result table.

Subtotal rows have the same format as any other row of the result table but contain the following values:

- The current value in any control-break column.
- A NULL in all other columns.

Subtotal rows are not marked by a special identifier; their presence in a result table can be detected only when the value in the *order_reference* column or any column preceding it in the ORDER BY clause changes.

Because subtotal rows are not identified as such, they can be confused with base rows. To avoid any confusion, client applications that access a database through a communication gateway should detect and identify the subtotal rows. This identification might not be possible for SQL tools that display but do not format result tables.

Example

The following query returns the quarterly sales of Aroma Roma in San Jose for the first quarter of 1998 and 1999 and includes subtotals for each quarter in each year. (In the result set, the subtotal rows contain NULLs in the Month column, and the last row is the grand total row.)

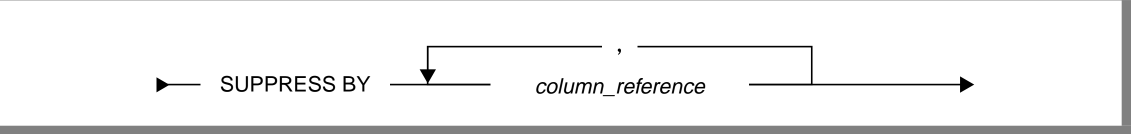
```
select month, qtr, sum(dollars) as dollars,
    sum(quantity) as qty
from sales natural join product
    natural join period
    natural join store
where prod_name like 'Aroma R%'
    and city like 'San J%'
    and year in (1998, 1999)
    and month in ('JAN', 'FEB', 'MAR')
group by month, qtr
order by qtr
break by qtr summing 3, 4
MONTH QTR   DOLLARS        QTY
JAN   Q1_98       1653.00    228
FEB   Q1_98       2495.50    350
MAR   Q1_98       1341.25    185
NULL  Q1_98       5489.75    763
JAN   Q1_99       1950.25    269
FEB   Q1_99       2022.75    279
MAR   Q1_99       3048.50    426
NULL  Q1_99       7021.50    974
NULL  NULL       12511.25   1737
```

SUPPRESS BY Clause

The SUPPRESS BY clause removes rows from the result table if the specified columns all contain NULLS, spaces, or zeroes. The rows are removed just prior to the computation of any RISQL display functions used in the query.

Syntax

The following syntax diagram shows how to construct a *suppress_by_ clause*:



column_reference Specifies a column by which to evaluate the result table for nulls, spaces, or zeros. Only column names that occur in the select list can be referenced from a SUPPRESS BY clause. The reference can be one of the following:

A positional number (integer).

A column name or alias.

A SUPPRESS BY clause cannot reference a RISQL display function.

Example

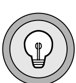
The following query removes all rows of the result table that have NULLs, zeroes, or spaces in both their Q198_Sales and Q198_Qty columns:

```
select store_name,
    sum(dollars) as q198_sales,
    sum(quantity) as q198_qty
from sales natural join store
    natural join period
where year = 1998
    and qtr = 'Q1_98'
group by store_name
suppress by 2, 3

STORE_NAMEQ198_SALESQ198_QTY
Beaches Brew57893.007452
Moroccan Moods44065.006323
Instant Coffee43129.506239
Roasters, Los Gatos43011.506114
Cupertino Coffee Supply44280.756459
Moulin Rouge Roasting44353.256517
The Coffee Club30962.253839
...
```

How SELECT Statements Are Processed

When a SELECT statement is issued, the server takes the following actions in logical order:

1. Retrieves 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. If a WHERE clause is not specified, the server retains the entire intermediate result table.



***Tip:** If the row size of the final result or any intermediate result is greater than the maximum size of a row, the SELECT statement fails. For more information about the maximum size of a row, refer to the "Informix Red Brick Decision Server Administrator's Guide."*

3. Divides the result table into groups specified in the GROUP BY clause.
4. Processes all set functions on specified groups or on the entire result table if no groupings were specified.

5. If a HAVING condition is present, retains all groups that satisfy that condition.
6. Removes rows according to the SUPPRESS BY clause, if present.
7. Orders the rows of the result table according to the ORDER BY clause, if present.
8. Processes all RISQL display functions (and computes values according to the RESET BY subclause).
9. Retains only those rows of the result table that satisfy the conditions specified in the WHEN clause.
10. Eliminates duplicate rows according to the use of the DISTINCT keyword (in the select list).
11. Creates BREAK BY rows. (The BREAK BY subclause is part of the ORDER BY clause.)

After completing these actions, the query returns the final result table to the user.

Tip: *Not all of these steps are available to a given query. For example, the RESET BY clause is not allowed if the query_expression is a union_expression.*

Subqueries

A subquery is a query expression enclosed in parentheses. Subqueries can be nested inside INSERT, DELETE, UPDATE, and SELECT statements or other query expressions to an arbitrary depth. The statement or expression that contains the subquery is called the subquery's *parent*. Typically, subqueries are used to derive a set of results that can be evaluated in conjunction with the result set of the parent query.

Several detailed examples of subqueries used in the select list, the FROM clause, and the WHERE clause are presented in the *SQL Self-Study Guide*.

Scalar Subqueries and Table Subqueries

According to the ANSI standard, subqueries fall into three categories: *scalar*, *row*, and *table*. Red Brick Decision Server supports scalar subqueries and table subqueries:

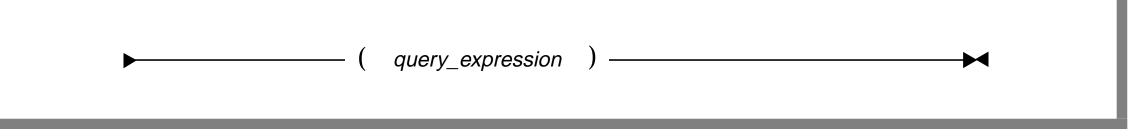
- A scalar subquery returns a single scalar value (one column, one row) and can occur either in a select list or in a *condition* as an argument of a comparison operator.
- A table subquery returns a result table of zero or more rows and can occur either in a FROM clause or in a *condition* as an argument of an EXISTS, IN, SOME, ANY, or ALL predicate.

  FROM clause subqueries and subqueries used as arguments to EXISTS predicates may consist of multiple columns as well as multiple rows. However, the select lists of subqueries used as arguments to IN, SOME, ANY, or ALL predicates are restricted to one column.

For the syntax of conditions and predicates, refer to Chapter 3, "Expressions and Conditions."

Syntax

The following syntax diagram shows how to construct a subquery:



query_expression Specifies any valid join or non-join query expression, as defined on page 7-5. The expression must be enclosed in parentheses.

Like other query expressions, subqueries cannot contain ORDER BY and SUPPRESS BY clauses.

The select list of the subquery is limited to one expression if the subquery is one of the following:

- A *scalar subquery*
- A *table subquery* used in a condition as an argument of a SOME, ANY, ALL, or IN predicate.

Correlations between queries and subqueries are explained in the example of a correlated subquery on page 7-64.

Examples

A subquery typically returns a set of values that provide input to the parent query. In the following statement, the subquery retrieves the names of all products packaged in gift boxes, and the parent query retrieves sales totals for these products in San Jose during 1999:

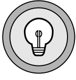
```
select prod_name, sum(dollars) as sales_99
from sales natural join product
    natural join period
    natural join store
where year = 1999
    and city like 'San J%'
    and prod_name in
        (select prod_name
        from product
        where pkg_type like 'Gift%')
group by prod_name

PROD_NAMESALES_99
Spice Sampler1860.00
Tea Sampler4207.00
Coffee Sampler3420.00
Christmas Sampler780.00
```

The previous subquery is a table subquery because it can return multiple rows. The following subquery is a scalar subquery that cannot return more than a single value; if it does, the server returns an error. This query places the December 1999 sales of products packaged in gift boxes next to the dollar sales of all products during the entire year:

```
select prod_name, sum(dollars) as sales_dec,
    (select sum(dollars)
    from sales natural join period
    where year = 1999
        and pkg_type like 'Gift%') as sales_99
from sales natural join product
    natural join period
where month = 'DEC'
    and year = 1999
    and pkg_type like 'Gift%'
group by prod_name

PROD_NAME                       GIFT_SALES_DEC  ALL_SALES_99
Tea Sampler                            1625.00      3279991.05
Coffee Sampler                         1140.00      3279991.05
Christmas Sampler                      1230.00      3279991.05
Spice Sampler                           384.00      3279991.05
```

The following example shows how a subquery can be placed in the FROM clause. The subquery returns a list of promotion descriptions and the sum of dollar sales for each one; the main query calculates the sum of those dollar amounts, not including the "No promotion" sales.

```
select sum(promo_sales)
from (select promo_desc, sum(dollars)
    as promo_sales from promotion natural join sales
    group by promo_desc) as promos
where promo_desc not like 'No%'
PROMO_DOLLARS
267296.40
```



Tip: There are several more detailed examples of FROM clause subqueries in the "SQL Self-Study Guide." In general, subqueries in the FROM clause run faster than equivalent correlated subqueries in the select list.

Correlated Subqueries

The subqueries in each of the previous examples need to be executed only once. A correlated subquery, however, contains cross-references to the parent query that can force the execution of the subquery each time the parent retrieves a new row. For example, a subquery that contains the following condition must be evaluated each time the parent retrieves a row:

```
parent.month = child.month
```

When the value referenced by the Parent.Month column changes, the condition itself changes and the subquery must be executed again.

Example

The following subquery compares daily sales of bulk Lotta Latte coffee beans at the San Jose Roasting Company in January of 1998 and 1999. The comparison is limited to non-promotional sales of this product.

```
select prod_name, substr(string(date),6,5) as date,
    dollars as sales_99,
    (select dollars
    from store st2 natural join sales sa2
        natural join product pr2
        natural join class cl2
        natural join period pe2
        natural join promotion po2
    where pe2.date = dateadd(year, -1, pe1.date)
        and pr2.prod_name = pr1.prod_name
        and cl2.class_type = cl1.class_type
        and st2.store_name = st1.store_name
        and po2.promo_desc = po1.promo_desc) as sales_98
from store st1 natural join sales sa1
    natural join product pr1
    natural join class cl1
    natural join period pe1
    natural join promotion po1
```

```
where year = 1999
    and month = 'JAN'
    and prod_name = 'Lotta Latte'
    and class_type = 'Bulk_beans'
    and promo_desc = 'No promotion'
    and store_name = 'San Jose Roasting Company'

PROD_NAME        DATE  SALES_99 SALES_98
Lotta Latte      01-03   368.00     NULL
Lotta Latte      01-06   256.00   248.00
Lotta Latte      01-09   168.00     NULL
Lotta Latte      01-18    96.00     NULL
```

The main query retrieves sales figures for days in January, 1999, and the subquery retrieves the corresponding figures for 1998 (using the same set of constraints except for the year). The query takes the following actions:

1. The parent retrieves a row that contains January, 1999, in its Date column; the row identifies the specified product, class type, date, and store name.
2. The subquery retrieves the corresponding row for January, 1998.
3. A single row is constructed for the result table that contains the corresponding information for 1999 and 1998.

Qualified Column Names

In the previous example, both the main query and the subquery assign correlation names to the tables they reference. For example, the Store table is assigned the name *st1* in the main query and *st2* in the subquery. These correlation names are then used to qualify the column names specified in each correlation condition:

```
pe2.date = dateadd(year, -1, pe1.date)
pr2.prod_name = pr1.prod_name
cl2.class_type = cl1.class_type
st2.store_name = st1.store_name
po2.promo_desc = po1.promo_desc
```

When a subquery references columns defined in the parent query, it is recommended that all column names be qualified. This approach ensures that there is no ambiguity and that column names are correctly "resolved," as discussed in the following section. Unless there are specific reasons for not qualifying column names, they should be qualified.

Column Name Resolution

The server resolves *unqualified* column name references by searching the column aliases specified in the subquery's select list, then the tables in the subquery's FROM clause:

- If the column name is found in the select list, the search terminates successfully.
- If the column name is found in exactly one table, the search terminates successfully.
- If the column name is found in more than one table, the server returns an error.
- If the column name is not found, the server searches the column aliases specified in the parent query's select list, then the parent query's FROM clause. If the column name is found in exactly one table, the search terminates successfully. If the name is still not found, and the parent is the child of another query, the search continues until the reference is resolved.

Because of this approach to column name resolution, queries might return unexpected results when a column name is not explicitly qualified in a set of nested subqueries.

The server resolves *qualified* column name references by searching the tables specified in the FROM clause. The *closest* query specification containing the qualifier is used.

The rules for column name resolution apply to all types of subqueries.

Example

This example illustrates column name resolution. The *col_a* column exists in the *table_1* table but not in *table_2*. To find the *col_a* column referenced in the subquery, the server searches *table_2* first. Because *col_a* does not exist in *table_2*, the server searches the tables listed in the parent query. The *col_a* column is found in *table_1* listed in the FROM clause of the parent query and the query is processed correctly.

```
select col_a
from table_1
where col_a in
    (select col_a
    from table_2)
```

However, if *col_a* exists in both *table_1* and *table_2*, the subquery will select *col_a* from *table_2* unless the column reference is qualified:

```
(select table_1.col_a from table_2)
```

The preceding example is included only to illustrate the concept of column name resolution. This approach to naming columns and tables is not recommended.

Groups of Rows

When a parent query that contains set functions in its select list also contains a correlated subquery, the correlation columns must be included in the parent query's GROUP BY clause. This means that the parent query's GROUP BY clause will contain column names from the parent query's select list as well as correlation column names from the subquery, as shown in the following example.

This rule does not apply to correlated subqueries that occur in the parent query's WHERE clause.

Example

The following query compares the monthly sales of Lotta Latte in San Jose during the first quarter of 1998 and 1999:

```
select pr1.prod_name, pe1.month, sum(sa1.dollars) as
sales_99,
    (select sum(sa2.dollars)
    from store st2 natural join sales sa2
        natural join product pr2
        natural join period pe2
    where pe2.month = pe1.month
        and pe2.year = pe1.year-1
        and pr2.prod_name = pr1.prod_name
        and st2.city = st1.city) as sales_98
from store st1 natural join sales sa1
    natural join product pr1
    natural join period pe1
where year = 1999
    and qtr = 'Q1_99'
    and prod_name = 'Lotta Latte'
    and city = 'San Jose'
group by pe1.month, pe1.year, pr1.prod_name, st1.city

PROD_NAME        MONTH SALES_99 SALES_98
Lotta Latte      JAN    1611.00  3195.00
Lotta Latte      FEB    3162.50  4239.50
Lotta Latte      MAR    2561.50  2980.50
```

The correlation conditions of the subquery reference columns that must also occur in the parent’s GROUP BY clause:

```
pe2.month = pe1.month
pe2.year = pe1.year-1
pr2.prod_name = pr1.prod_name
st2.city = st1.city
```

Chapter

8

SQL Commands and RISQL Extensions

In This Chapter 8-5

ALTER DATABASE 8-6

ALTER INDEX 8-14

ALTER MACRO 8-19

ALTER ROLE. 8-21

ALTER SEGMENT 8-22
 ALTER SEGMENT—ATTACH Clause 8-23
 ALTER SEGMENT—Other Clauses 8-30

ALTER SYNONYM. 8-50

ALTER SYSTEM 8-51
 Alter User Activity Specification 8-55
 Alter User Priority Specification. 8-58
 Alter Logging Specification 8-59
 Alter Accounting Specification 8-61

ALTER TABLE 8-62
 Add Column Specification 8-67
 Drop Column Specification. 8-70
 Alter Column Specification. 8-72
 Add Constraint Specification 8-75
 Drop Constraint Specification 8-79
 Alter Constraint Specification 8-80

ALTER USER 8-83

ALTER VIEW 8-84

CHECK INDEX 8-86

CHECK TABLE 8-89

CREATE HIERARCHY 8-92

CREATE INDEX 8-96
Index Specifier 8-102
Segment Specification. 8-106
Segment Range Specification 8-109
B-TREE and TARGET Indexes 8-109
STAR Indexes 8-111

CREATE MACRO 8-118

CREATE ROLE 8-125

CREATE SEGMENT 8-126

CREATE SYNONYM 8-131

CREATE TABLE 8-132
Column Definitions 8-135
Primary-Key Reference 8-139
Foreign-Key Reference 8-141
Primary-Key and Foreign-Key Constraint Names 8-144
Segment Specification. 8-145
Segment Range Specification 8-149
More CREATE TABLE Examples 8-153

CREATE TEMPORARY TABLE. 8-154

CREATE VIEW 8-157
Usage Notes for Precomputed Views 8-163

DELETE. 8-166

DROP HIERARCHY 8-171

DROP INDEX. 8-172
Dropping System-Generated B-TREE Indexes 8-174

DROP MACRO 8-175
DROP ROLE. 8-177
DROP SEGMENT 8-178
DROP SYNONYM 8-180
DROP TABLE 8-181
DROP VIEW. 8-183
EXPAND 8-184
EXPLAIN. 8-185
EXPORT 8-187
GRANT Authorization and Role 8-192
GRANT CONNECT 8-198
GRANT Privilege 8-202
INSERT 8-205
LOCK Table 8-212
LOCK DATABASE 8-214
REVOKE Authorization and Role 8-216
REVOKE CONNECT. 8-218
REVOKE Privilege 8-219
SELECT 8-222
UNLOCK Table. 8-222
UNLOCK DATABASE 8-223
UPDATE 8-224

In This Chapter

The Structured Query Language (SQL) contains data control, data definition, and data manipulation commands. This chapter serves as a convenient reference by describing these commands, as well as the RISQL extensions to SQL, in alphabetical order.

This chapter describes commands that:

- Alter tables, indexes, and segments.
- Create and drop database objects.
- Modify database rows.
- Grant and revoke database authorizations and table privileges.
- Lock and unlock tables and databases.
- Control database and user activity.

SET commands are documented separately in Chapter 9.

For detailed information about SELECT statements, refer to Chapter 7, "Query Expressions."

ALTER DATABASE

The ALTER DATABASE command has the following purposes:

- To specify a segment as the backup segment.
- To drop the backup data stored in that segment.
- To create a version log in a segment.
- To drop the version log.
- To start and stop the processing of versioning transactions.
- To start and stop the vacuum cleaner daemon or thread for the database.
- To freeze a revision number to be read by subsequent queries.
- To unfreeze a revision number, allowing queries to read the latest revision.
- To end a database session.

Only one segment per database can be defined as the backup segment. If no backup segment is defined, SQL-BackTrack backup operations cannot be performed.

Only one segment per database can be defined as the version log. If no version log is defined, versioning transactions cannot be performed.

The backup-related components of the ALTER DATABASE command are available only for Red Brick Decision Server installations that have SQL-BackTrack enabled with a license key.

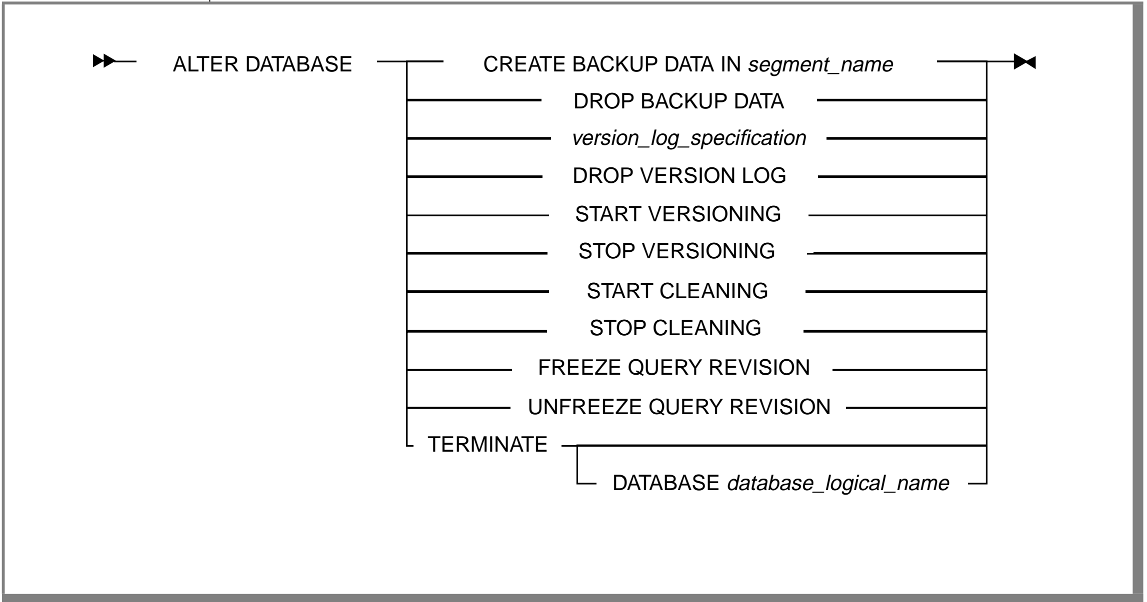
Authorization

To issue the ALTER DATABASE command, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have BACKUP_DATABASE and RESTORE_DATABASE authorization, either explicitly or through membership in a user-created role.
- Have ALTER_SYSTEM authorization, either explicitly or through membership in a user-created role (to perform version log operations).

Syntax

The following diagram shows how to construct an ALTER DATABASE statement:



CREATE BACKUP DATA IN segment_name	Names an existing but unused segment as the backup segment for the database. The segment is created in the usual way with a CREATE SEGMENT statement and can consist of multiple PSUs. When the command is issued, the named segment is marked as the backup segment in the DST_DATABASES table. For information about the CREATE SEGMENT command, refer to "CREATE SEGMENT" on page 8-126.
DROP BACKUP DATA	Removes the backup data (bitmap information) from the database and changes the backup segment to a regular segment; the segment itself is not dropped. When this command has been issued, backup and recovery operations can no longer be performed. Do not drop the backup data unless you no longer intend to perform backup and recovery operations on the database. Databases cannot be recovered to a checkpoint backup that used a backup segment that has since been dropped and re-created. Database recoveries do not work across multiple instances of the backup segment.

version_log_specification

Includes the CREATE VERSION LOG statement, which creates a version log for the database using the disk space in the named, unused segment. (The syntax diagram follows this description.)

A database can have only one version log. The version log must be created before versioning transactions can be enabled for the database. All of the disk space for the version log is allocated by the CREATE VERSION LOG statement; therefore, this operation might take some time (all of the PSUs in the segment are extended to their MAXSIZE values before the operation completes).

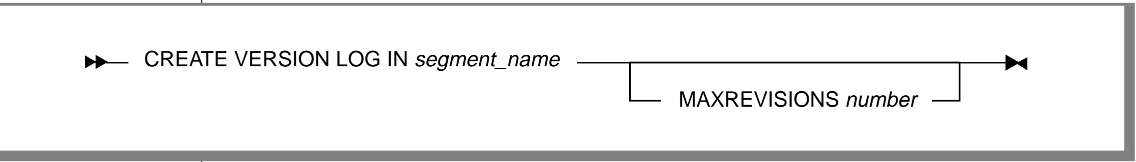
No other users can be connected to the database during the CREATE VERSION LOG operation; all other connections are refused after the operation begins.

If an error occurs during the CREATE VERSION LOG operation, drop the version log and run the operation again.

After creating the version log, you must start versioning (ALTER DATABASE START VERSIONING) before the database accepts versioning transactions.

Creating the version log starts the vacuum cleaner daemon or thread for the database.

The following syntax diagram shows how to construct a *version_log_specification*. In the diagram, MAXREVISIONS specifies the maximum number of versions of the database allowed in the version log at a given time. The default is 5000.



DROP VERSION LOG	Terminates the vacuum cleaner, releases the versioning shared resources, and detaches the segment containing the version log. The disk space used by the segment is not released. This behavior differs from the DROP TABLE statement, which shrinks the PSUs back to their *initsize* values. If versioning is still active or the version log still contains active data that has not yet been processed by the vacuum cleaner daemon or thread, no action is taken and an error occurs. You must stop versioning with an ALTER DATABASE STOP VERSIONING command and wait for the vacuum cleaner to finish cleaning the version log before you can drop the version log. The version log is empty when the value of the CURRENT_REVISION column in the DST_DATABASES table is equal to the value of the LATEST_MERGED_REVISION column. It is necessary to drop the version log before performing an UPGRADE operation. No other user can be connected to the database during a DROP VERSION LOG operation.
START VERSIONING	Allows new versioning transactions to be processed by the database. It is only necessary to start versioning if versioning was explicitly stopped or if the version log was just created for the database. The version log must exist in order to execute this command.

STOP VERSIONING	Stops the database from accepting new versioning transactions. After performing an ALTER DATABASE STOP VERSIONING operation, new transactions that attempt to modify the database versioning will fail with an error (as opposed to automatically becoming blocking transactions). The versioning state of a database remains persistent after a restart of Red Brick Decision Server. Therefore, if the *rbwapid* daemon (UNIX systems) or the Red Brick Decision Server service (Windows NT systems) is shut down while versioning is started for a database, versioning is restarted when the daemon or service is restarted. Similarly, if versioning is stopped for a database when the *rbwapid* daemon or Red Brick Decision Server service is shut down, versioning for that database remains stopped when the daemon or service is restarted.
START CLEANING	Starts the vacuum cleaner daemon or thread for the database if it is not already running.
STOP CLEANING	Stops the vacuum cleaner daemon or thread for the database if it is running. If the *rbwapid* daemon (UNIX systems) or the Red Brick Decision Server service (Windows NT systems) is shut down while the vacuum cleaner is stopped, the vacuum cleaner is automatically restarted when the daemon or service is restarted.
FREEZE QUERY REVISION	Reads the current revision number and sets that number as the default read revision number to be used by all subsequent queries, unless specifically overridden in a session. After the statement has completed successfully, it displays the revision number chosen to be the query revision. If the query revision is already frozen, the command fails with an error message.

UNFREEZE QUERY REVISION	Undoes the effect of the ALTER DATABASE FREEZE QUERY REVISION command. Any new query reads the latest revision, and all the other restrictions that are placed on the system are removed. If query revision is not frozen, the command fails with an error message.
TERMINATE	Ends the database session and releases all the shared resources for that database. If other users are connected to the database, the command fails with an error.
DATABASE ***database_logical_name***	If you run the ALTER DATABASE TERMINATE command from the ADMIN database, you must specify the DATABASE ***database_logical_name*** clause to indicate from which database the shared resources are released.

Usage Notes

The following usage notes apply to the ALTER DATABASE command.

Notes for the Backup Segment

The backup segment does not need to be offline for the ALTER DATABASE command to work.

You cannot use a DROP SEGMENT statement to remove the backup segment from the database. However, after an ALTER DATABASE DROP BACKUP DATA statement has been issued, the segment is a regular segment and can be dropped.

Notes for the Version Log

Extending the PSUs in the segment specified for the version log can take some time. The system administrator is advised to use care when setting the MAXSIZE parameters on the PSUs of this segment as they define the amount of disk storage that is immediately allocated and assigned to the version log.

The location of the version log can affect performance. If the version log is heavily used (for example, if your LOAD operations change a large number of blocks in the database), access to the version log can become a bottleneck to system performance; therefore, it is best to place the version log on high-performance storage such as is provided by RAID level 0 or level 1. To ensure the best performance, the devices in which the version log resides should be independent of the devices in which your database files reside. Because good write performance is needed, higher RAID levels are not recommended.

Because the version log might contain data from any number of segments in the database, loss of the version log can potentially require the entire database to be recovered from backup. To safeguard against this, the version log should be placed on reliable storage such as is provided by RAID level 0.

Examples

The following SQL statements show how to create a segment, then define it as the backup segment.

1. Create the segment:

```
create segment sqlbacktrack_seg
storage '/test/bt1' maxsize 1024,
storage '/test/bt2' maxsize 1024,
storage '/test/bt3' maxsize 1024,
storage '/test/bt4' maxsize 1024;
```

2. Define the segment as the backup segment:

```
alter database create backup data in sqlbacktrack_seg;
```

The following SQL statements show how to create a segment, then define it as the version log.

1. Create the segment:

```
create segment versionlog_seg
storage '/test/vl1' maxsize 1024,
storage '/test/vl2' maxsize 1024,
storage '/test/vl3' maxsize 1024,
storage '/test/vl4' maxsize 1024;
```

2. Define the segment as the version log:

```
alter database create version log in versionlog_seg;
```

   This command allocates all of the physical space for the version log, so it might take some time to complete.

3. Start versioning for the database:

```
alter database start versioning;
```

ALTER INDEX

The ALTER INDEX command serves three purposes:

- To change the fill factor of an index. The fill factor setting determines the amount of space used in each index node after data is loaded.
- To change the optional domain size specification for TARGET indexes. (Based on the domain size, the appropriate storage method, or "representation," is selected for the TARGET index information. If no domain size is specified, the server automatically chooses the appropriate representation.)
- To assign descriptive comments to an index, which are then stored in the RBW_INDEXES system table.

Authorization

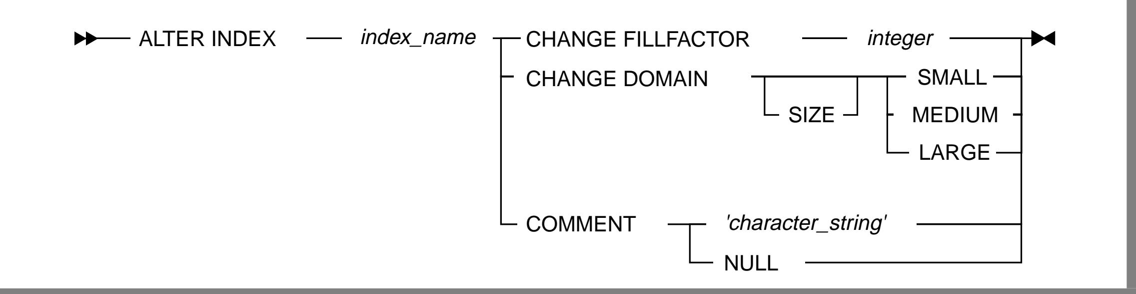
To alter an index, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have ALTER_ANY authorization, either explicitly or through membership in a user-created role.

- Be a member of the RESOURCE system role and the creator of the index.
- Be the creator of the index and have ALTER_OWN authorization, either explicitly or through membership in a user-created role.

Syntax

The following syntax diagram shows how to create an ALTER INDEX statement:



index_name Specifies the name of the index to be altered. Fill factors and comments for both user-created and system-generated indexes can be changed. Domain sizes can be changed only for user-created TARGET indexes. To change the fill factor, domain size, or comment of a user-created index, you must use the index name specified in the CREATE INDEX statement. To change the fill factor or comment of a system-generated index, you must use the index name generated by the system.

To determine the name of a system-generated index, check the RBW_INDEXES table or use the following naming conventions:

- A primary key index is named by adding the string *_PK_IDX* to the table name on which the index was generated. For example, the primary key index on the Market table is named:

  ```
  MARKET_PK_IDX
  ```

- An index name can be no longer than 128 characters. If the table name and index string combined results in an index name that is longer than 128 characters, the table name is truncated.

An index name must be unique. If the table name and index string combined results in a non-unique name, a number from 001 to 999 is added to the end of the index name until a unique name is produced.

CHANGE FILLFACTOR integer	Specifies a new setting for the percentage of space to fill in each new index node when data is initially loaded into the table or when the index is rebuilt as a result of a REORG operation. As rows are later inserted, the index nodes continue to fill until they reach 100 percent of capacity. If the index nodes need to fill beyond 100 percent, they split to accommodate the overflow. Legal values for *integer* can range from 1 to 100; however, fill factors should generally be greater than 50 percent. For information about loading data, reorganizing data, turning on the optimize option, and setting the fill factor, refer to the *Administrator's Guide*.
CHANGE DOMAIN SIZE SMALL, MEDIUM, LARGE	Specifies a new DOMAIN setting for the named TARGET index; you cannot set or change the domain size for any other type of index. For detailed information about choosing a domain size, which is optional, see "CREATE INDEX" on page 8-96. Although you can change the domain size of a TARGET index, you cannot remove the domain size specification altogether (that is, change the index to a mixed-domain TARGET index). The SIZE keyword is optional.
COMMENT	Assigns a descriptive comment string to the index, which is stored in the RBW_INDEXES system table. A comment can contain up to 256 bytes. The server makes no use of the comment text. Specifying NULL replaces the comment string with NULL.

Usage Notes

Fill Factors The fill factor of an index is originally set upon index creation. For user-created indexes, the fill factor can be set with the CREATE INDEX...WITH FILLFACTOR command.

For system-generated primary-key indexes, the fill factor is automatically set with the default value specified in the FILLFACTOR PI parameter in the *rbw.config* file. The original default specified for these indexes is 100 percent.

You can specify the fill factor of a system-created index in two ways:

- Before creating the table, change the default, if necessary, in the *rbw.config* file.
- Create the table, then alter the index with the ALTER INDEX command using the system-created index name.

For TARGET indexes with SMALL domain sizes, keep the fill factor set to 100 percent (the default). For indexes with MEDIUM or LARGE domain sizes, use 100 percent unless you plan to update or delete rows; in this case, use a lower percentage.

Domain Sizes An ALTER INDEX statement that changes the domain size of a TARGET index marks the index invalid until a REORG operation is performed; therefore, if you are changing the domain of one index and intend to immediately rebuild it, you might prefer to simply drop and re-create the index.

On the other hand, you can rebuild several indexes at once with a REORG operation, which consumes fewer resources than dropping and re-creating each index individually. Queries can still be issued against a table that has an invalid index if the column with the invalid index is not constrained. Therefore, you might be able to postpone the REORG operation until a more convenient time.

Examples

The following statement sets the fill factor of the primary key index of the **market** table to 75 percent:

```
alter index market_pk_idx change fillfactor 75;
```

The following statement alters the domain size of a TARGET index:

```
alter index tgt_idx1 change domain large;
```

ALTER MACRO

The ALTER MACRO command assigns a descriptive comment to a macro, which is stored in the RBW_MACROS system table.

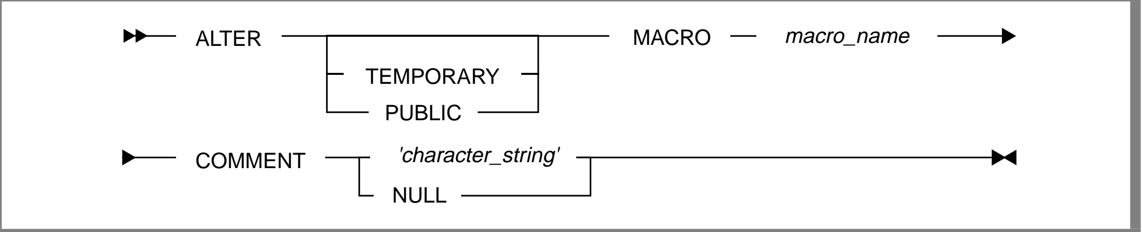
Authorization

To alter a macro, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have ALTER_ANY authorization, either explicitly or through membership in a user-created role.
- Be a member of the RESOURCE system role and be the creator of the macro.
- Be the creator of the macro and have ALTER_OWN authorization, either explicitly or through membership in a user-created role.

Syntax

The following syntax diagram shows how to create an ALTER MACRO statement:



TEMPORARY, PUBLIC, Private — Specifies the type of macro to be altered. If neither TEMPORARY or PUBLIC is specified, a private macro is altered.

macro_name — Specifies the name of the macro to be altered.

COMMENT — Assigns a descriptive comment string to the macro, which is stored in the RBW_MACROS system table. A comment can contain up to 256 bytes. The server makes no use of the comment text.

Specifying NULL replaces the comment string with NULL.

ALTER ROLE

The ALTER ROLE command assigns a descriptive comment to a user-created role, which is stored in the RBW_ROLES and RBW_USERAUTH system tables.

Authorization

To issue an ALTER ROLE command, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have the ROLE_MANAGEMENT task authorization.

Syntax

The following syntax diagram shows how to create an ALTER ROLE statement:

role_name Specifies the name of the role to be altered.

COMMENT Assigns a descriptive comment string to the role, which is stored in the RBW_ROLES and RBW_USERAUTH system tables. A comment can contain up to 256 bytes. The server makes no use of the comment text.

Specifying NULL replaces the comment string with NULL.

ALTER SEGMENT

The ALTER SEGMENT command modifies a segment. This command can be used to:

- Attach a segment to any table or index.*
- Detach a segment from any table or index.*
- Verify a segment to determine if it is damaged (or mark a segment as intact if a segment is known to have undamaged PSUs).
- Modify a segment by:
  - Specifying a segmenting column.*
  - Changing the range specification for the segment.*
  - Taking the segment offline.*
  - Bringing the segment online.*
  - Clearing the segment of all data.*
  - Renaming the segment.
  - Changing the maximum size of a physical storage unit (PSU) in the segment.
  - Changing the extend size of a PSU in the segment.
  - Changing the location/path of a PSU in the segment.
  - Move an entire segment from one location to another.
  - Assigning a comment to the segment, which is stored in the RBW_SEGMENTS system table.
  - Adding a new PSU to the segment.

* These features are not available in Red Brick Decision Server for Workgroups databases, nor are they available for use with the backup segment or version log segment. For more information about the backup segment and the version log segment, refer to "ALTER DATABASE" on page 8-6.

For information about attaching a segment, refer to "ALTER SEGMENT—ATTACH Clause" on page -23. For information about detaching, verifying, or otherwise modifying a segment, refer to "ALTER SEGMENT—Other Clauses" on page -30.

Authorization

To use the ALTER SEGMENT command, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have ALTER_ANY authorization, either explicitly or through membership in a user-created role.
- Be a member of the RESOURCE system role and be the creator of the segment and any affected tables.
- Be the creator of the segment and any affected tables and have ALTER_OWN authorization, either explicitly or through membership in a user-created role.

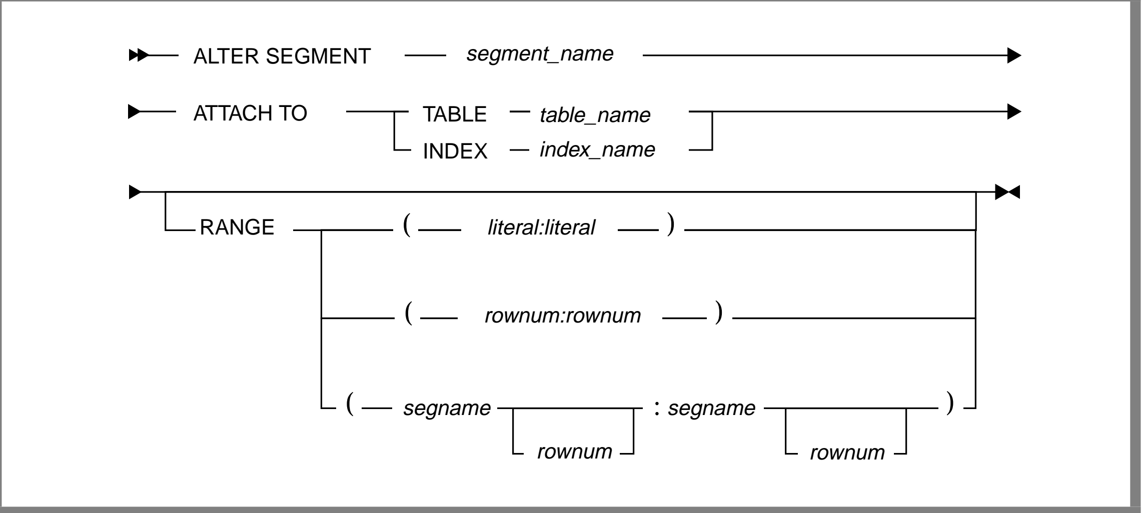
ALTER SEGMENT—ATTACH Clause

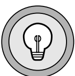
This section describes how to attach a segment to a table or index. This clause cannot be used in Red Brick Decision Server for Workgroups databases nor can it be used on the backup segment.

To detach, verify, or otherwise modify a segment, refer to page 8-30.

Syntax

The following syntax diagram shows how to construct an ALTER SEGMENT statement to attach a segment to a table or index:





***Tip:** In all the variations of the RANGE specification, MIN and MAX are also valid values on the left and right sides of the colon, respectively.*

For example, you can specify ranges such as:

```
(min:literal)
(literal:max)
(min:rownum)
(segname rownum:max)
```

segment_name Specifies the name of an existing segment to be attached to the table or index. A segment must be created with the CREATE SEGMENT command before it can be attached.

ATTACH TO TABLE, INDEX Attaches a segment to a table or an index. To attach a segment, specify a range of values based on the segmenting column. The segmenting column is assigned with the CREATE TABLE, CREATE INDEX, or ALTER SEGMENT command.

After the attach operation has completed, the segment is automatically set to ONLINE mode. If necessary, the segment can be taken offline while it is still attached.

Before using an ALTER SEGMENT statement to attach a segment to a table, note the following restrictions:

A segment cannot be attached to a table if attaching it would cause the table's MAXSEGMENTS value to be exceeded.

Segments cannot be attached to tables whose data is distributed among segments by hash values. For information about distribution by hash values, refer to page 8-148.

RANGE Specifies the values in the segmenting column to be stored in the newly attached segment. The following restrictions apply to range specifications:

A segment must be attached within an existing segment of the table or index; therefore, one end of the range for the new segment must coincide with the boundary of an existing segment and the range of the new segment must not span any other existing boundaries.

The range for the attached segment cannot span any data values that are already stored in the table; this would require the data to move to the new segment, which is not possible. If the new segment is attached to an index, it can span existing index values but the index is subsequently marked as invalid and must be rebuilt.

If only one segment has been attached to a table or an index, its range is (*min:max*). To attach a second segment, the range in the ALTER SEGMENT statement must be specified with either *min* or *max* as one of the boundaries.

literal:literal Literal range values must be used for segments of tables, B-TREE indexes, and TARGET indexes. These values are based on the datatype of the segmenting column. For example, if the segmenting column is an INTEGER, the range must be between –2,147,483,648 and 2,147,483,647. If the segmenting column is of character datatype, the range must be specified with character values.

rownum: rownum, segname rownum: segname rownum	The segment range of a STAR index is based on row IDs of the table referenced by the segmenting column:

If the referenced table resides in a single segment, the *rownum:rownum* format must be used, where *rownum* identifies a row within that segment.

If the referenced table resides in multiple segments, the *segname rownum:segname rownum* format must be used, where *segname* identifies a segment attached to the referenced table and *rownum* is optional. If *rownum* is omitted, the minimum row number (*min*) for the named segment is assumed.

Row numbers start at 0 and cannot exceed or equal the value of MAXROWS PER SEGMENT. To specify the range of a STAR index manually, determine the ROW numbers and segment names of the table that contains the segmenting column by issuing the following query:

select *primary_key*, rbw_segname, rbw_rownum
from *table_name*

The above query returns all the rows of the table. If the table is large, the result set will be large.

The RANGE specification is optional for STAR index segments. If it is omitted, the range is calculated automatically based on the total number of segments in the STAR index and the MAXSEGMENTS and MAXROWS PER SEGMENT values defined for the referenced table. In this case, the STAR index must be reorganized with the REORG command. For information about REORG operations, refer to the *Table Management Utility Reference Guide.*

Example

Suppose the Sales table was created with data distributed among four segments based on values of the Mktkey column:

```
create table sales
...
data in (data1, data2, data3, data4)
    segment by values of (mktkey)
    ranges (min:500, 500:1000, 1000:3000, 3000:max)
```

An existing segment, *data5*, can be attached to the Sales table with the following range:

```
alter segment data5
    attach to table sales
    range (1000:1500)
```

The above statement is accepted because the following conditions are true:

- The minimum value of the new range (1000) is an existing boundary between the *data2* and *data3* segments.
- The maximum value of the new range (1500) is within the range of the *data3* segment, which has an upper boundary of 3000.
- No rows exist in the Sales table whose values for the Mktkey column fall between 1000 and1499, inclusive.

  Figure 8-1 shows the new segment ranges of the Sales table:

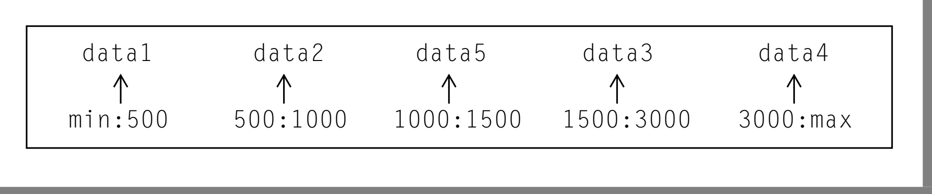


Figure 8-1
New Segment Ranges for Sales Table

Assume there is a STAR index built on the Sales table, which references the Market table and is stored in multiple segments:

```
create star index sales_star
on sales (mktkey)
in (ix_seg1, ix_seg2, ix_seg3, ix_seg4)
segment by references of (mktkey)
ranges (min:tab_seg1 500, tab_seg1 500:tab_seg2 1000,
    tab_seg2 1000:tab_seg3 3000, tab_seg3 3000:max)
```

To attach another segment (*ix_seg5*) to the index, you could issue the following ALTER SEGMENT statement:

```
alter segment ix_seg5
attach to index sales_star
range (tab_seg2 1000:tab_seg3 1500)
```

The above statement is accepted because the following conditions are true:

- The range specification uses the *segname rownum* format, as required for STAR indexes that reference multi-segment tables.
- The minimum value of the new range (*tab_seg2 1000*) is an existing boundary between the *ix_seg2* and *ix_seg3* segments.
- The maximum value of the new range (*tab_seg3 1500*) is within the range of the *ix_seg3* segment, which has an upper boundary of 3000.

The statement will be accepted regardless of the existence of data that needs to be moved from the old segment to the new one. If such data exists, the index will be marked invalid and will need to be reorganized with the TMU REORG command.

Example

Assume that only one named segment (*seg_sales1*) has been attached to the Sales table and a segmenting column has been specified; the segment has a range of (*min:max*). To attach the *seg_sales2* segment to the Sales table, either of the following ALTER SEGMENT statements can be issued:

```
alter segment seg_sales2
    attach to table sales
    range (500:max)
alter segment seg_sales2
    attach to table sales
    range (min:500)
```

The new ranges will be (*min:500, 500:max*).

ALTER SEGMENT—Other Clauses

This section describes how to modify an attached or unattached segment. (To attach a segment, refer to "ALTER SEGMENT—ATTACH Clause" on page 8-23.)

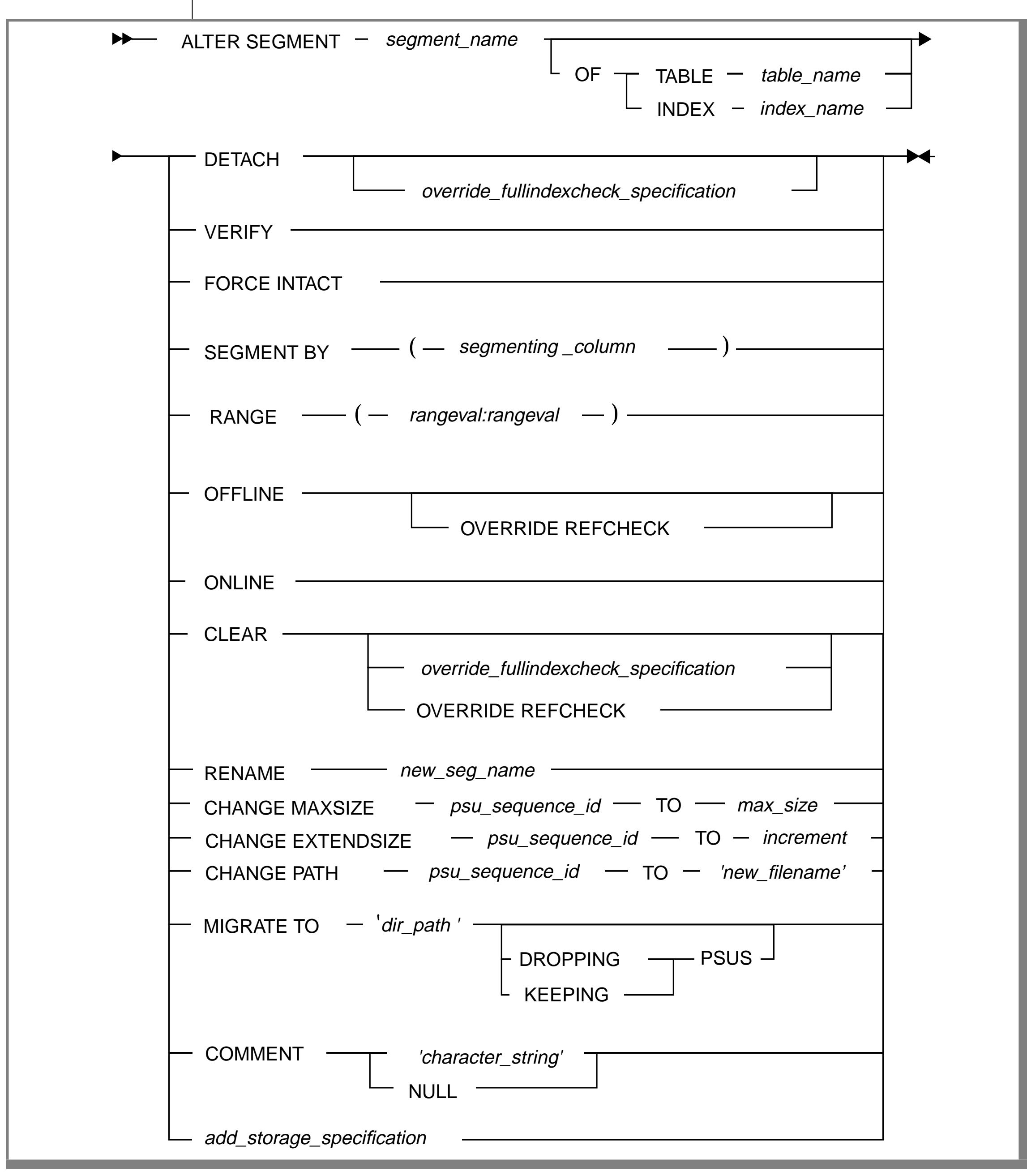
The following clauses cannot be used in Red Brick Decision Server for Workgroups databases, nor can they be used on the backup segment or the version log segment:

- DETACH
- SEGMENT BY
- RANGE
- OFFLINE
- ONLINE
- CLEAR

For more information about the backup segment and the version log segment, refer to "ALTER DATABASE" on page 8-6. For more information about the Informix Red Brick SQL-BackTrack system, refer to the *Informix Red Brick SQL-BackTrack User's Guide*.

Syntax

The following syntax diagram shows how to construct an ALTER SEGMENT statement to modify a segment.



ALTER SEGMENT
segment_name
OF
TABLE
table_name
INDEX
index_name
DETACH
override_fullindexcheck_specification
VERIFY
FORCE INTACT
SEGMENT BY
(
segmenting _column
)
RANGE
(
rangeval:rangeval
)
OFFLINE
OVERRIDE REFCHECK
ONLINE
CLEAR
override_fullindexcheck_specification
OVERRIDE REFCHECK
RENAME
new_seg_name
CHANGE MAXSIZE
psu_sequence_id
TO
max_size
CHANGE EXTENDSIZE
psu_sequence_id
TO
increment
CHANGE PATH
psu_sequence_id
TO
'new_filename'
MIGRATE TO
'dir_path '
DROPPING
KEEPING
PSUS
COMMENT
'character_string'
NULL
add_storage_specification

segment_name Specifies the name of the segment to be modified.

table_name Specifies the name of the table to which the segment is attached. If the segment is attached to a table, this clause must be specified. If the segment is not attached to a table or index, this clause must not be specified.

index_name Specifies the name of the index to which the segment is attached. If the segment is attached to an index, this clause must be specified. If the segment is not attached to an index or table, this clause must not be specified.

DETACH Removes the segment from the specified table or index, deleting all row data or index data residing in the segment. A segment must be set to OFFLINE mode before it can be detached. After a segment has been detached, a separate ALTER SEGMENT command can be issued to re-attach it to the table or index, or to attach it to a different table or index.

If the detached segment belonged to an index, in most cases the index is marked invalid and must be reorganized with the REORG command. However, the index is not invalidated if the data and index for a table are segmented identically, and the corresponding data segment has already been detached, leaving the index segment empty.

If a default segment is detached, both the data in the segment and the segment itself are deleted. If a named segment is detached, the data in the segment is deleted but the segment itself is not deleted. The named segment remains available to be attached to a table or index.

The range of a detached segment is merged into the ranges of the remaining segments in the table or index. The range of the lower neighbor of the detached segment is extended to cover the detached region.

The DETACH clause can detach only a segment that is attached to a table or index with multiple named segments. It is not possible to detach an unattached segment or the segment of a single-segment table or index. (Therefore, a segment cannot be detached from a Red Brick Decision Server for Workgroups table or index.)

You cannot detach a segment from a referenced table if the segment is named in the range specification of a STAR index. Detaching such a segment would render the range specification invalid; therefore, an error message is displayed, identifying the STAR index(es) in question.

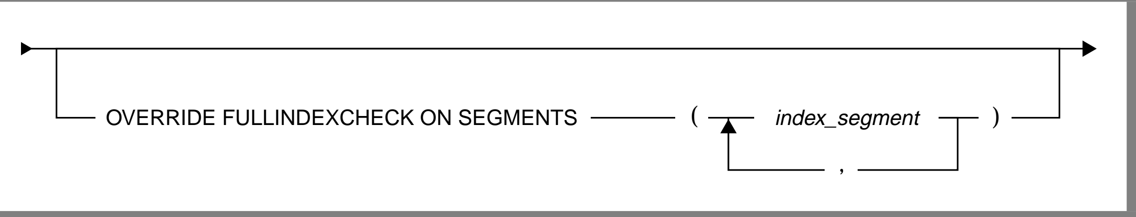
The DETACH clause is not available for use on the backup segment.

override_ fullindexcheck _specification

Detaches (or clears) a data segment from the specified table without performing a potentially time-consuming full index referential integrity check. This check can only be done when the data segment and one or more STAR index or primary key index segments are segmented *identically* on a specific column. This option saves time when you are detaching a data segment from a very large fact table (more than a billion rows).

This option must be used with great caution and with a clear understanding of the risks involved. Do not use this option unless the DETACH or CLEAR operation takes an inordinate amount of time, and you are certain that the data and index segments in question correspond exactly.

The following syntax diagram shows how to construct an *override_fullindexcheck_specification*:



index_segment Specifies a segment of a primary key index or a STAR index that is segmented identically to the data segment. After validating that the data and index segments correspond, the DETACH or CLEAR operation clears the data segment specified. Indexes that do not have segments referenced in the *index_segment* list still receive a full index scan.

If the validation process fails, an error message is issued, warning the user that the operation is invalid. The user can then specify the correct index segment name and issue the command again.

Examples

Suppose the ranges for a table are as shown in Figure 8-2:

Figure 8-2
Table Segment Ranges

If *seg3* is detached, the new ranges are as shown in Figure 8-3:

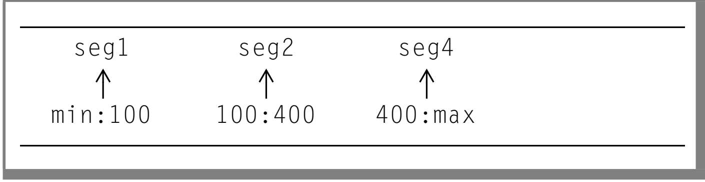


Figure 8-3
Segment Ranges with seg3 Detached

If the detached segment was the lowest segment in the range, the next lowest segment is extended to cover the range. Using the original ranges, if *seg1* is detached, *seg2* is extended, and the new ranges are as shown in Figure 8-4:

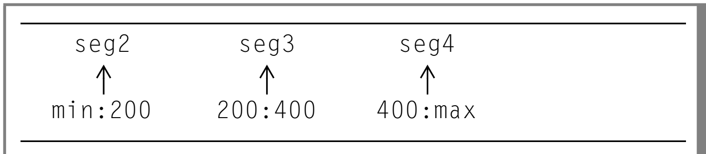


Figure 8-4
Segment Ranges with Lowest Segment Detached

In the following example, the OVERRIDE FULLINDEXCHECK option is used to save time when the *sales_data1* segment is detached from the Sales table:

```
alter segment sales_data1
    of table sales
    detach override fullindexcheck on segments
        (sales_star1, sales_pk1)
```

The full index scan is bypassed for the *sales_star1* and *sales_pk1* index segments because they are segmented identically to the *sales_data1* segment.

VERIFY	Examines the PSUs in the specified segment and determines whether the segment is intact or physically damaged. If it is unclear why a segment is damaged, use the VERIFY clause to determine the cause of damage. After repairing the damage, use VERIFY again to check that the PSU(s) in the segment are undamaged and to mark the segment intact. VERIFY marks a segment intact by updating the INTACT column of the RBW_SEGMENTS table. Note that the process of examining each PSU in a segment can be lengthy. The ONLINE clause performs the same verification tasks and returns the same information as VERIFY and in addition brings the segment online. ONLINE cannot be used on segments that are already online; use VERIFY to check online segments. The VERIFY option is available for use on the backup segment. For information about recovering a damaged segment, refer to the *Informix Red Brick Decision Server Administrator's Guide*.
FORCE INTACT	Marks an unavailable segment intact by updating the INTACT column of the RBW_SEGMENTS table without first examining the PSUs for possible physical damage. Both the FORCE INTACT clause and the VERIFY clause mark a segment intact; however, because FORCE INTACT does not examine the PSUs, FORCE INTACT takes significantly less time than VERIFY. If the PSUs might be damaged, use VERIFY to examine them. Use FORCE INTACT only if the segment is unavailable because of a minor error that had no effect on data integrity. The FORCE INTACT clause can be used on the backup segment, although it should be used with caution. Refer to the *Informix Red Brick SQL-BackTrack User's Guide* for more information.

SEGMENT BY segmenting_ column	Specifies a segmenting column for a table or index that resides in one segment and was not assigned a segmenting column when the table or index was created. For tables, the segmenting column must not allow NULL values. For indexes, the segmenting column must be the first column specified in the index key.

When the first column of a STAR index key is a multi-column foreign key, the foreign key constraint name must be used to identify the first component of the index key in the SEGMENT BY specification.

After the segmenting column is assigned to the table or index, the segment has an implicit range of (*min:max*) based on values or row IDs in the specified segmenting column. Additional segments can be attached to the table or index and their ranges merged with the implicit range.

This clause cannot be used to change the segmenting column of a table or index that has already been assigned a segmenting column, or to assign a segmenting column to a table that is segmented by hash values.

(In Red Brick Decision Server for Workgroups databases, a segmenting column cannot be specified. The SEGMENT BY clause is also not available for use on the backup segment.)

RANGE Specifies a new range of values or row IDs for a segmenting column:

- The range specified for a segment attached to a table or to a B-TREE or TARGET index is based on values in the segmenting column. In this case, the *rangeval* must be a literal value.
- The range specified for a segment of a STAR index is based on ROW IDs (segment name, row number) in the table referenced by the segmenting column of the index. The first row number is 0. Range specifications for STAR indexes must follow one of the formats described on page 8-27.

For more information about segment range values, refer to the CREATE TABLE and CREATE INDEX command descriptions.

Range modifications must neither produce any gaps and overlaps in the segmentation ranges for the table or index, nor require the movement of any existing row data or index data from one segment to another as a result of the change.

For example, if an existing segment has the range (1000:2000) and is altered to have the range (1000:2500), then the lower boundary of the segment above this segment is automatically adjusted to start at 2500. However, the ALTER SEGMENT command will fail if any existing data falls in the range 2000 to 2500.

This clause can be used only for a segment attached to a table or index with multiple named segments. This clause cannot be used to:

- Change the range of an unattached segment.
- Change the range of a segment that is the only segment of a table or index.
- Add a range to a table that is segmented with the hashing scheme.

(In Red Brick Decision Server for Workgroups databases, a segment range cannot be specified. Likewise, a segment range cannot be specified on the backup segment.)

OFFLINE Sets the specified segment to OFFLINE mode, which temporarily makes the segment unavailable for use in the database. When a segment is offline, an administrator can load the segment with new data, restore it in case of media failure or other data loss, or detach it to remove it from the table or index.

If a segment that contains either row data or index data related to a table is in OFFLINE mode, the table is partially available. Users can access the online segments of a partially available table. Query behavior of partially available tables is set in the OPTION PARTIAL_AVAILABILITY parameter of the rbw.config file or with the SET PARTIAL AVAILABILITY command. For information about the *rbw.config* file, refer to the *Informix Red Brick Decision Server Administrator's Guide*. For information about the SET PARTIAL AVAILABILITY command, refer to page 9-38.

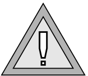
This clause can be used only for a segment attached to a table or index with multiple named segments. It is not possible to take offline:

- An unattached segment
- The only segment of a table or index
- The last online segment of a table or index

(In Red Brick Decision Server for Workgroups databases, a segment cannot be set to OFFLINE mode. Likewise, because it always remains online following its creation, the backup segment cannot be set to OFFLINE mode.)

OVERRIDE REFCHECK Overrides referential integrity checking when an ALTER SEGMENT statement is used to take offline (or clear) a segment that belongs to a table that is referenced by another table.

If both the segment and the referencing table contain data but you are sure that clearing the segment or taking it offline will not result in a violation of referential integrity, this override option allows you to force the operation to proceed.



Warning: *The OVERRIDE REFCHECK option should be used only with great caution and a clear understanding of its consequences. Clearing a segment in a referenced table will violate referential integrity if any values in the referencing table correspond to values in the cleared segment.*

If you do not use this option and both the segment and the referencing table contain data, a referential integrity violation might occur; therefore, the ALTER SEGMENT command will fail and an error message will be displayed. (If the segment or the referencing table is empty, however, no violation of referential integrity is possible and the command will succeed.)

Example

For example, if the Sales fact table contains sales figures for 1998 through 2000 and references a Period table whose data is stored in two segments, *period_seg1* and *period_seg2*, you will receive an error message if you try to take either of those segments offline. But if *period_seg2* contains data for years 1997 through 1999 only, it would be safe to override the referential integrity check and take the segment offline:

```
alter segment period_seg2 of table period
    offline override refcheck
```

ONLINE

Sets an offline segment to ONLINE mode, which makes the segment available for use with the database. As part of the process of setting a segment to ONLINE mode, the server attempts to verify that the segment is undamaged. If the server finds damage, the segment remains in OFFLINE mode and the server reports possible causes of the damage.

When all row data and index segments of a table are online, the table is fully available for use with the database.

Before setting an offline segment to ONLINE mode after an offline load has occurred, the segment must first be synchronized with the TMU SYNCH command. For information about the TMU SYNCH command, refer to the *Table Management Utility Reference Guide*.

(In Informix Red Brick Decision Server for Workgroups databases, a segment cannot be set to ONLINE mode. Similarly, because the backup segment is brought online as soon as it is created and always remains online, the ONLINE option is also unavailable for the backup segment.)

CLEAR
Removes all rows in a data segment and the index entries that reference the rows. Clearing a segment effectively performs a bulk delete on the segment. Bulk deletes can be used to undo a load into an offline segment, if necessary.

Deleting all the rows from the data segment is a relatively quick operation; however, deleting the corresponding index entries requires significantly more time, particularly when the number of affected entries is small compared to the total number of index entries. Therefore, use CLEAR to empty a segment with a large number of rows (and a corresponding large number of index entries). Write a DELETE statement with the appropriate constraints to empty a segment with a small number of rows.

This clause can be used to clear online and offline segments attached to a table that resides in multiple named segments. It cannot be used to clear an index segment, an unattached segment, or the only segment of a table.

(In Informix Red Brick Decision Server for Workgroups databases, a segment cannot be cleared. The CLEAR option is also unavailable for use on the backup segment.)

override_ fullindexcheck_ specification
Clears a data segment from the specified table without doing a full index referential integrity check. For details, refer to page 8-33.

OVERRIDE REFCHECK
Overrides referential integrity checking when an ALTER SEGMENT statement is used to clear a segment that belongs to a table that is referenced by another table. For details, refer to page 8-39.

RENAME new_seg_name
Changes the name of the specified segment. Both default and named segments can be renamed while detached from or attached to a table or index.

The RENAME option is available for use on the backup segment.

psu_sequence_id
Specifies the server-assigned sequence number for each PSU in a segment. For example, sequence ID 1 is assigned to the first PSU in a segment and sequence ID 2 is assigned to the next PSU. Sequence ID numbers are stored in the RBW_STORAGE table.

CHANGE MAXSIZE psu_sequence_id TO max_size
Changes the maximum size of the PSU. The maximum size of PSUs within both default and named segments can be changed. Although *max_size* must be specified in kilobytes, the actual size is rounded up to the next multiple of 8 kilobytes. The lowest valid MAXSIZE parameter is 16 kilobytes.

This clause cannot be used to increase the maximum size of a PSU if the PSU with the next PSU sequence ID in the segment has data stored in it. However, the maximum size of a PSU can be increased if the next PSU has been pre-allocated space with the INITSIZE parameter but has no data stored in it.

The CHANGE MAXSIZE option is available for use on the backup segment.

Example

As Figure 8-5 shows, the *psu2* PSU has been allocated data; therefore, it is not possible to increase the maximum size of *psu1*. It is possible to increase the maximum size of *psu2* because *psu3* has not been allocated data.

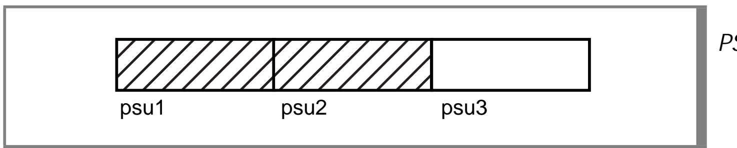


Figure 8-5
PSU Data Allocation

Example

The following ALTER SEGMENT statement increases the maximum size of *psu1* to 100 kilobytes:

```
alter segment
    segment2 of table sales
    change maxsize 1 to 100
```

CHANGE EXTENDSIZE psu_sequence_id TO increment — Changes the amount by which the PSU will expand each time it needs to expand. The increment value is specified in kilobytes and will be rounded up to the next multiple of 8.

The CHANGE EXTENDSIZE option is available for use on the backup segment.

CHANGE PATH psu_sequence_id TO 'new_filename' — Changes the location of the specified PSU.

Specifying a new location does not actually move the PSU; it only updates the LOCATION column in the RBW_STORAGE table. To physically move or copy a file, use the appropriate operating-system command.

The server verifies the following information:

- The file exists in the directory path; a warning is issued if it does not.
- File permissions allow access by the *redbrick* user; an error occurs if they do not.
- File size is a multiple of 8 kilobytes; an error occurs if it is not.
- The file is a regular file, not a directory, character device, block device, named pipe, hard link, symbolic link, or socket. An error occurs if the file named is any of these other items, except a symbolic link, which causes an informational message to be issued.

MIGRATE TO 'dir_path' — Copies an entire segment to the location specified by *dir_path*, which can be a full pathname or relative to the database directory.

You can use the MIGRATE TO clause to move an entire segment from one location to another—for example, from disk to optical storage, from optical storage to disk, or from one disk to another.

If the migration operation cannot complete because it runs out of space or encounters duplicate filenames, you can restart it after correcting the problem and it will continue copying the remaining PSUs.

If you encounter duplicate filenames because PSUs that formerly resided in different directories are now being moved into a single directory, follow this procedure to solve the problem:

1. Use an operating system command to rename the file containing the PSU with a unique name.
2. Use ALTER SEGMENT…CHANGE PATH to change the PSU name in the database system tables.
3. Use ALTER SEGMENT…MIGRATE TO to complete the migration operation.

The MIGRATE TO option is available for use on the backup segment.

DROPPING, KEEPING — Specifies whether the original PSUs are kept or dropped after the copy is completed. The default is DROPPING.

COMMENT — Assigns a descriptive comment string to the segment, which is stored in the RBW_SEGMENTS system table. A comment can contain up to 256 bytes. The server makes no use of the comment text.

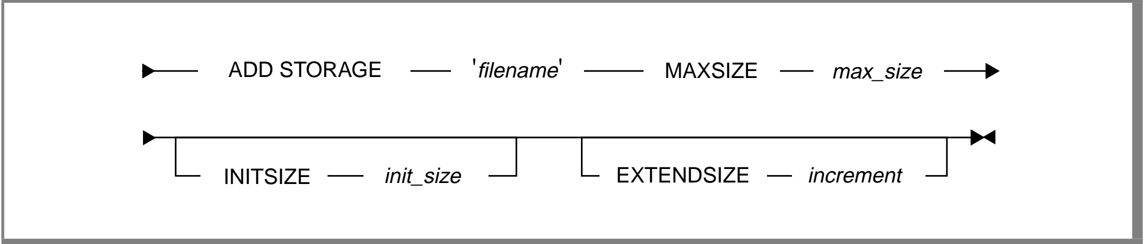
Specifying NULL replaces the comment string with NULL.

This clause is available for use on the backup segment.

add_storage_ specification — Adds a new physical storage unit (PSU) to a segment. A segment can contain multiple PSUs each with a different size.

The ADD STORAGE option is available for use on the backup segment.

The following syntax diagram shows how to construct an *add_storage_specification*. To see how the storage specification relates to the ALTER SEGMENT statement, refer to page 8-30.



ADD STORAGE filename — Adds a new PSU to the specified segment. The new PSU is added to the end of the existing PSUs in the segment.

The pathname of the new PSU can be a pathname relative to the database directory or it can be an absolute pathname. All specified directories must exist.

MAXSIZE max_size — Specifies the maximum number of kilobytes of data that will be loaded into the PSU before the next PSU in the sequence is used. Although the MAXSIZE parameter must be specified in kilobytes, the actual size is rounded up to the next multiple of 8 kilobytes. The lowest valid MAXSIZE parameter is 16 kilobytes.

The MAXSIZE parameter is stored in the MAXSIZE column of the RBW_STORAGE system table. This system table also contains the number of kilobytes that have been used (USED column) in a PSU.

INITSIZE init_size

Specifies the amount of initial space pre-allocated for the PSU. The value is specified in kilobytes and is rounded up to the next multiple of 8 kilobytes. The value must be less than or equal to the maximum size specified with the MAXSIZE parameter.

The INITSIZE parameter is stored in the INITSIZE column of the RBW_STORAGE system table.

The initial size of the first PSU in a segment is always at least 16 kilobytes. If an initial size between zero (0) kilobytes and less than 9 kilobytes is specified for the first PSU, the server returns an error. If an initial size between 9 kilobytes and less than 16 kilobytes is specified, the value is rounded up to 16 kilobytes. The initial size of subsequent PSUs can be from zero (0) to the maximum size.

The default is 16 kilobytes.

EXTENDSIZE increment

Specifies the amount the PSU expands beyond the initial size each time it becomes full and needs to expand. The increment value is specified in kilobytes and will be rounded up to the next multiple of 8 kilobytes. The default is 8 kilobytes.

Example

For the following sequence, assume the Budget table is divided into two segments and the Mktkey column is the segmenting column. Figure 8-6 shows the ranges of the segments:

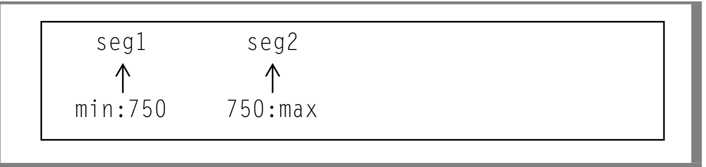


Figure 8-6
Budget Table Segment Ranges

The following ALTER SEGMENT statement attaches a third segment to the Budget table. The statement is legal only if no rows exist in the Budget table whose values for the Mktkey column fall between 1500 and the maximum possible value.

```
alter segment seg3
    attach to table budget
    range (1500:max)
```

The upper range of *seg2* is now 1500, and the new segment ranges of the Budget table are as shown in Figure 8-7.

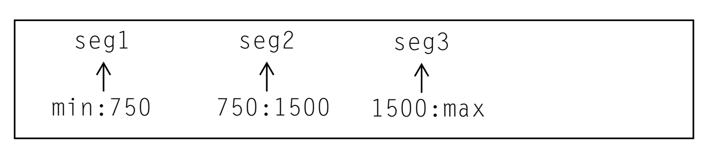


Figure 8-7
Segment Ranges after New Segment Is Attached

Next, *seg2* is set to OFFLINE mode, then detached from the Budget table. All of the rows that reside in *seg2* are now removed from the database.

```
alter segment seg2 of table budget
    offline
alter segment seg2 of table budget
    detach
```

The segment ranges for the Budget table are now as shown in Figure 8-8.

Figure 8-8
Segment Ranges after seg2 Is Set to OFFLINE

Example

For the following sequence, assume that the Sales table was created with no segmenting column; all data resides in the *seg_sales1* segment.

The following ALTER SEGMENT statement specifies the Perkey column as the segmenting column of the Sales table. The implicit range for the *seg_sales1* segment is (*min:max*).

```
alter segment seg_sales1 of table sales
    segment by perkey
```

Now that a segmenting column has been specified, a new segment can be attached to the Sales table based on values of the Perkey column.

```
alter segment seg_sales2
    attach to table sales
    range (500:max)
```

The segment ranges for the Sales table are as shown in Figure 8-9.

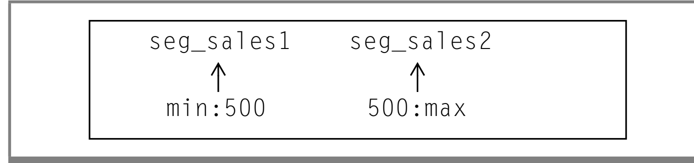


Figure 8-9
Segment Ranges for Sales Table

Example

The following ALTER SEGMENT statement sets a segment of a STAR index to OFFLINE mode:

```
alter segment default_segment_7 of index sales_star_idx
    offline
```

If a table or an index resides in a single segment, it cannot be taken offline.

ALTER SYNONYM

The ALTER SYNONYM command assigns a descriptive comment to a synonym or to one of its columns. Descriptive comments for synonyms are stored in the RBW_SYNONYMS and RBW_TABLES system tables. Descriptive comments for columns are stored in the RBW_COLUMNS system table.

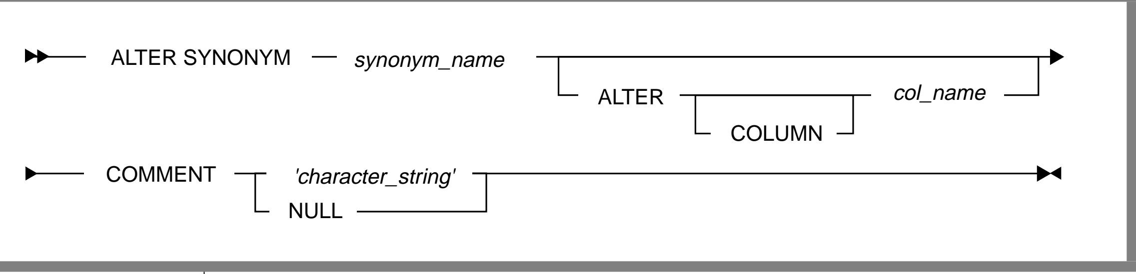
Authorization

To alter a synonym, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have ALTER_ANY authorization, either explicitly or through membership in a user-created role.
- Be a member of the RESOURCE system role and be the creator of the synonym.
- Be the creator of the synonym and have ALTER_OWN authorization, either explicitly or through membership in a user-created role.

Syntax

The following syntax diagram shows how to create an ALTER SYNONYM statement:



synonym_name	Specifies the name of the synonym to be altered.
ALTER COLUMN col_name	Specifies the name of a column to be altered in the specified synonym. The COLUMN keyword is optional.
COMMENT	Assigns a descriptive comment string to the synonym or to one of its columns. Comments for synonyms are stored in the RBW_SYNONYMS and RBW_TABLES system tables. Comments for columns are stored in the RBW_COLUMNS system table. A comment can contain up to 256 bytes. The server makes no use of the comment text. Specifying NULL replaces the comment string with the NULL indicator.

ALTER SYSTEM

The ALTER SYSTEM command allows database administrators and other users with the necessary authorization to control database activity and to perform various administrative actions.

Authorization

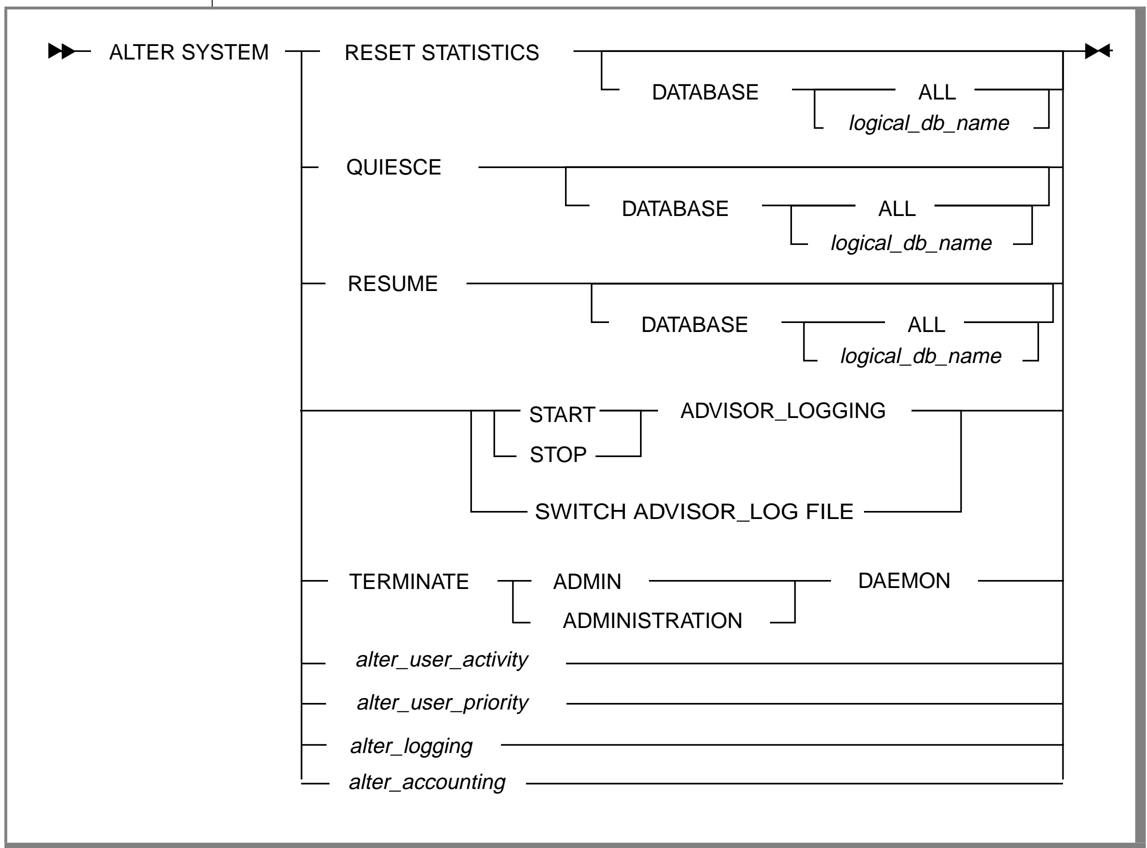
To issue ALTER SYSTEM commands against a database, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have ALTER_SYSTEM authorization, either explicitly or through membership in a user-created role.

A user who is a member of the DBA system role (or has the ALTER_SYSTEM authorization) for the administration database and is connected to that database can issue ALTER SYSTEM commands that affect all warehouse databases.

Syntax

The following syntax diagram shows how to construct an ALTER SYSTEM statement:



RESET STATISTICS	Resets all statistics in the dynamic statistic tables for the current database to zero. If the current database is the administration database, you must specify a DATABASE clause.
DATABASE	Specifies a single database or ALL databases. If the current database is the administration database, this clause is *required*. Otherwise, this clause does not apply.
ALL	Indicates that the ALTER SYSTEM command applies to all warehouse databases.
logical_db_name	Specifies the database name, which can be a logical database name as listed in the *rbw.config* file.
QUIESCE	Changes the state of the current or specified database to quiescent. No new commands are accepted by a quiescent database and no new connections can be made to it. Currently executing commands are allowed to complete. If the current database is the administration database, you must include the DATABASE clause and specify the name of the target database. This command does not prevent members of the DBA system role or users with the IGNORE_QUIESCE task authorization from connecting to the database or from executing new commands.
RESUME	Changes the state of one or more quiescent databases to active. This command must be issued by an existing session (since you cannot start a new session on a quiescent database) or by a user who is connected to the administration database and has ALTER_SYSTEM authorization for the administration database. If the current database is the administration database, you must include the DATABASE clause and specify the name of the target database.
TERMINATE ADMIN DAEMON	Terminates the administration daemon (*rbwadmd*). All information held in the dynamic statistic tables (DSTs) is lost when *rbwadmd* terminates. An administrator can restart the administration daemon with the *rbw.start* script.

START, STOP ADVISOR_ LOGGING	Offers the option to start or stop logging information into the log file. There is no default setting for this command. This command overrides the value set with the ADMIN ADVISOR_LOGGING parameter in the rbw.config file.
SWITCH ADVISOR_LOG FILE	Creates a new active log file with a default name and logs the following information: ■ Timestamp: indicates the date and time the message was logged. ■ Database name: specifies the name of the database being used. ■ Base table identification: integer that identifies the base table that was used to create the precomputed view. ■ View identification used to answer a query: integer that identifies a precomputed view that was used to answer a query. ■ Rollup information: integer that indicates the number of times a view was referenced to answer queries asking for either a subset of the view's grouping columns or an attribute of a dimension with less granularity. ■ Elapsed time for the query and each aggregate block within the query: integer that indicates the total amount of time spent executing the aggregate parts of a query. ■ SQL text for the aggregate block: represents the candidate view's definition.
alter_user_ activity	The *alter_user_activity* clause includes two ALTER SYSTEM options: CLOSE USER SESSION and CANCEL USER COMMAND. Both of these options cancel currently running user commands. The difference is that the CLOSE USER SESSION option also terminates the session or sessions that are running the commands. The *alter_user_activity* clause is further defined on page 8-55.

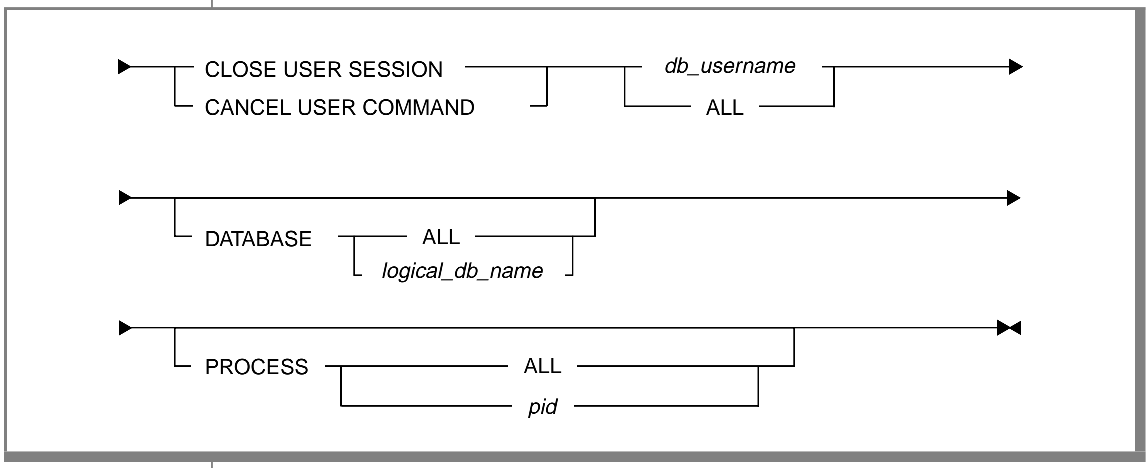
alter_user_priority — The *alter_user_priority* clause changes the priorities of current user sessions. Changes to user priority take place immediately for the current sessions. These changes are not permanent however. In other words, any new sessions started for the user have the original priority. To make a permanent change to a user priority, use the ALTER USER command. Your platform must have the UNIX *renice* command in order to support user priorities. You must specify the full pathname of the *renice* script with the ADMIN RENICE_COMMAND configuration parameter. The *alter_user_priority* clause is further defined on page 8-58.

alter_logging — The *alter_logging* clause contains options for controlling logging operations. The *alter_logging* clause is further defined on page 8-59.

alter_accounting — The *alter_accounting* clause contains options for controlling accounting operations. The *alter_accounting* clause is further defined on page 8-61.

Alter User Activity Specification

The following syntax diagram shows how to construct an *alter_user_activity* clause. To see how the *alter_user_activity* clause relates to the ALTER SYSTEM statement, refer to page 8-51.



CLOSE USER SESSION	Cancels currently executing commands for one or all user sessions on the current database and terminates those sessions. If the current database is the administration database, the DATABASE clause is required.
CANCEL USER COMMAND	Cancels currently executing commands for one or all user sessions on the current database. If the current database is the administration database, the DATABASE clause is required.
db_username	Specifies a valid database username.
ALL	Specifies that the command applies to all users of the specified database or databases.
DATABASE	Specifies a database or all databases. If the current database is the administration database, this clause is required. If the current database is not the administration database, this clause is not allowed.
ALL	Indicates that the command applies to all warehouse databases.
logical_db_name	Specifies a logical database name as listed in the *rbw.config* file.
PROCESS	Specifies a particular session by its process ID. If you are connected to the administration database, this clause must follow a DATABASE clause. For example, if you are connected to the administration database and you want to terminate all processes for a specific user on the database DB1, this clause must follow a DATABASE DB1 clause.
ALL	Specifies all sessions. This is the default if no PROCESS option is specified.
pid	Specifies the process ID of a particular user session.

Examples

The following statement cancels the currently executing command for a particular session run by user *diaz* on the current database:

```
alter system cancel user command diaz
    process 23581
```

The following statement cancels the commands for all sessions run by user *intern* on the database *marketing*, and terminates those sessions. To execute this command, the user must have the ALTER SYSTEM authorization for the administration database and must be connected to that database.

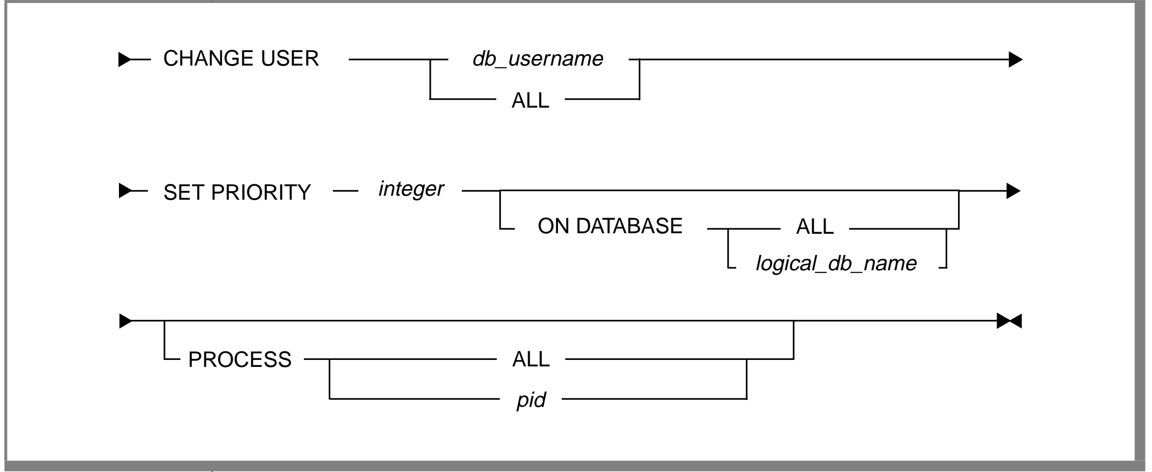
```
alter system close user session intern
    database marketing
    process all
```

The following example cancels the commands for all sessions on all warehouse databases that are running for user *jones*, and terminates all those sessions as well. To execute this command, the user must have the ALTER SYSTEM authorization for the administration database and must be connected to the administration database.

```
alter system close user session jones database all
```

Alter User Priority Specification

The following syntax diagram shows how to construct an *alter_user_priority* clause. To see how this clause relates to the ALTER SYSTEM statement, refer to page 8-51.



CHANGE USER	Changes the priority of one or more user sessions. (The ADMIN RENICE_COMMAND configuration parameter must be set in order to make use of this clause.)
db_username	Specifies a valid database username.
ALL	Specifies that the command applies to all users of the specified database or databases.
SET PRIORITY integer	Sets the priority of the session or sessions to the value specified by *integer*. This value can be between 0 and 100, inclusive. The highest priority has value 0.
ON DATABASE	Specifies a database or all databases. If the current database is the administration database, this clause is required. Otherwise, this clause does not apply.
ALL	Indicates that the command applies to all warehouse databases.

logical_db_name Specifies a logical database name as listed in the *rbw.config* file.

PROCESS Use this keyword to specify a particular session by its process ID.

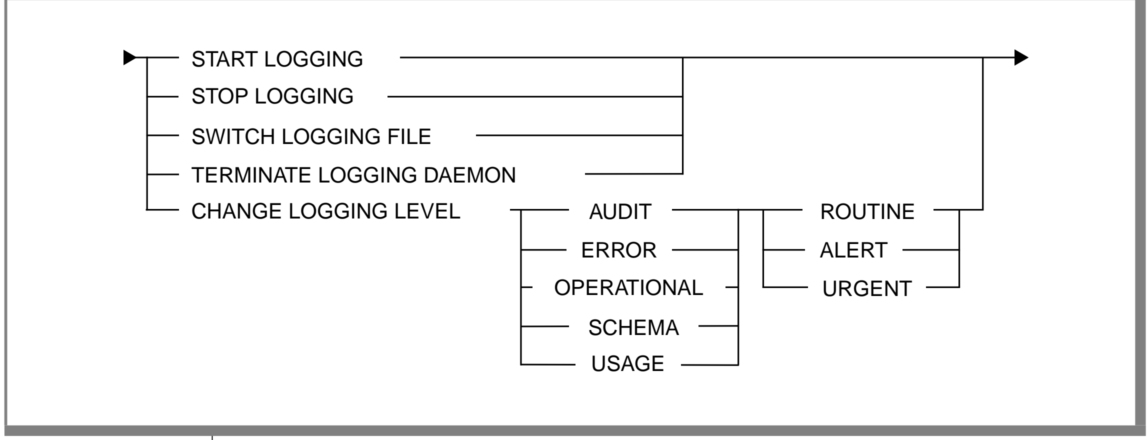
If you are connected to the administration database, this clause must follow a DATABASE clause. For example, if you are connected to the administration database and you want to change the priority for a specific user on the database DB1, this clause must follow a DATABASE DB1 clause.

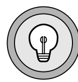
ALL Specifies all sessions. This is the default if no PROCESS option is specified.

pid Specifies the process ID of a particular user session.

Alter Logging Specification

The following syntax diagram shows how to construct an *alter_logging* clause. To see how this clause relates to the ALTER SYSTEM statement, refer to page 8-51.





***Tip:** The log daemon must be running to perform any of these operations.*

.

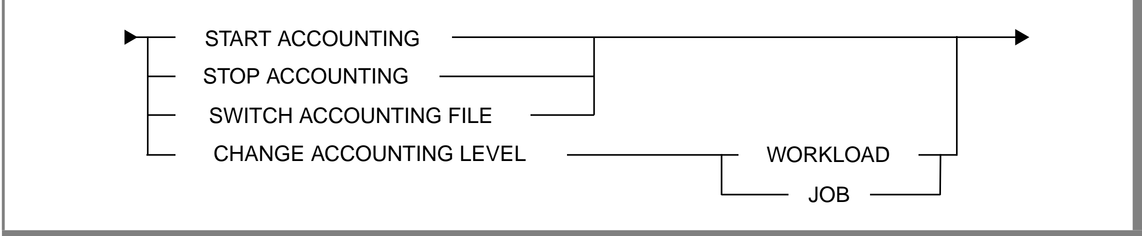
START LOGGING	Starts event logging. The log daemon begins accepting log request messages from warehouse processes and writes corresponding log records to a new log file.
STOP LOGGING	Stops event logging. The log daemon stops logging and closes the active log file. The log daemon continues to run, therefore logging can be restarted at any time.
SWITCH LOGGING FILE	Closes the active log file and creates a new active log file for subsequent log records. The closed file is renamed from *rbwlog.<daemon_name>.active* to *rbwlog.<daemon_name>.<datetime_stamp>*. If logging is stopped, this command has no effect.
TERMINATE LOGGING DAEMON	Terminates the log daemon process (*rbwlogd*) that performs both logging and accounting tasks.
CHANGE LOGGING LEVEL	Changes the log severity level for a selected log event category. The change takes effect immediately. The event categories are as follows:

- AUDIT (events relating to security and access control)
- ERROR (error events)
- OPERATIONAL (administrative actions)
- SCHEMA (changes to physical and logical database structures)
- USAGE (load, unload, and DML operations)

ROUTINE, ALERT, URGENT	Only log events having severity equal to or higher than the specified level will be logged for that event category. The lowest severity level is ROUTINE and the highest is URGENT. The ALERT severity level is higher than ROUTINE but lower than URGENT.

Alter Accounting Specification

The following syntax diagram shows how to construct an `alter_accounting` specification. To see how this clause relates to the ALTER SYSTEM statement, refer to page 8-51.



START ACCOUNTING	Starts accounting operations. The log daemon begins accepting accounting request messages and writes corresponding account records to a new account file. If accounting is already running, this option has no effect.
STOP ACCOUNTING	Stops accounting operations. The log daemon closes the active account file. The log daemon continues to run; therefore, accounting can be restarted at any time.
SWITCH ACCOUNTING FILE	Closes the active account file and creates a new active file for subsequent account records. The closed file is renamed from *rbwacct.<daemon_name>.active* to *rbwacct.<daemon_name>.<datetime_stamp>*. If accounting is not running, this command has no effect.
CHANGE ACCOUNTING LEVEL	Sets the level of detail of the captured account records to job accounting or workload accounting. This change takes effect immediately.
WORKLOAD, JOB	Specifies job accounting or workload accounting. Job accounting is limited to basic resource utilization information. Workload accounting includes additional detail, intended primarily for the use of Informix support personnel.

ALTER TABLE

An ALTER TABLE statement can be used to:

- Add, modify, or drop table columns.
- Change the maximum number of segments and rows per segment allowed for a table.
- Assign descriptive comments to a table or one of its columns. Descriptive comments for tables and columns are stored in the RBW_TABLES and RBW_COLUMNS tables, respectively.
- Add, drop, or alter foreign key constraints.

The table is locked automatically while the ALTER TABLE command is executed.

The ALTER TABLE statement is not supported for temporary tables.

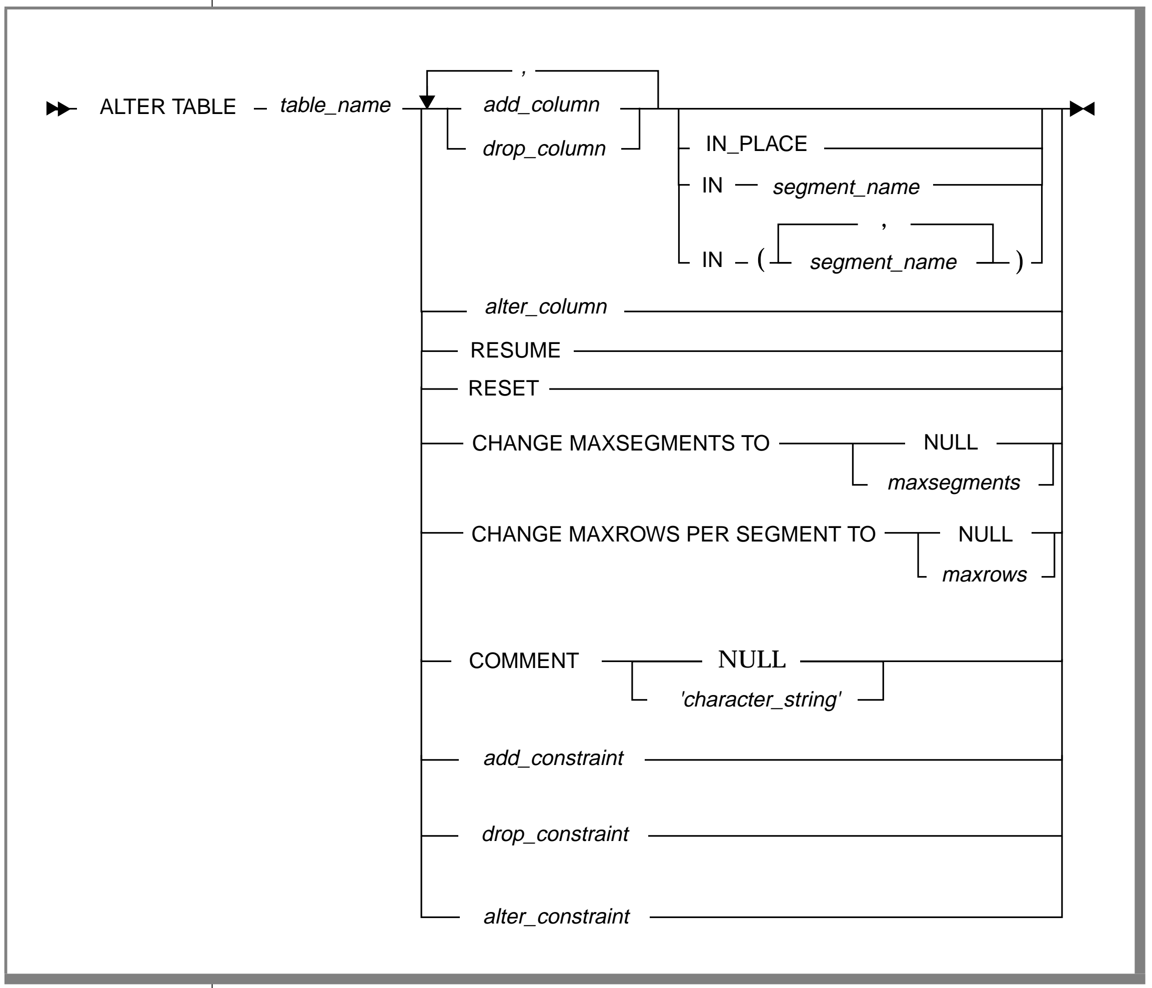
Authorization

To alter a table, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have ALTER_ANY authorization, either explicitly or through membership in a user-created role.
- Be the creator of the table and be a member of the RESOURCE system role.
- Be the creator of the table and have ALTER_OWN and ALTER_TABLE_INTO_ANY authorization. However, to use the IN_PLACE option, the user need not have ALTER_TABLE_INTO_ANY authorization.

Syntax

The following syntax diagram shows how to create an ALTER TABLE statement:



table_name Specifies the name of the table to alter. The table must be a user-defined table.

Synonyms, views, temporary tables, and system tables cannot be altered. Any synonyms defined on the table to be modified are modified simultaneously; however, views are not affected. To reflect the changes of the base table(s) in the view, the view must be dropped and re-created.

add_column, drop_column Describes the column to be added or dropped from the specified table.

A single ALTER TABLE statement can contain multiple specifications to allow for the simultaneous addition and removal of columns. There is no restriction on the number or combination of ADD or DROP specifications.

The *add_column* and *drop_column* specifications are further defined on page 8-67 and page 8-70 respectively.

alter_column Describes changes to be made to existing columns. Any existing column can be renamed or assigned a new default value. If the specified column is a foreign key column, the alter_column specification can be used to change the behavior when rows are deleted from the referenced table.

The alter_column specification can also be used to change the fill factor of a column or to assign a comment to it. Comments are stored in the RBW_COLUMNS table.

The *alter_column* specification is further defined on page 8-72.

IN_PLACE Adds or drops columns within existing segments.

If the new row is larger than the original row (that is, the total width of the added columns is larger than the total width of the dropped columns), then the segments must be large enough to hold all the rows in the table.

Note that if this option is used and the operation fails, the table cannot be reset.

IN segment_name Rebuilds the table in the named segment or segments. All rows are copied from the old segments into the new segments, applying the ADD or DROP specifications as the rows are copied.

If the table resides in a single segment, it must be copied to only one segment. If the table resides in multiple segments, it must be copied to an equal number of segments.

The original segment is either detached from the table or dropped after the modifications are complete. Named (user-created) segments are detached. Default segments are dropped.

The new segment or segments must be large enough to hold the entire table.

If neither IN_PLACE nor IN *segment_name* is specified, the modified table is built in a default segment. The original segment is either detached or dropped after the modifications are complete. Named (user-created) segments are detached. Default segments are dropped.

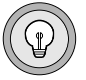
RESUME Completes an interrupted ALTER TABLE ADD COLUMN or DROP COLUMN operation.

A failed ALTER TABLE ADD CONSTRAINT or DROP CONSTRAINT operation cannot be resumed.

RESET Restores a table to its original state. RESET is valid only if the following conditions are met:

- The ALTER TABLE statement did not run to completion because it was interrupted by the user or failed during execution. (This does not include privilege violations such as lack of DBA authority.)
- The ALTER TABLE statement did not include the IN_PLACE option.

CHANGE MAXSEGMENTS TO...CHANGE MAXROWS PER SEGMENT TO	Changes *maxsegments* value or *maxrows per segment* value of the table. These values are used to calculate the size of the index key when a table is referenced by the foreign keys in a STAR index key, and to ensure that the space allocated for the index allows for anticipated growth. You can use an ALTER TABLE statement to change either value; you cannot use a single statement to change both values. Specifying NULL for either value is equivalent to omitting the specification in the CREATE TABLE statement, and is not recommended. For information about sizing STAR indexes, refer to the *Administrator's Guide*.



***Tip:** You cannot change the MAXROWS PER SEGMENT value of a referenced table if the change would render the range specification of a STAR index invalid; if you attempt to make such a change, an error message is displayed, identifying the STAR index (or indexes) in question.*

COMMENT	Assigns a descriptive comment string to the table, which is stored in the RBW_TABLES system table. A comment can contain up to 256 bytes. The server makes no use of the comment text. Specifying NULL replaces the comment string with NULL.
add_constraint	Specifies a foreign key constraint to be added to the table. For details, see "Add Constraint Specification" on page 8-75.
drop_constraint	Specifies a foreign key constraint to be dropped from the table. For details, see "Drop Constraint Specification" on page 8-79.
alter_constraint	Specifies a foreign key constraint to be altered so that it references a synonym or the base table for a synonym. For details, see "Alter Constraint Specification" on page 8-80.

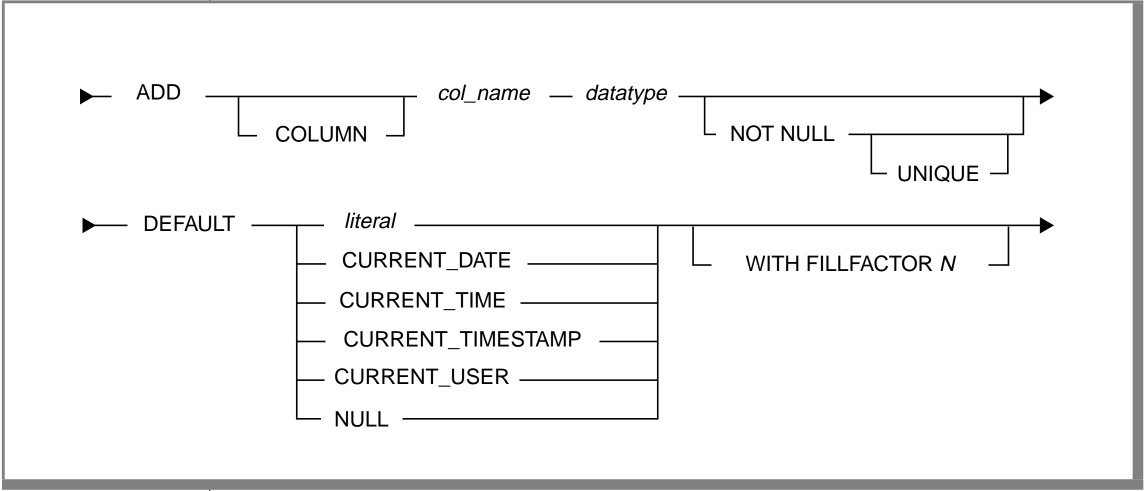
Add Column Specification

The *add_column* specification describes the column to be added to the specified table. To see how *add_column* fits within the ALTER TABLE statement, refer to “ALTER TABLE” on page -62.

In Red Brick Decision Server for Workgroups databases, if adding a new column would cause a table to exceed 5 gigabytes, the column will not be added.

Syntax

The following syntax diagram shows how to construct an *add_column* specification:



ADD	Creates a new column and adds it to the specified table as the last column in the table (as if the column had been created as the last column in a CREATE TABLE statement). The keyword COLUMN is optional.
col_name	Specifies the column to be added to the specified table. All column names specified in a single ALTER TABLE statement must be unique; they must also differ from the names of columns already defined in the table.
datatype	Specifies a datatype for the column. Any datatype used in a CREATE TABLE statement can be used in ALTER TABLE. The datatype and the value specified in the DEFAULT clause must be compatible. For information about datatypes, see "Datatypes" on page 2-18 and "Column Definitions" on page 8-135.
NOT NULL	Declares a column to be NOT NULL. If a column is declared NOT NULL, each row of the table must contain a value in that column; no missing or unknown values are allowed.

UNIQUE	Declares a column to be UNIQUE. If a column is declared unique, duplicate values are not allowed in it. Uniqueness is enforced only on B-TREE-indexed columns with single-column keys.
DEFAULT	Defines the DEFAULT value for the column. This value will be used for every existing row. The datatype of the new column must be type-compatible with the default value. For example, assigning 1 to a REAL column is legal, but assigning the following string to an INTEGER column is not legal. `'larry, moe, and curly'` Refer to page 8-138 for information about the legal keywords and values that can follow the DEFAULT keyword: ***literal,*** CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, CURRENT_USER, and NULL.
WITH FILLFACTOR N	This clause applies only to columns of the VARCHAR datatype. Specifies the percentage of the column length that a typical VARCHAR value is expected to take. This value is used to calculate the size of a row and the number of rows to allocate to a block. If WITH FILLFACTOR is not specified, the default column fill factor in the rbw.config file is used; if no column fill factor is defined there, the system default of 10 percent is used. For more information how the VARCHAR FILLFACTOR is used to calculate the number of rows per block, refer to the ***Administrator's Guide***.

Usage Notes

If there is more than one ADD specification, the columns are added at the end of the table in the order in which the ADD specifications appear in the ALTER TABLE statement.

Adding a column to a table does not affect any view because all column references in a view are resolved when the view is created. Thus, if a column is added to a view's base table(s), a view that contains SELECT * FROM as its query expression does not change to reflect columns added after the view was defined.

Examples

The following statement adds Sales_93, an integer column, to a table named Sales:

```
alter table sales
    add column sales_93 int default 0
in_place
```

The following statement adds two columns—Origin and Ranking—to the Product table, specifying a new segment, seg30, to hold the new larger table:

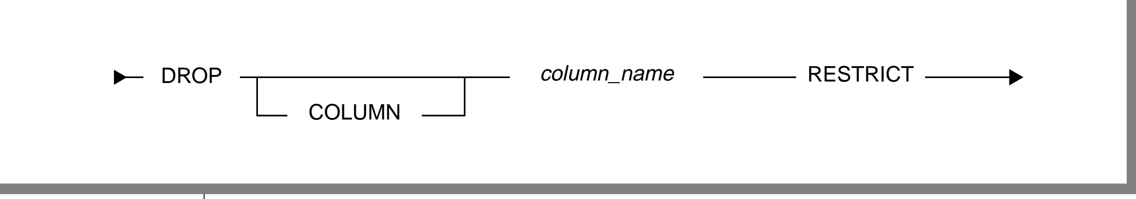
```
alter table product
    add column origin char(8) default 'ABC',
    add column ranking int default 1
in seg30
```

Drop Column Specification

The *drop_column* specification describes the column to be dropped from the specified table. To see how *drop_column* fits within the ALTER TABLE statement, refer to "ALTER TABLE" on page -62.

Syntax

The following syntax diagram shows how to construct a *drop_column* specification:



DROP Removes the column from the specified table. The keyword COLUMN is optional. After a column is dropped, the data is no longer available; the operation is not reversible.

column_name Specifies the column within the specified table that is to be dropped.

RESTRICT Causes the DROP COLUMN operation to fail if one or more of the following conditions is true:

- The dropped column is referenced either directly or indirectly (as in SELECT *) in any view. The view must be dropped first.
- The column is the primary key of the table.
- The column participates in any index. The index must be dropped first.
- The column participates in any foreign key.

Examples

The following statement performs a restricted delete in place on the Body column in the Product table:

```
alter table product drop body restrict in_place
```

The following statement performs multiple add and drop operations on the Product table:

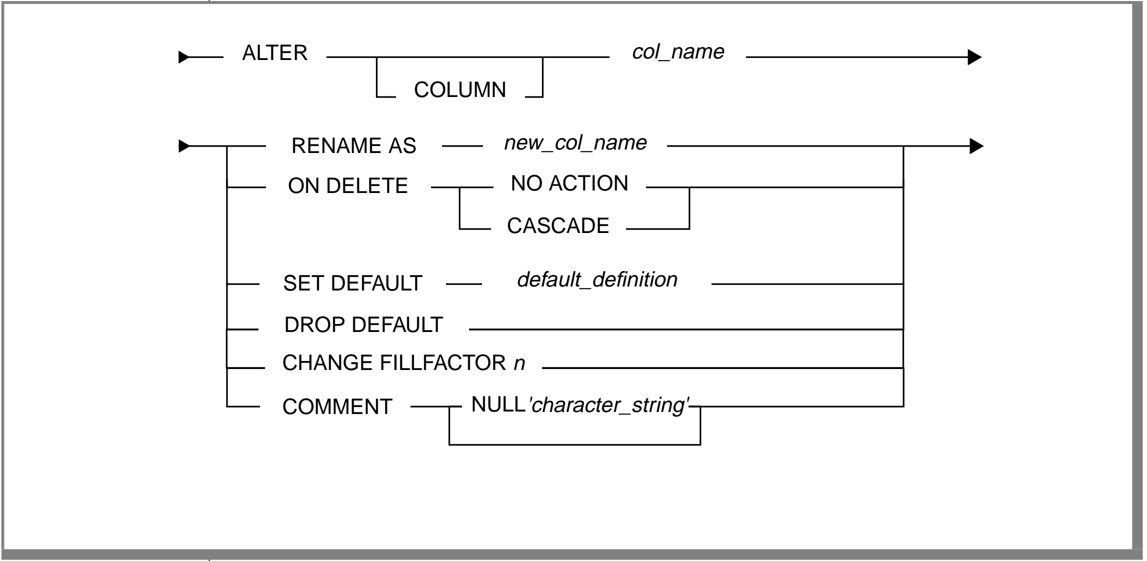
```
alter table product
    drop body restrict,
    add column origin char(8) default 'ABC',
    add column rank int default 2
in seg30
```

Alter Column Specification

The *alter_column* specification describes the column to be modified in the specified table. To see how *alter_column* fits within the ALTER TABLE statement refer to “ALTER TABLE” on page -62.

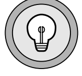
Syntax

The following syntax diagram shows how to construct an *alter_column* specification:



ALTER Specifies changes to an existing column.

col_name	Specifies the name of an existing column within the specified table.
RENAME AS new_col_ name	Changes the name of an existing column. The new name must be unique in the table. If a column is named in a view, the column cannot be renamed until the view is dropped.
ON DELETE	Changes the referential integrity check mode of a column. This is the behavior that occurs when a row is deleted from the table referenced by the foreign key. The ON DELETE clause can be applied only to a column declared as a foreign key. The NO ACTION keywords specify that if a row is to be deleted from the referenced table and deleting that row would violate the referential integrity of the referencing table, the row is not deleted. The CASCADE keyword specifies that if a row is to be deleted from the referenced table and deleting that row would violate the referential integrity of the referencing table, those rows in the referencing table that reference the row to be deleted are also deleted. This behavior also applies to outboard tables and their referencing tables.



Tip: *To change the delete action of a multi-column foreign key, you must specify a foreign-key constraint name as the col_name. (For single-column foreign keys, col_name can be a simple column name.)*

For examples illustrating the delete operation modes, refer to "DELETE" on page 8-166 and the *Informix Red Brick Decision Server Administrator's Guide*.

SET DEFAULT ***default_definition***	Sets a new default value for the specified column. The default value for a column is used for new rows that do not contain a value for the column. If a default is not specified for a column, the default is NULL. When no default value is specified for a NOT NULL COLUMN, a value *must* be included for the column during an insertion operation; otherwise, the server or TMU returns an error message. The *default_definition* can be a literal, a default function, or NULL, as described for the *column_definitions* clause of the CREATE TABLE statement. For a description of legal default values, refer to page 8-138.
DROP DEFAULT	Removes a default setting that was specified during table creation or in a previous ALTER TABLE statement. The default value for the column returns to NULL, which is the implicit default setting.
CHANGE FILLFACTOR n	Changes the fill factor for a VARCHAR column that is used to decide how many rows fit in a block. This statement does not change any existing data or segments that are attached to the table. The new fill factor is used in ALTER TABLE ADD COLUMN or DROP COLUMN statements that rewrite the table. For more information about fill factors, refer to the *Administrator's Guide*.
COMMENT	Assigns a descriptive comment string to the column, which is stored in the RBW_COLUMNS system table. A comment can contain up to 256 bytes. The server makes no use of the comment text. Specifying NULL replaces the comment string with NULL.

Examples

The following statement renames a column Distributor to Distrib:

```
alter table product
    alter column distributor rename as distrib
```

The following statement specifies the default value CA for the State column of the Market table:

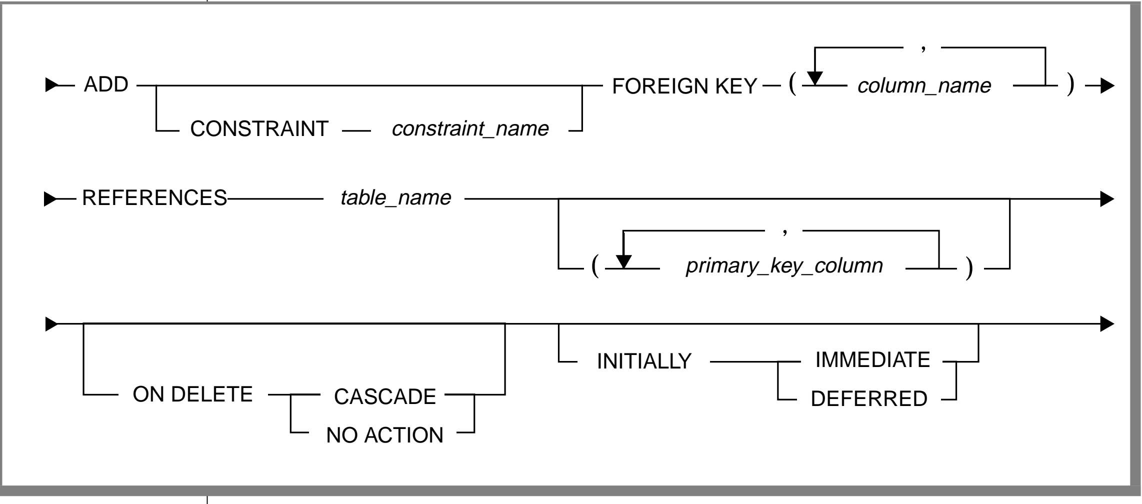
```
alter table market
    alter column state set default 'CA'
```

Add Constraint Specification

The *add_constraint* specification is used to add a foreign key constraint to a referencing table.

Syntax

The following syntax diagram shows how to construct an *add_constraint* specification:



constraint_name Specifies the name of the foreign key constraint to be added. The use of constraint names is optional. If a constraint name is specified, it must be preceded by the CONSTRAINT keyword and it must not be an existing constraint name. If a constraint name is not specified, a default constraint name is assigned. For detailed information about constraint names, see "Primary-Key and Foreign-Key Constraint Names" on page 8-144.

column_name Specifies a column name in the referencing table (that is, in the table being altered). One or more columns from that table must be specified. Each column must be declared NOT NULL. The list must be enclosed in parentheses and be preceded by the FOREIGN KEY keywords. The datatypes of the listed columns must exactly match the datatypes of the primary key columns in the referenced table.

table_name Specifies the referenced table (that is, the table that contains a primary key that will become a foreign key reference in the table being altered).

primary_key_column Specifies a column name in the referenced table. One or more columns can be specified, but the list must match the primary key columns in the referenced table. The list must be enclosed in parentheses.

ON DELETE

Defines the referential-integrity check mode of a constraint. This is the behavior that occurs when a row is deleted from the table referenced by the foreign key.

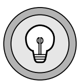
The NO ACTION keywords specify that if a row is to be deleted from the referenced table and deleting that row would violate the referential integrity of the referencing table, then the row is not deleted.

The CASCADE keyword specifies that if a row is to be deleted from the referenced table and deleting that row would violate the referential integrity of the referencing table, then those rows in the referencing table that reference the row to be deleted are also deleted. This behavior also applies to outboard tables and their referencing tables.

For examples illustrating the delete operation modes, refer to "DELETE" on page 8-166 of this document and the *Administrator's Guide*.

INITIALLY IMMEDIATE, DEFERRED

Specify whether referential integrity checking is done on the column values for the new constraint (foreign key reference). IMMEDIATE denotes that referential integrity will be immediately checked, and DEFERRED that it will not be checked; the default is IMMEDIATE.



Tip: *The INITIALLY DEFERRED option does not check for referential integrity violations and should only be used when you are certain that referential integrity is enforced. For example, say you create a synonym for a table that has a foreign key reference to it and you add a new foreign key constraint from the referencing table to the new synonym. In this case, you know that referential integrity is enforced in the synonym because it is enforced in the underlying base table.*

Usage Notes

A table can have a maximum of 256 foreign keys.

It is recommended that user-defined constraint names be given to all foreign key references. User-defined constraint names must be used to refer to multi-column foreign keys; otherwise, STAR indexes cannot be built on those keys.

A failed ALTER TABLE ADD CONSTRAINT statement can be reset with the ALTER TABLE RESET command, but it cannot be resumed with ALTER TABLE RESUME.

An ALTER TABLE ADD CONSTRAINT statement will fail if the referenced table does not have a system-generated or user-defined B-TREE index built on its primary key columns.

An ALTER TABLE ADD CONSTRAINT statement will fail if the new constraint would violate the referential integrity of the rows in the altered table or introduce a referential integrity cycle. All existing row values in the columns associated with the new constraint must have corresponding primary key values in the referenced table.

You cannot add a constraint from a table to itself. For example, you cannot add a constraint from *table1* to *table2* if there is already a constraint from *table2* to *table1*. This kind of table definition would produce a referential integrity cycle.

Example

To add a foreign key reference in the Orders table to the Promokey column of the Promotion table, issue an ALTER TABLE statement like this:

```
alter table orders
    add foreign key(promokey) references promotion(promokey)
```

In order for this statement to work, the Promokey column must already exist in the Orders table and it must be the primary key of the Promotion table.

Assume that the Sales table is created with primary key columns Perkey, Promokey, and Custkey, which reference the Period, Promotion, and Customer tables, respectively. To add a foreign-key reference to the Product table, which has a two-column primary key, issue an ALTER TABLE statement like this:

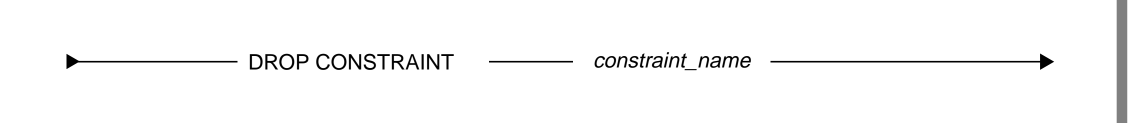
```
alter table sales
    add constraint product_fkc
    foreign key(classkey, prodkey) references
product(classkey,
        prodkey)
```

Drop Constraint Specification

The *drop_constraint* specification is used to drop a foreign key constraint.

Syntax

The following syntax diagram shows how to construct a *drop_constraint* specification:



constraint_name Specifies the name of the foreign key constraint to be dropped. For detailed information about constraint names, see "Primary-Key and Foreign-Key Constraint Names" on page 8-144.

Usage Notes

An ALTER TABLE DROP CONSTRAINT statement will fail if the specified foreign key is part of a STAR index key. The index must be dropped first. (The statement will succeed if a STAR index is defined on the altered table but its key consists of foreign keys other than the one being dropped.)

Primary key constraints cannot be dropped.

Example

The following example drops the product_fkc constraint from the Sales table:

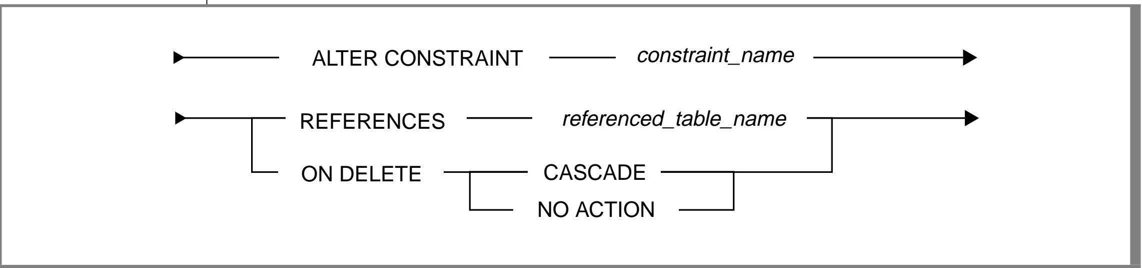
```
alter table sales drop constraint product_fkc
```

Alter Constraint Specification

The *alter_constraint* specification allows a foreign key reference to a base table to be changed to reference a synonym. A foreign key reference to a synonym can likewise be changed to reference the base table or another synonym created for that table.

Syntax

The following syntax diagram shows how to construct an *alter_constraint* specification:



constraint_name Specifies the name of the foreign key constraint to be altered, as defined in the CREATE TABLE statement for the referencing table. For single-column foreign key references, the column name can also be used to specify the constraint.

For detailed information about constraint names for foreign key references, see "Foreign-Key Reference" on page 8-141.

referenced_table_name Specifies any synonym created for the base table originally referenced in the CREATE TABLE statement or the name of the base table itself. If the original reference was to a synonym, the *referenced_table_name* can be the base table or another synonym created for that table.

ON DELETE Defines a constraint's referential-integrity check mode, the behavior that occurs when a row is deleted from the table referenced by the foreign key.

The NO ACTION keywords specify that if a row is to be deleted from the referenced table and deleting that row would violate the referential integrity of the referencing table, then the row is not deleted.

The CASCADE keyword specifies that if a row is to be deleted from the referenced table and deleting that row would violate the referential integrity of the referencing table, then those rows in the referencing table that reference the row to be deleted are also deleted. This behavior also applies to outboard tables and their referencing tables.

Usage Notes

The ALTER TABLE ALTER CONSTRAINT command can be used to tune the performance of some queries that join multiple referencing tables that have shared referenced tables. If the referencing tables reference synonyms instead of base tables, hash join or B-TREE 1-1 match algorithms can be used (rather than the standard STARjoin algorithm).

Example

The following ALTER TABLE command alters the Sales table in the Aroma database by changing one of its foreign key references.

```
alter table sales
    alter constraint sales_date_fkc references period_syn1
```

Instead of referencing the Period table, as defined in the CREATE TABLE statement, the foreign key constraint *period_fkc* now references the Period_Syn1 synonym.

Because *sales_date_fkc* is a single-column foreign key reference, *perkey*, the column name, could also be used to specify the constraint:

```
alter table sales
    alter constraint perkey references period_syn1
```

The following statement changes the referenced table back to the original base table:

```
alter table sales
    alter constraint sales_date_fkc references period
```

The following statement changes the referenced table to another synonym of the Period table:

```
alter table sales
    alter constraint sales_date_fkc references period_syn2
```

ALTER USER

The ALTER USER command changes a user's priority or assigns a descriptive comment to a user.

Authorization

To issue an ALTER USER command, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have the USER_MANAGEMENT task authorization.

Syntax

The following syntax diagram shows how to construct an ALTER USER statement:

db_username Specifies a valid database username.

SET PRIORITY integer Specifies the priority value for all sessions started for the user. The priority is an integer between 0 and 100, inclusive. The highest priority is the value 0.

COMMENT Assigns a descriptive comment string to the username, which is stored in the RBW_USERAUTH system table. A comment can contain up to 256 bytes. The server makes no use of the comment text.

Specifying NULL replaces the comment string with NULL.

Usage Notes

User priority is specified as an integer between 0 and 100 where 0 represents the highest priority. Priority values determine the relative importance of sessions started by different users. A query running for a user with a high priority value will have greater access to the CPU than a query running for a user with a lower priority value. User priority does not affect any other aspect of system resource use besides CPU access, nor does it affect access to table locks.

The database administrator can set the user priority initially when adding the user to the database with the GRANT CONNECT command. If the database administrator has not specified a priority for the user, that user will have a priority of 50 by default.

UNIX Your platform must have the UNIX *renice* command in order to support user priorities. You must specify the full pathname of the *renice* script with the ADMIN RENICE_COMMAND configuration parameter.

Windows NT The integer specified for user priority is mapped to a corresponding priority level (1–36).

ALTER VIEW

An ALTER VIEW command assigns a descriptive comment to a view or to one of its columns. Descriptive comments for views are stored in the RBW_VIEWS and RBW_TABLES system tables. Descriptive comments for view columns are stored in the RBW_COLUMNS table.

Authorization

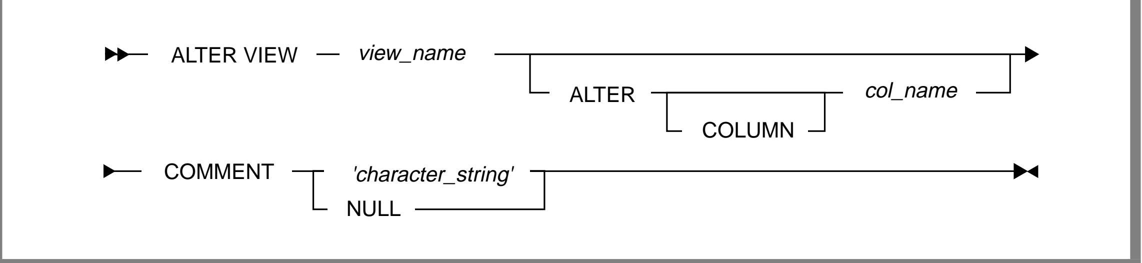
To alter a view, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have ALTER_ANY authorization, either explicitly or through membership in a user-created role.

- Be a member of the RESOURCE system role and be the creator of the view.
- Be the creator of the view and have ALTER_OWN authorization, either explicitly or through membership in a user-created role.

Syntax

The following syntax diagram shows how to create an ALTER VIEW statement:



view_name Specifies the name of the view to be altered.

ALTER col_name Specifies the name of a column to be altered in the specified view.

COMMENT Assigns a descriptive comment string to the view or to one of its columns. Comments for views are stored in the RBW_VIEWS and RBW_TABLES system tables. Comments for columns are stored in the RBW_COLUMNS system table. A comment can contain up to 256 bytes. The server makes no use of the comment text.

Specifying NULL replaces the comment string with NULL.

CHECK INDEX

The CHECK INDEX command checks for index corruption and provides configuration and size information about the index.

The CHECK INDEX command can be used on any type of index. In addition to checking for corruption, the command also reports configuration and size information for the index. Configuration information includes:

- The size and type of the index key
- The maximum number of entries per index block
- The number of segments in which the index is stored
- The MAXROWS PER SEGMENT value for each table referenced in the index (STAR indexes only)

Size information includes:

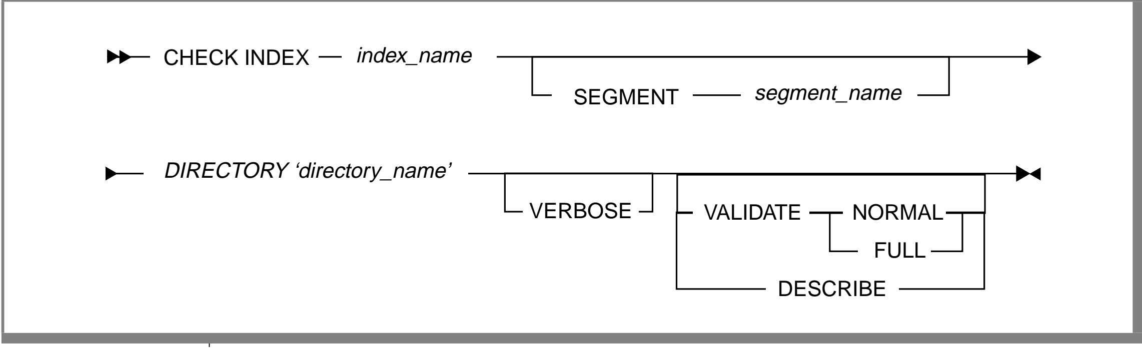
- The number of index entries
- The number of index blocks used as leaf nodes and inner nodes for each segment of the index.

Authorization

To issue the CHECK INDEX command, a user must be a member of the DBA system role.

Syntax

The following syntax diagram shows how to construct a CHECK INDEX statement:



index_name	Specifies the name of the index to be checked.
SEGMENT segment_name	Specifies a named segment on which CHECK INDEX is to be run. Only one segment may be specified. When *segment_name* is specified, CHECK INDEX checks only the specified index segment and displays results pertaining only to that segment.
DIRECTORY directory_name	Specifies the directory where the detailed output from the CHECK INDEX operation will be written. A separate file is written for each segment; the filename includes the table identifier, segment identifier, process ID, and a timestamp.

Only the *redbrick* user can read and write these files unless you set the CHECK_REPORT_FILE_PERMISSIONS option in *rbw.config* to allow different file permissions. The possible values are:

- SERVER_OWNER: Gives read and write permission to the *redbrick* user.

  This is the default option.
- SERVER_GROUP: Gives read-only permission to the group.

  This value is valid only on UNIX systems.
- ALL: Gives read-only permission to all users.

If the disk containing the directory is not writable or does not exist, an error is returned.

VERBOSE Specifies output to be provided in verbose format, which provides a brief explanation of each function that CHECK INDEX is performing.

VALIDATE If FULL is selected, CHECK INDEX traverses the index and, for each index key, fetches the row from the table that the index references. After fetching the row, CHECK INDEX compares the key in the row to the key in the index to ensure that they match. It also ensures that each index entry points to a unique row and that every row in the referenced table is pointed to by one index entry. This operation can be extremely slow because the rows in the referenced table are fetched and the entire index is traversed.

If a segment name was specified, CHECK INDEX cannot guarantee that every row in the referenced table is pointed to by one index entry; however, it performs all other validations described.

The default is NORMAL, which performs the validation without the row-by-row verification.

DESCRIBE Displays configuration information quickly and terminates without validating the index or calculating the size information. Because CHECK INDEX requires a significant amount of time to validate a very large index, specify DESCRIBE if you are interested only in the configuration information.

Example

The following statement performs the CHECK INDEX operation on the Market table and stores the results in the directory */docs/idx_results*:

```
check index market_pk_idx directory '/docs/idx_results'

INFORMATION
Index: 0 Segment: 2 is ok
Index validation succeeded
```

In this example, more detailed results of the command are stored in the */docs/idx_results* directory.

CHECK TABLE

The CHECK TABLE command checks for and optionally repairs damage to table storage data structures and row counts of tables.

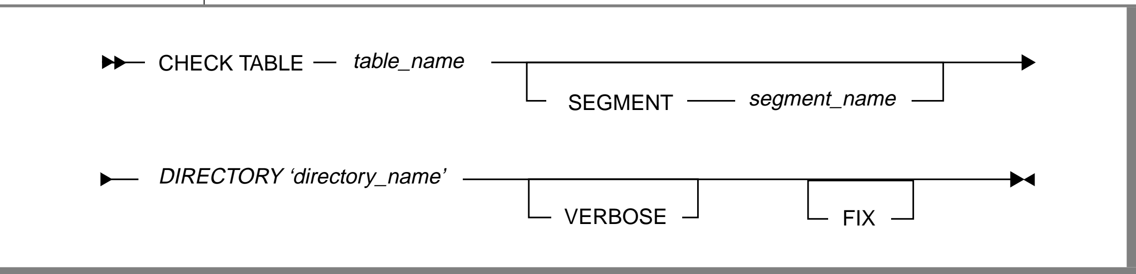
The CHECK TABLE command returns its output as a series of rows. The rows are defined as a single CHAR(1024) column, regardless of the schema of the table being analyzed. Additionally, CHECK TABLE generates a set of text files, which contain the detailed output from the CHECK TABLE operation. These text files are created in a user-specified directory.

Authorization

To issue the CHECK TABLE command, a user must be a member of the DBA system role.

Syntax

The following syntax diagram shows how to construct a CHECK TABLE statement:



table_name	Specifies the name of the table to be checked. The table must be a base table; synonyms, views, temporary tables, and system tables cannot be checked.
SEGMENT segment_name	Specifies a named segment on which CHECK TABLE is to be run. Only one segment can be specified. If *segment_name* is specified, CHECK TABLE reports validation only on that named segment of the table. This option can save considerable time when validation is required on a specific segment of a very large table.
DIRECTORY directory_name	Specifies the directory where the detailed output from the CHECK TABLE operation will be written. A separate file is written for each segment; the filename includes the table identifier, segment identifier, process ID, and a timestamp.

Only the *redbrick* user can read and write these files unless you set the CHECK_REPORT_FILE_PERMISSIONS option in *rbw.config* to allow different file permissions. The possible values are:

- SERVER_OWNER: Gives read/write permission to the *redbrick* user.

  This is the default option.
- SERVER_GROUP: Gives read-only permission to the group.

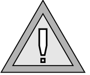
  This value is valid only on UNIX systems.
- ALL: Gives read-only permission to all users.

If the disk containing the directory is not writable or does not exist, an error is returned.

VERBOSE Specifies output to be reported in verbose format in the text files created in the user-specified directory. The VERBOSE option provides a brief explanation of each function CHECK TABLE is performing and provides segment statistics.

You can refer to these text files to determine the effectiveness of the VARCHAR FILLFACTOR value or to troubleshoot a failure reported during the check operation. For more details on interpreting these text files, refer to the *Administrator's Guide*.

FIX Specifies that the CHECK TABLE operation should fix any problems it encounters. If any repairs are made, you must then run the REORG command on the table. If FIX is specified, CHECK TABLE will only run serially.



Warning: *Before performing a CHECK TABLE operation in FIX mode, make sure that a current backup of your database exists and contact Informix Customer Support.*

Example

The following statement performs the CHECK TABLE operation on the Market table and stores the results in the directory */docs/tbl_results*:

```
check table market directory '/docs/tbl_results'

INFORMATION
Table: 1 Segment: 1 is ok
No inconsistencies were detected.
```

In this example, more detailed results of the command are stored in the */docs/tbl_results* directory.

CREATE HIERARCHY

The CREATE HIERARCHY command declares one or more functional dependencies between columns in the same table or from different tables. A functional dependency is a many-to-one relationship that exists between two columns. Hierarchy definitions are used by the Informix Vista query rewrite system to determine which precomputed views can be used to rewrite aggregate queries.

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

For more information about the use of hierarchies, refer to the *Informix Vista User's Guide*.

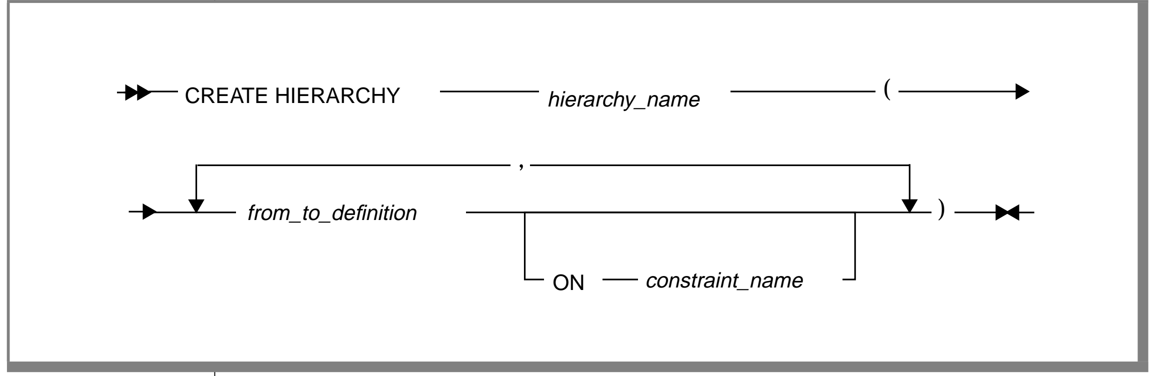
Authorization

To create a hierarchy, a user must meet at least one of the following requirements:

- Be a member of the DBA or RESOURCE system role.
- Have CREATE_ANY authorization, either explicitly or through membership in a user-created role.

Syntax

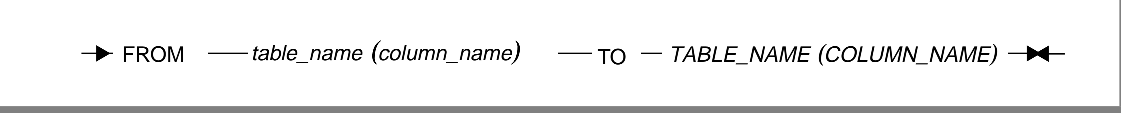
The following syntax diagram shows how to construct a CREATE HIERARCHY statement:



hierarchy_name Specifies the name of the hierarchy, which can refer to one or more functional dependencies. Each hierarchy must have a unique name.

from_to_definition Specifies the tables and columns between which functional dependencies have been established. The relationship can be either between columns in the same table or between columns from two different tables. The first column (the *from* column) must have been declared NOT NULL when the table was created. Otherwise, an error message is displayed.

The following syntax diagram shows how to construct the from_to_definition clause:

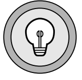


ON constraint_name Identifies the foreign key constraint through which a dependency is defined. The foreign key constraint name specified in the ON clause should be the same as the foreign key constraint name specified in the CREATE TABLE statement.

The ON clause is optional if there is a single foreign-key and primary-key relationship between the named tables. However, if there is more than one such relationship, the ON clause is required. If the relationship is between columns from the same table, the ON clause cannot be used.

A relationship between columns from two different tables must be based on the foreign-key and primary-key relationship. In this case, the hierarchy expresses a functional dependency between the column in the first table and the foreign-key columns in the first table that references the second table. Through this functional dependency, rollups to any column in the referenced table are implied.

Several dependencies can be established under one hierarchy name; however, an independent FROM_TO_DEFINITION is required for each relationship.



Tip: ***You cannot define pairs of columns that roll up to one other column. Hierarchies must be defined from one column to one column.***

Examples

This example declares a functional dependency between columns in the same table.

```
create hierarchy district_region (
    from market (district) to market (region))
```

This example declares a functional dependency between columns in two different tables.

```
create hierarchy store_market (
    from store (store_name) to market (district)
    on store_fkc)
```

This example declares a hierarchy that contains multiple functional dependencies between columns in the same table and across tables.

```
create hierarchy store_market_relationship (
    from store (store_type) to market (district) on store_fkc,
    from store (zip) to market (region) on store_fkc,
    from store (store_name) to store (city))
```

Compare the pairs of values in the following table. If the Period table contains the second set of values, a hierarchy from Qtr to Year would be valid because there is a unique first-quarter value for each year, a unique second-quarter value for each year, and so on. If the Period table contains the first set of values, however, the hierarchy would not be valid because the Qtr column has the same first-quarter value (Q1) for 1998, 1999, and beyond.

Invalid Relationship		**Valid Relationship**	
Qtr Column	**Year Column**	**Qtr Column**	**Year Column**
Q1	1998	Q1_98	1998
Q2	1998	Q2_98	1998
Q3	1998	Q3_98	1998
Q4	1998	Q4_98	1998
Q1	1999	Q1_99	1999
...	...	...	...

CREATE INDEX

The CREATE INDEX command creates one or more indexes in addition to the primary key index that is automatically created for each table. Multiple indexes can be created with a single CREATE INDEX statement, but they must all index the same table.

A STAR, B-TREE, or TARGET index can be created on an empty table or on a table filled with data. An index can be dropped whenever it is not being used by an active query.

A STAR index cannot be created on a temporary table.

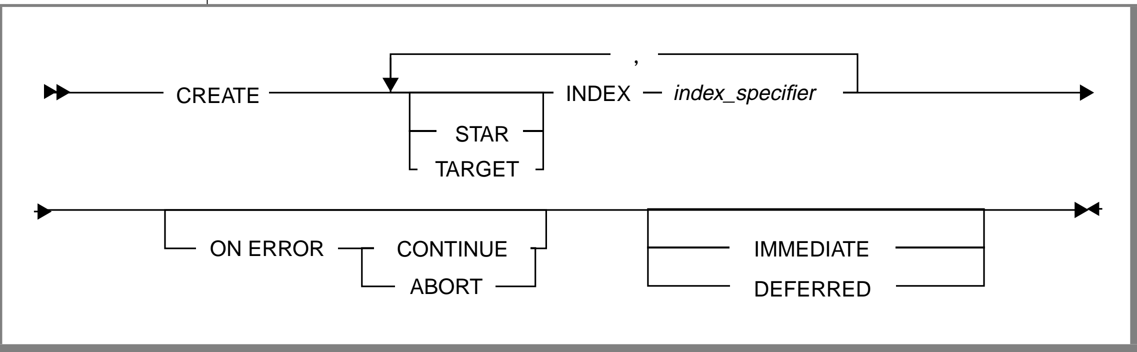
Authorization

To create an index on a table, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have CREATE_ANY authorization, either explicitly or through membership in a user-created role.
- Be a member of the RESOURCE system role and be the creator of the table or have SELECT privilege on the table.
- Have CREATE_OWN authorization and be the creator of the table or have SELECT privilege on the table.

Syntax

The following syntax diagram shows how to construct a CREATE INDEX statement:



STAR Creates a STAR index on the specified foreign keys of the table. Each STAR INDEX on a given table must use a different subset and/or order of foreign keys.

A CREATE STAR INDEX statement will fail if it includes a foreign-key reference to the primary key of another table that was created without a MAXROWS PER SEGMENT value. For more details, see "CREATE TABLE" on page 8-132.

A STAR index cannot be created on a temporary table.

TARGET Creates a TARGET index on the specified column of the specified table.

You can create multiple TARGET indexes on a table, but each one must be created on a single, non-unique column. You cannot create a TARGET index across multiple columns of a table.

A table cannot be unloaded in TARGET index order; an UNLOAD operation can use only a B-TREE or a STAR index.

TARGET indexes improve performance when queries consist of multiple weakly selective constraints. Weak selectivity typically occurs when a column in a very large table has a small *domain* (set of possible values). For example, the domain of a Gender column in an Employees table consists of only two possible values—*Male* or *Female.* Constraints on that column will be weakly selective; they will usually retrieve a very large list of rows.

Much larger domains might also give rise to weak selectivity. For example, an Age column in the same table would have a much larger domain than a Gender column, but constraints on age might still be weakly selective, especially if the data is not uniformly spread across the domain or if the constraints specify values that dominate the domain.

TARGET indexes can also be used to enable TARGETjoin processing. For information about TARGETjoin processing, refer to the *Administrator's Guide*.

B-TREE (default index type)	A B-TREE index is created for a table if you do not specify an index type before the INDEX keyword. This kind of index reduces the search time for a fixed or known value in a condition expressed with a comparison predicate. B-TREE indexes can be created on columns of any datatype. When a table is created, a default B-TREE index is automatically created on the primary key of the table. In the segment specification syntax of the CREATE TABLE statement, a default B-TREE index is referred to as a primary key index. For additional information about selecting indexes for tables, refer to the *Administrator's Guide*.
index_specifier	Specifies the name of the index, the table on which the index is being created, the column name(s) of the index key, and the segment specification. In a temporary table, foreign key constraint and segment specification are not allowed. For the *index_specifier* syntax, see "Index Specifier" on page 8-102.
ERROR	If ON ERROR ABORT is specified, construction of all indexes stops when construction of any one index fails. If ON ERROR CONTINUE is specified or if the ERROR clause is omitted, construction of the other indexes continues when one index fails. This clause is useful only when multiple indexes are created in a single statement; if the clause is omitted, CONTINUE is the default behavior.
IMMEDIATE	Creates and builds the index at the time the command executes.

DEFERRED Creates the index structures and creates an entry in the RBW_INDEXES system table with DEFERRED as the value in the STATE column. The index is not populated with data at this time. The index must be populated with a TMU REORG operation before it can be used.

A CREATE INDEX...DEFERRED operation skips the potentially time-consuming phase of building the index from the data (or deriving it from another index). Everything that can be done without building the index is done; this includes associating the specific segment(s) with the table and various semantic validation checks. This operation is normally followed by a parallel TMU REORG operation, which builds the index more efficiently than the server can.

An index in the DEFERRED state is empty and thus invisible to all functions that read or write the contents of an index. These functions include:

- Queries
- INSERT, UPDATE, and DELETE operations.
- Load and unload operations.

Additionally, deferred indexes are not visible to the compiler; therefore, query plans cannot be generated for them.

An index in the DEFERRED state is visible to those functions that operate only on the structure of an index. In general, these functions make no distinction between DEFERRED indexes and fully built indexes. DEFERRED indexes are visible to the following functions:

- DROP INDEX: The index can be dropped.
- ALTER INDEX: All operations are allowed.
- DROP TABLE: Deletes all indexes in the table including the DEFERRED indexes.
- ALTER TABLE DROP COLUMN: This operation is prohibited if the specified column is a component of the DEFERRED index.
- CREATE INDEX: Another index with the same name or the same type cannot be created on the same columns.
- ALTER SEGMENT: All operations allowed on non-DEFERRED indexes are allowed on DEFERRED indexes.

Examples

The following CREATE INDEX statement creates two indexes in parallel, both on the Market table and one in a user-created segment. If an error occurs during creation of the first index, neither index is created.

```
create index mkt_city_idx
    on market (city)
    in mkt_idx_seg,

index mkt_district_idx
    on market (district)
on error abort
```

The following statement creates a mixed-domain TARGET index on the Color column of the Car_Model table:

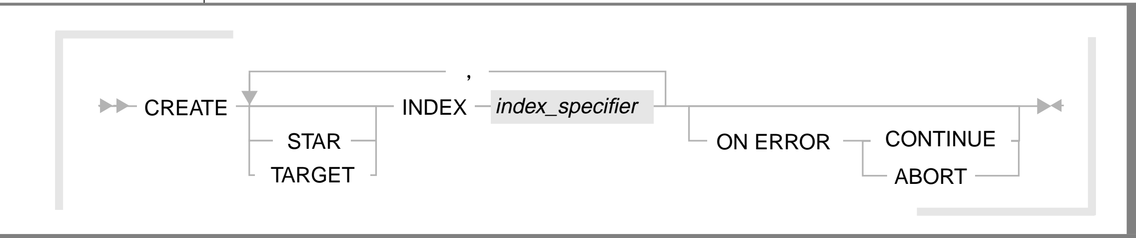
```
create target index tgt_idx1
    on car_model (color)
    in sgmt_1
```

Index Specifier

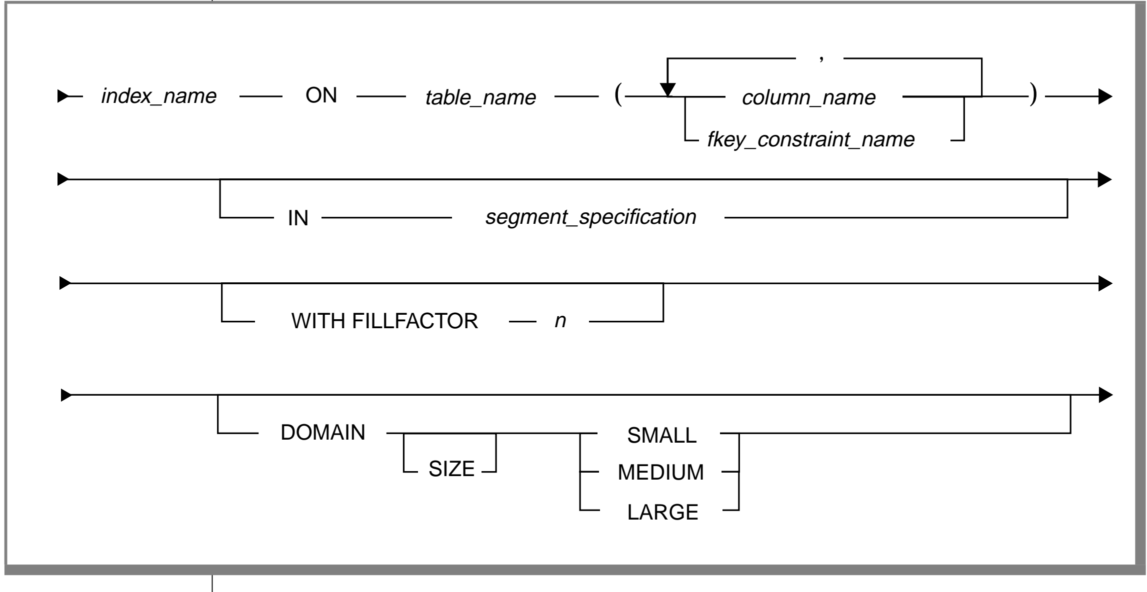
The *index_specifier* specifies the name of the index, the table on which the index is being created, the column name(s) of the index key, and the segment information.

Syntax

The following syntax diagram for CREATE INDEX is repeated to provide a point of reference for the *index_specifier* syntax:



The following syntax diagram shows how to construct an *index_specifier*:



index_name	Specifies the name of the index. Each index name must be a unique index name in the database.
table_name	Specifies the name of the table for which the index is being created. A temporary table name can be specified when creating a TARGET or B-TREE index, but not when creating a STAR index. A table that is foreign key–referenced by a STAR index must have a MAXROWS PER SEGMENT value specified in its CREATE TABLE statement.
column_name	Specifies the index key for TARGET and B-TREE indexes: ■ For B-TREE indexes, if more than one column is specified, each column must be defined as NOT NULL in the CREATE TABLE statement. ■ For TARGET indexes, only one non-unique column can be specified.
column_name, fkey_constraint_name	Specifies the index key for STAR indexes. A STAR index key is composed of foreign keys, which can be identified in two different ways: ■ With column names. Each column name must identify a *distinct, single-column* foreign-key reference in the CREATE TABLE statement. ■ With foreign-key constraint names, which can identify foreign keys that consist of one or more columns. (Constraint names are defined in the CREATE TABLE statement; for details, see "Primary-Key and Foreign-Key Constraint Names" on page 8-144.)

Although the index key can mix column names and constraint names, it is advisable to use one format consistently, with regard to the following rules:

- If a foreign key references a table with a single-column primary key, the name of the foreign-key column is sufficient to define the column in the index key; however, if a foreign key references a table with a multi-column primary key, the foreign-key constraint name for the set of columns must be specified instead of a series of individual column names.
- Only *user-defined* foreign-key constraint names are allowed in CREATE STAR INDEX statements.
- Each name entered in a CREATE STAR INDEX statement is first assumed to be a constraint name. If a matching constraint name exists, the foreign key that it identifies is used as the index key. Otherwise, the name is assumed to be a column name. If no match is found, the CREATE STAR INDEX statement will fail.

A foreign key is a column that is defined as NOT NULL and listed in the FOREIGN KEY REFERENCES clause of the CREATE TABLE statement. Any subset or ordering of the foreign keys is allowed in a STAR index key, unless the intent is to produce a simple star schema. See page 8-172 for further information. The order in which the foreign keys are listed determines their order in the index key. The first name represents the leading key column, the second the next leading key column, and so on.

Examples—Creating STAR Indexes

Assume the Sales table was created as follows. The default B-TREE index would be created on the primary key columns Perkey, Classkey, Prodkey, Storekey, and Promokey:

```
create table sales (
    perkey integer not null,
    classkey integer not null,
    prodkey integer not null,
    storekey integer not null,
    promokey integer not null,
    quantity integer,
    dollars dec(7,2),
    constraint sales_pkc primary key (perkey, classkey,
        prodkey, storekey, promokey),
    constraint sales_date_fkc foreign key (perkey) references
        period (perkey),
    constraint sales_product_fkc foreign key
        (classkey,prodkey)
        references product (classkey, prodkey),
    constraint sales_store_fkc foreign key (storekey)
    references store (storekey),
    constraint sales_promo_fkc foreign key (promokey)
    references promotion (promokey))
    data in (daily_data1, daily_data2)
        segment by values of (perkey)
        ranges (min:415, 415:max)
    maxsegments 2
    maxrows per segment 50000
```

A STAR index can be created on any order or subset of the foreign keys; however, to form a *simple star schema*, where the STAR index is built on all the foreign-key columns that make up the primary key, the default B-TREE index must be dropped. (It can be dropped before or after the STAR index is created.)

The following CREATE INDEX statement creates a STAR index on the Perkey column to improve queries that constrain that column the most frequently.

```
create star index time_sales_ix
on sales (sales_date_fkc)
in (sales_seg_ix1, sales_seg_ix2)
...
```

In this case, the foreign key is identified by the constraint name *sales_date_fkc*, although it would also be possible to use the column name, since the referenced table has a single-column primary key:

```
create star index time_sales_ix
on sales (perkey)
in (sales_seg_ix1, sales_seg_ix2)
...
```

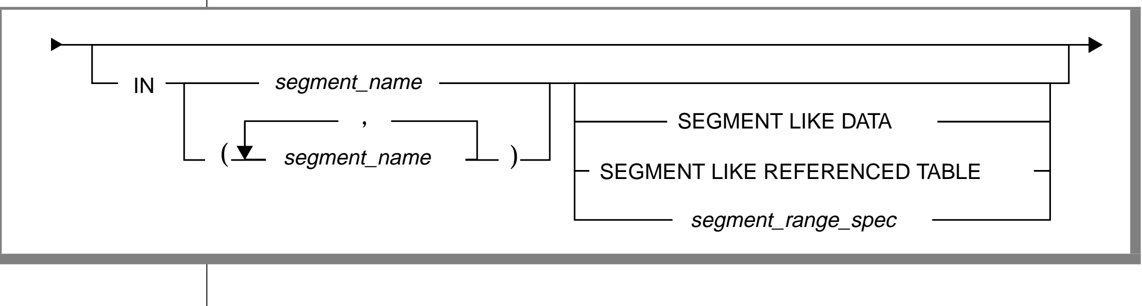
The following CREATE INDEX statement creates an index on the Classkey and Prodkey columns to improve queries that constrain on those columns the most frequently.

```
create star index prod_per_sales_ix
on sales (sales_product_fkc)
in (sales_seg_ix3, sales_seg_ix4)
    segment by references of (sales_product_fkc)
    ranges (...)
```

In this case, the foreign key references a table with a multi-column primary key (the Product table), so the constraint name *sales_product_fkc* must be used to define the index key. (Note that the segmenting column is also identified by the constraint name in this case.) For more information about constraint names, see "Primary-Key and Foreign-Key Constraint Names" on page 8-144.

Segment Specification

The following syntax diagram shows how to construct a segment specification:



segment_name	Specifies the name of the segment(s) in which the index will reside. If no segment names are specified, the index resides in a default segment. Specified segments cannot be attached to any other index or table.
	If a single (user-defined or default) segment is specified, additional segments can later be assigned to the index with the ALTER SEGMENT command.
	In Red Brick Decision Server for Workgroups databases and in temporary tables, only one segment name can be specified, so the segment range specification does not apply.
SEGMENT LIKE DATA	Specifies that the segment range specification for a B-TREE or TARGET index is identical to the segment range specification for the table data. This option is valid only if:

- The data is segmented by values, not by hashing.
- The leading column of the index is the same as the segmenting column for the data.
- The same number of segments is specified for the index and for the data.

This option does not apply to STAR indexe*s.*

SEGMENT LIKE REFERENCED TABLE	Specifies that the segment range specification for a STAR index is identical to the segment range specification for the referenced table. This option defines a one-to-one correspondence between the segments of the index and those of the referenced table; therefore, the range values used to segment the referenced table also define the segmentation of the STAR index.

Before using the SEGMENT LIKE REFERENCED TABLE option, note the following conditions:

- This option is valid only if exactly one *segment_name* is specified for each segment in the table referenced by the first foreign key listed in the *index_specifier*.
- This option reflects the referenced table's segmentation scheme *statically*. If that scheme is changed, the changes are not automatically reflected in the definition of the STAR index. Any changes made to the referenced table must also be explicitly made to the STAR index.
- This option does not apply to B-TREE and TARGET indexes.

Example

The following CREATE INDEX statement defines an index for the Market table and specifies a named (user-created) segment:

```
create index mkt_state_idx
    on market (state)
    in mkt_state_idx_seg
```

Segment Range Specification

Specifies the segmenting column and range of values to be distributed among each segment.

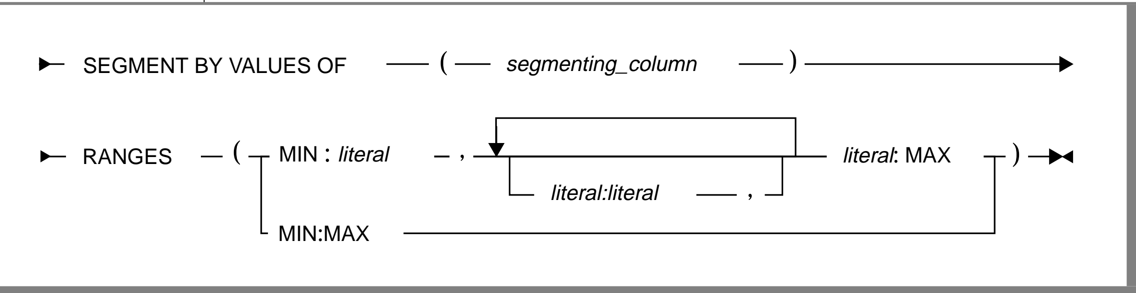
TARGET and B-TREE indexes are segmented according to index key values. STAR indexes are segmented based on row IDs of the referenced table; the segment range specification assigns the segmenting column (from the referencing table) and the range of rows that will be distributed among the segments.

To see how the segment range specification relates to the *index_specifier*, refer to page 8-102.

Tip: *In Red Brick Decision Server for Workgroups databases, the segment range specification does not apply and cannot be used.*

B-TREE and TARGET Indexes

The following syntax diagram shows how to construct a *segment_range_spec* for a B-TREE index or a TARGET index. This specification is required only if the index resides in more than one segment.



SEGMENT BY VALUES OF segmenting_column

The *segment_range_spec* for B-TREE and TARGET indexes must observe the following conditions:

- The *segmenting_column* must be the first column in the *index_specifier*.
- A range must be specified for each segment defined in the segment specification.
- The *literal* values must identify values from the *segmenting_column*; the *literal* and *segmenting_column* values must be of the same datatype.

RANGES (MIN:MAX)

Specifies distribution of the index among the segments based on index key values.

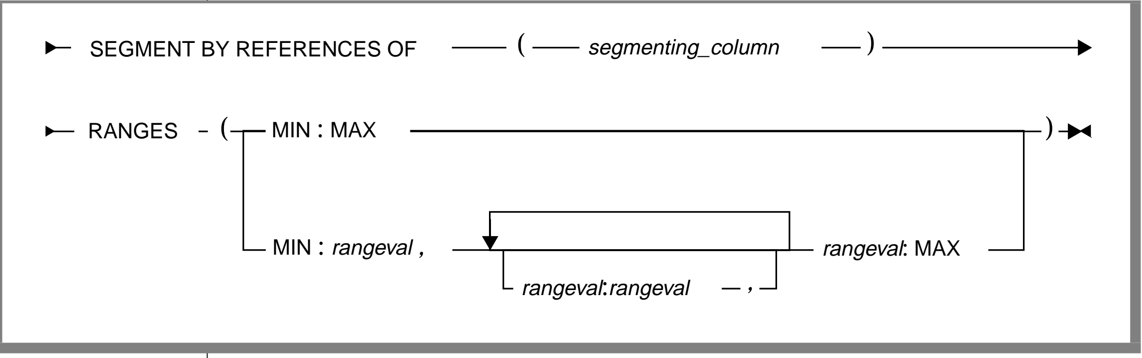
Ranges must observe the following conditions:

- Separate the lower and upper value of each pair with a colon (:).
- Separate each pair of values from the next with a comma.
- Start with the MIN keyword (which indicates the lowest key value) and end with the MAX keyword (which indicates the highest key value).
- Be in ascending order and not have any overlaps or gaps. The upper range of one segment must be the lower range of the next segment. The values entered into each segment are greater than or equal to the lower range and less than the upper range.

STAR Indexes

The following syntax diagram shows how to construct a ***segment_range_spec*** for a STAR index. This specification is optional.

If a range is not specified, the range is calculated automatically, based on the number of segments in the STAR index and the MAXSEGMENTS and MAXROWS PER SEGMENT values of the first table referenced in the index key. The range is divided evenly across the segments.



SEGMENT BY REFERENCES OF ***segmenting_column***	Specifies the segmenting column or foreign key constraint name, which is used to determine how the index is distributed among the segments. The following rules define the ***segmenting_column*** specification:

- Specifying a segmenting column (and ranges) is optional; ranges for multiple segments are calculated automatically if you do not include this clause.
- If the first foreign key in the STAR index key consists of one column, that column must be specified as the only segmenting column (using either the column name or the foreign key constraint name).
- If the first foreign key consists of two or more columns, you must use the foreign key constraint name to identify these columns (as defined in the FOREIGN KEY REFERENCES clause of the CREATE TABLE statement).
- The table referenced by the segmenting column must have an assigned MAXROWS PER SEGMENT value.

RANGES (MIN:MAX)	Specifies distribution of the STAR index among the segments based on row numbers (or a combination of segment names and row numbers) in the first table referenced in the index key. The first row number is 0, and the highest row number must be less than the value of MAXROWS PER SEGMENT. The range must be specified for each segment of the STAR index if the segmenting column is specified. If the segmenting column is not specified, the ranges are calculated automatically and must not be specified.

To determine the segment names and row numbers of a referenced table for specifying a range of a STAR index, issue the following query:

```
select primary_key, rbw_segname, rbw_rownum
from table_name;
```

The previous query returns all rows of the referenced table; if the table is large, the result set will be large.

Ranges must observe the following general conditions:

- Separate the lower and upper value of each pair with a colon (:).
- Separate each pair of values from the next with a comma.
- Start with the MIN keyword (which indicates the first row in the first segment of the first referenced table) and end with the MAX keyword (which indicates the last row in the last segment of the first referenced table).
- Be in ascending order and not have any overlaps or gaps. The upper range of one segment must be the lower range of the next segment. The values entered into each segment are greater than or equal to the lower range and less than the upper range.

rangeval

There are two ways to define the specific range values for the segments of a STAR index. The *rangeval* (range value) variable can represent either of the following:

- *rownum*

  Use this kind of range value when the table referenced by the segmenting column resides in one segment. Each row number (*rownum*) identifies a single row in the referenced table. For example, one pair of range values might be:

  ```
  100:200
  ```

- *segname rownum*

  Use this kind of range value when the table referenced by the segmenting column resides in multiple segments. Each segment name (*segname*) identifies a segment attached to the referenced table, and each row number (*rownum*) identifies a single row of the referenced table stored in that segment. For example, one pair of range values might be:

  ```
  seg1 100:seg1 200
  ```

  The *rownum* part of the range value is optional; if it is omitted, the minimum row number of the named segment is assumed.

Examples—Segment Specifications and Ranges

The following statement defines a STAR index for the Sales table in three named segments. The index key is defined to maximize the performance of queries that constrain the Perkey and Mktkey columns. The Perkey column must be the segmenting column because the Period table is the first table referenced in the index key. The range references row IDs of the Period table.

```
create star index sales_mkt_prod_idx
    on sales (perkey, mktkey)
    in (sales_idx_seg1, sales_idx_seg2, sales_idx_seg3)
        segment by references of (perkey)
        ranges (min: 300, 300:600, 600:max)
```

The following statement defines a STAR index for the Orders table in a named segment with no segmenting column. An additional segment can be attached to this index later with the ALTER SEGMENT command.

```
create star index sales_cust_prod_idx
    on orders (custkey, prodkey)
    in orders_idx_seg
```

The following specification is for a single-segment B-TREE index. The range must be specified as (min:max).

```
segment by values of (perkey)
ranges (min:max)
```

The following specification is for a multi-segment B-TREE index. The segmenting column (Perkey) is a datetime column, so the range values are date values:

```
segment by values of (perkey)
ranges (min: '1-01-1998', '01-01-1998':'01-01-1999',
    '01-01-1999':'01-01-2000', '01-01-2000': max)
```

Notice that the upper range of the first segment is the lower range of the second segment. When data is inserted into the index, rows referenced by pre-1998 date values are inserted into the first segment, 1998 values into the second segment, 1999 values in the third segment, and 2000 values and beyond into the last segment.

The following specification is for a multi-segment STAR index that references a single-segment table; the numbers specified in each range represent row IDs for the Perkey column, not Perkey values:

```
segment by references of (perkey)
ranges (min:100, 100:200, 200:500, 500:max)
```

The following specification is for a multi-segment STAR index that references a multi-segment table:

```
...
in (ix_seg1, ix_seg2, ix_seg3, ix_seg4)
segment by references of (mktkey)
ranges (min:tab_seg1 1000, tab_seg1 1000:tab_seg2 2000,
    tab_seg2 2000:tab_seg3 3000, tab_seg3 3000:max)
```

In the last two examples, a range is specified for each segment defined in the segment specification.

WITH FILLFACTOR n

Specifies the percentage of space to initially fill in each index node. As rows are inserted later, the index nodes continue to fill until they reach 100% of capacity. If the index nodes need to fill beyond 100%, they split to accommodate the overflow. A fill factor that has been set correctly allows many insert and update operations to occur without node splitting.

Legitimate values for *n* range from 1 to 100; however, fill factors should generally be greater than 50%.

If an index is created on a table that contains data, the fill factor setting is used upon index creation to determine the space available in each index node. If an index is created on an empty table, the fill factor setting is used when data is loaded into the table. The fill factor setting is also used when data is reloaded into a table or a table is reorganized with the REORG option. For the fill factor setting to be used when data is loaded or reorganized, the TMU optimize option must be on.

If the WITH FILLFACTOR option is not specified, the default fill factor, which is set in the *rbw.config* file, is used. The original default set in the *rbw.config* file for each type of index is 100%. After an index has been created, the fill factor setting can be changed with the ALTER INDEX command.

For TARGET indexes with SMALL domain sizes, always set the fill factor to 100% (the default). For indexes with MEDIUM or LARGE domain sizes, set the fill factor to 100% unless you plan to update or delete rows; in this case, use a lower percentage.

For more information about fill factors, refer to the *Administrator's Guide*.

Example

In the following example, a fill factor of 60 is set, which initially fills each index node to only 60% of capacity.

```
create index promotion_idx
    on promotion (promo_type)
    with fillfactor 60
```

DOMAIN SIZE

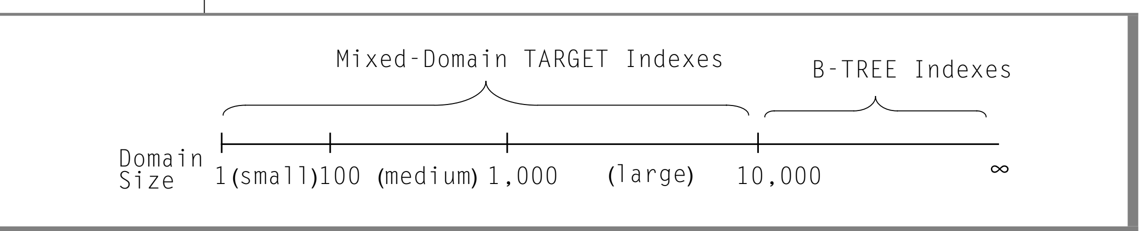
Specifies (optionally) the domain size of the indexed column for TARGET indexes: SMALL, MEDIUM, or LARGE. The term ***domain size*** refers to the number of unique values in the indexed column.

The DOMAIN SIZE clause applies only to TARGET indexes.

Based on your choice, Red Brick Decision Server selects the appropriate storage method, or "representation" for the index. If you do not specify a domain size, the server chooses the appropriate representation for each distinct value in the indexed column, creating a mixed-domain TARGET index with optimized processing speed and storage. You should only specify a domain size when one size suits most or all of the values in the column—for example, because the data is uniformly spread across the domain or the domain is very small. In most cases—and especially when the spread of values in the column is skewed or unknown—the default mixed-domain approach is the best choice.

Figure 8-10 illustrates approximate boundaries, in terms of actual domain size, for using mixed-domain TARGET indexes versus B-TREE indexes, which are more appropriate for domains that exceed 10,000 distinct values. How actual domain sizes should map to SMALL, MEDIUM, and LARGE domain specifications is more difficult to judge, but some general guidelines are presented.

Figure 8-10
Domain-Size Boundaries



The domain size SMALL tends to provide the best performance; however, it requires the most space, growing linearly as the domain size grows.

TARGET indexes can also be used to enable TARGETjoin processing. For information about TARGETjoin processing, refer to the *Informix Red Brick Decision Server Administrator's Guide*.

Examples

If a large Demographics table contains an Occupation column with approximately 5,000 distinct values, but those values are known to be skewed to the extent that a small number of those values account for more than half of the table's rows, a mixed-domain TARGET index would be the best choice:

```
create target index code_tgt_ix
    on promotion(promo_code)
```

If a Marital_Status column in the same table is known to contain fewer than 10 distinct and fairly evenly distributed values (*Single*, *Married*, *Divorced*, and so on), a TARGET index on that column might best be specified as follows:

```
create target index city_tgt_ix
    on store(city)
    domain size small
```

CREATE MACRO

The CREATE MACRO command creates an abbreviation (a macro name) for a partial or complete SQL statement. The CREATE MACRO statement can contain an optional category to record the relationship of the macro to the overall SQL statement and a descriptive comment about the macro.

Authorization

To create a PUBLIC macro, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have CREATE_ANY or PUBLIC_MACROS authorization, either explicitly or through membership in a user-created role.

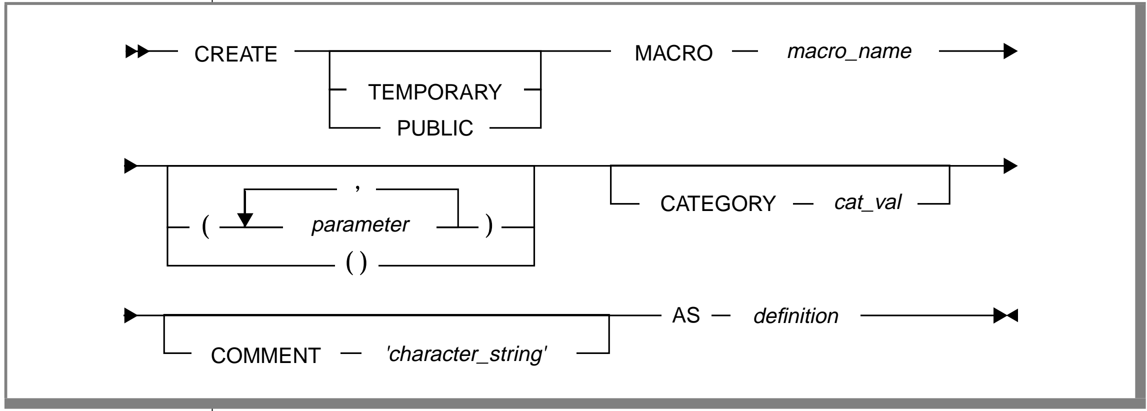
To create a private macro, a user must meet at least one of the following requirements:

- Be a member of the DBA or RESOURCE system role.
- Have CREATE_ANY or CREATE_OWN authorization, either explicitly or through membership in a user-created role.

Any user currently connected to a Red Brick Decision Server database can create a TEMPORARY macro.

Syntax

The following syntax diagram shows how to construct a CREATE MACRO statement:



TEMPORARY, PUBLIC

A macro can be temporary, public, or private:

- A temporary macro name is created during a user's session (a connection to a Red Brick Decision Server database), exists only for the duration of that session, and can be accessed only by its creator. If a client tool disconnects from the server during an interaction, temporary macros will be dropped.
- A public macro name is created independently of the user's session, resides in the RBW_MACROS system table, and exists until removed by a DROP PUBLIC MACRO command. It is accessible by all users.
- If neither temporary nor public is specified, a private macro is created independently of the user's session, resides in the RBW_MACROS system table until dropped by its creator, and can only be accessed by its creator.

For additional information on the PUBLIC macros and where they reside, refer to the *Informix Red Brick Decision Server Administrator's Guide*.

macro_name, parameter Macro and parameter names must be database identifiers but cannot be keywords. The same macro name can abbreviate a temporary, a public, or a private macro. Any potential ambiguity is resolved when the macros are read or created by the server during a session. The rules are as follows: If a temporary macro with the name exists, it is used. If no temporary macro exists and if a private macro with the name exists, then the private macro is used. If no temporary or private macro exists and a public macro with the name exists, then the public macro is used.

A macro name can occur as part of an SQL statement or within another macro. During the execution of the SQL statement, each occurrence of the macro name is replaced by its *definition*.

Macro names that contain delimited identifiers must be surrounded by double quotes, and each double quote around a delimited identifier must be escaped with double quotes. See page 8-124 for an example.

CATEGORY Specifies the syntax category for the macro. This optional parameter allows a category value to be recorded with the macro to define how the macro should be used in an SQL statement.

This value is stored in the RBW_MACROS system table. The value can be retrieved by querying the RBW_MACROS table, but is not otherwise examined or processed by the server. The minimum value is zero, and the maximum value is 65,535. If this optional parameter is not specified or if a macro is updated from a previous release of Red Brick Decision Server, NULL is stored in the Category column of RBW_MACROS.

Informix reserves all values from 0 through 255 for common category definitions. To establish a category not already defined by Informix, choose a value greater than 255.

The defined values for *cat_val* are listed in the following table:

Category Value	Category Definition
1	Select list item. The macro defines a calculated column that can be specified in a query select list.
2	Search condition. The macro defines one or more search conditions, which can be included in the WHERE clause of a SELECT statement.
3	Sort term. The macro defines items that can be used in an ORDER BY clause.
10	Value. The macro specifies the text of either a literal value or a subquery value.
11	Value List. The macro specifies text of multiple literal values.
100	Complete SQL statement. The macro defines a syntactically complete SQL statement.
101	General. The macro defines SQL text that is undefined by another category.

COMMENT character_string

Assigns a descriptive comment string about the meaning or use of the macro. The string can contain up to 256 bytes. This optional parameter is stored in the RBW_MACROS system table and can be retrieved by querying that table.

If this parameter is not specified or if a macro is updated from a previous release of Red Brick Decision Server, NULL is stored in the COMMENT column of RBW_MACROS. The server makes no use of this text.

The contents of the COMMENT column of RBW_MACROS can be updated with the ALTER MACRO command.

AS definition

Defines the macro. The definition can be a partial or complete SQL statement and can contain:

- Macro and parameter names (referred to as embedded macros).
- No more than 1,024 bytes (each parameter specified uses 3 bytes, regardless of the size of the parameter name or the size of the value supplied when the macro is executed).
- No more than one complete statement.
- The escape character backslash (\) when a parameter includes a comma (,).

When the macro name is expanded during execution, each occurrence of a parameter name in the definition is replaced by its corresponding argument value. The EXPAND command is a pseudo-select command that returns an instance of a macro. For additional information, see "EXPAND" on page 8-184.

Examples

The following example abbreviates a complete SELECT statement and specifies the syntax category of the macro:

```
create macro select_star_lotta
    category 100
    as select * from product
    where product like 'Lotta%'
```

The following example defines a condition with parameters and specifies the syntax category of the macro and a comment:

```
create macro mo_sales(prod_name, mo, yr)
    category 2
    comment 'Search condition with parameters.'
    as where product like prod_name
        and month = mo
        and year = yr
```

A SELECT statement could reference this sales macro name as follows:

```
select prod_name, city, dollars
from store natural join sales
    natural join product
    natural join period
mo_sales('Lotta%', 'MAR', 1998)
```

The query returns sales for March 1998 for the product Lotta Latte. For additional examples using macros, refer to the *SQL Self-Study Guide*.

The following example creates a macro that expands to a CREATE TABLE statement that uses delimited identifiers:

```
create macro create_star as
          "create table ""table"" (""The """"STAR"""""" int)"
```

Entering *create_star* at the RISQL prompt creates a table named *table* with a column named *The "STAR"*.

CREATE ROLE

The CREATE ROLE command creates a role and optionally grants the role to one or more users and/or roles. After creating a role, use the GRANT command to grant task authorizations and object privileges to the role.

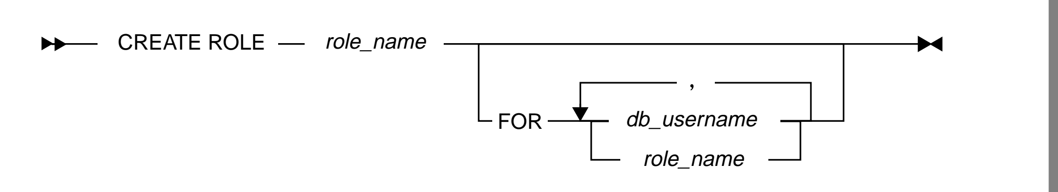
Authorization

To create a role, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have ROLE_MANAGEMENT authorization either explicitly or through membership in a user-created role.

Syntax

The following syntax diagram shows how to construct a CREATE ROLE statement:



role_name	Identifies the role being created. A role name must be a valid identifier and must be different from all other role names, database usernames, and task authorization names.
FOR db_username	Grants the new role to a database user. The user becomes a member of the new role and has all task authorizations and object privileges of the role.
FOR role_name	Grants the new role to an existing user-created role. This role cannot be the DBA or RESOURCE system role because system roles cannot be altered.

Usage Notes

The database users and roles specified in the FOR clause become direct members of the new role. A user or role can be a direct member of no more than 16 roles. If any user or role specified in the FOR clause is already a member of 16 other roles, the role is not created.

Examples

The following statement creates the *temp* role:

```
create role temp
```

The following statement creates the *contractor* role and grants the role to *jerry*:

```
create role contractor for jerry
```

The following statement creates the *market_research* role and grants the role to *alison*, *emily*, and *paul*:

```
create role market_research for alison, emily, paul
```

CREATE SEGMENT

A CREATE SEGMENT command defines a named segment, which contains one or more physical storage units (PSUs).

A table or an index can reside in a default segment or in one or more named segments. Default segments are created automatically when tables or indexes are created and named segments are not specified. Named segments are created with the CREATE SEGMENT command and assigned to a table or index with the CREATE TABLE, ALTER TABLE, CREATE INDEX, or ALTER SEGMENT command. Only one database object can reside in a named segment; however, any table or index can reside in multiple segments.

For information about creating the backup segment, refer to "ALTER DATABASE" on page 8-6, and to the *Informix Red Brick SQL-BackTrack User's Guide*.

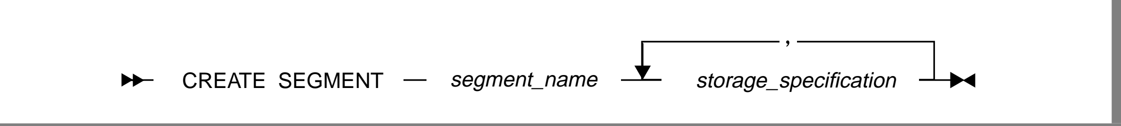
Authorization

To create a segment, a user must meet at least one of the following requirements:

- Be a member of the DBA or RESOURCE system role.
- Have CREATE_ANY or CREATE_OWN authorization, either explicitly or through membership in a user-created role.

Syntax

The following syntax diagram shows how to construct a CREATE SEGMENT statement:



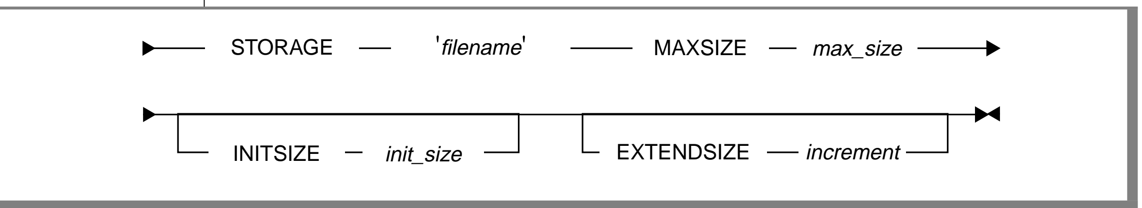
segment_name Specifies the name of the segment being created. This name must not occur in the current system catalog and must be a valid database identifier.

storage_ specification Specifies the pathnames and sizes for each PSU that will contain row data or index data for a segment. For the syntax of the storage specification, refer to the next section.

The *storage_specification* specifies the pathnames and sizes for each file, which are referred to as a physical storage units (PSUs), assigned to a segment. A segment can contain multiple PSUs, each with a different size.

Syntax

The following syntax diagram shows how to construct a *storage_specification*:



STORAGE filename — Specifies the filename of the PSU. The filename can be either a pathname relative to the database directory or an absolute pathname. All specified directories must exist.

MAXSIZE max_size — Specifies the maximum number of kilobytes of data that will be loaded into the PSU before the next PSU in the sequence is used. The value is specified in 1 kilobyte blocks and rounded up to the next multiple of 8 kilobytes. The lowest valid MAXSIZE value is 16 kilobytes.

The MAXSIZE value is stored in the MAXSIZE column of the RBW_STORAGE system table. This system table also contains the number of kilobytes that have been used in a PSU (USED column).

INITSIZE init_size

Specifies the amount of initial space pre-allocated for the PSU. The value is specified in kilobytes and rounded up to the next multiple of 8 kilobytes. The value must be less than or equal to MAXSIZE. The default is 16 kilobytes.

The initial size of the first PSU in a segment is always at least 16 kilobytes. If an initial size between zero (0) kilobytes and 9 kilobytes is specified for the first PSU, the server returns an error. If an initial size between 9 kilobytes and 16 kilobytes is specified, the value is rounded up to 16 kilobytes. The initial size of subsequent PSUs can be from zero (0) to the maximum size.

The INITSIZE value is stored in the INITSIZE column of the RBW_STORAGE system table.

EXTENDSIZE increment

Specifies the amount the PSU expands beyond the initial size each time it becomes full and needs to expand. The value is specified in 1 kilobyte blocks and rounded up to the next multiple of 8 kilobytes. The default is 8 kilobytes.

Example

The following CREATE SEGMENT statement defines a segment that consists of three PSUs. Each PSU can contain up to one megabyte (MAXSIZE) of data.

The first PSU, ***sales_area1***, is given 104 kilobytes of storage (INITSIZE) when the segment is created. Any time ***sales_area1*** becomes full, it expands in 104 kilobyte increments (EXTENDSIZE) until it reaches the maximum size. After ***sales_area1*** reaches the maximum size, data is stored in ***sales_area2***.

The ***sales_area2*** and ***sales_area3*** PSUs are given 16 kilobytes of storage when the segment is created, which is the default of INITSIZE. First, ***sales_area2*** expands in 8-kilobyte increments until it reaches the maximum size. Then, ***sales_area3*** expands in 8-kilobyte increments until it reaches the maximum size.

```
create segment sales_dataseg1
    storage 'sales_area1'
        maxsize 1024
        initsize 100
        extendsize 100,
    storage 'sales_area2'
        maxsize 1024,
    storage 'sales_area3'
        maxsize 1024
```

CREATE SYNONYM

A synonym is a logical name or an alias for an existing table. After a synonym has been defined, it can be used as if it were a copy of the original table. Synonyms can be created for any base table in the database, but not for views or temporary tables.

Authorization

To create a synonym for a table, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have CREATE_ANY authorization, either explicitly or through membership in a user-created role.
- Be a member of the RESOURCE system role and be the creator of the table.
- Have CREATE_OWN authorization and be the creator of the table.

Syntax

The following syntax diagram shows how to construct a CREATE SYNONYM statement:

```
►►─ CREATE SYNONYM ─ synonym_name ── FOR ─ table_name ───►◄
```

synonym_name Specifies the name of the synonym, which must be a database identifier that is different from any other view, table, or synonym identifier in the database.

table_name Specifies an existing database table for which the synonym is being created. Synonyms cannot be created for views or temporary tables.

Usage Note

Synonyms are simply a means of creating permanent aliases for table names. You do not need to create synonyms in order to write queries in which a fact table makes multiple references to the same dimension table; any table can make more than one primary-key or foreign-key reference to another table.

Example

The following example creates a synonym, Shipdate, for the Period table:

```
create synonym shipdate for period
```

CREATE TABLE

A CREATE TABLE statement defines a base database table (not a synonym) with a primary key, attributes, any foreign key references, and segments. When you create a table, a B-TREE index is automatically created on the table's primary key.

For the syntax of the CREATE TEMPORARY TABLE command, refer to page 8-154.

Authorization

To create a table, a user must meet at least one of the following requirements:

- Be a member of the DBA or RESOURCE system role.
- Have CREATE_ANY or CREATE_OWN authorization, either explicitly or through membership in a user-created role.

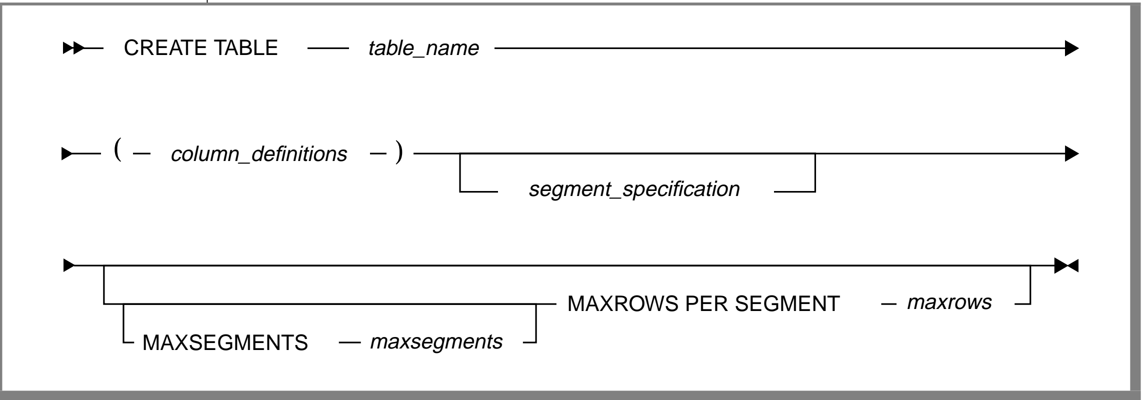
To create a table in another user's segment, a user must meet one of the following requirements:

- Be a member of the DBA system role.
- Have CREATE_ANY authorization.

Each table in a Red Brick Decision Server for Workgroups database must reside in a single segment and is limited to a maximum storage of 5 gigabytes of data.

Syntax

The following syntax diagram shows how to construct a CREATE TABLE statement:



table_name — Specifies the name of the base table being created, which must be:

- Different from any other view, table, or synonym in the database
- A valid database identifier

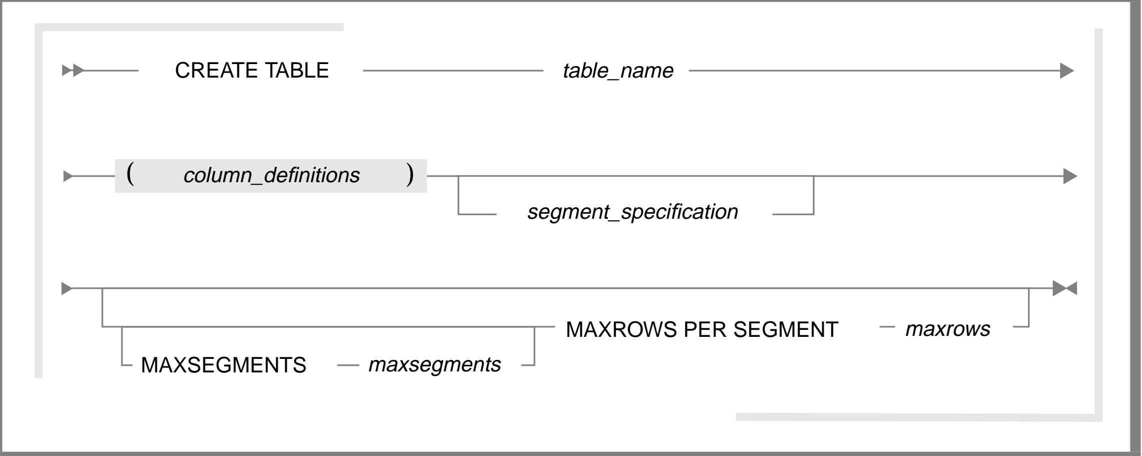
column_definitions	Specifies column definitions for the columns in the table. Each column in the database table must be completely defined with a column definition. For a description of column definitions, see page 8-135.
segment_specification	Assigns segments to the table's data and system-generated B-TREE index. If the segment specification is omitted from the CREATE TABLE statement, the data and index each reside in a default segment. Named segments must be created with the CREATE SEGMENT command before the tables and indexes that reside in them are created. For the syntax and a description of segment specifications, refer to page 8-145.
MAXSEGMENTS maxsegments, MAXROWS PER SEGMENT maxrows	MAXSEGMENTS represents the maximum number of segments in which a table can reside. MAXROWS PER SEGMENT represents the maximum number of rows in each of the table's segments. These values are used to calculate the size of the STAR index key when the table is referenced by the foreign key of an indexed table. If MAXSEGMENTS is not specified, the value of MAXSEGMENTS defaults either to the number of segments named in the *segment_specification* or to 1 if no *segment_specification* is given. If MAXROWS PER SEGMENT is not specified, it will not be possible to define a STAR index on the foreign key of another table that references the first table's primary key. The CREATE INDEX statement will fail.

Column Definitions

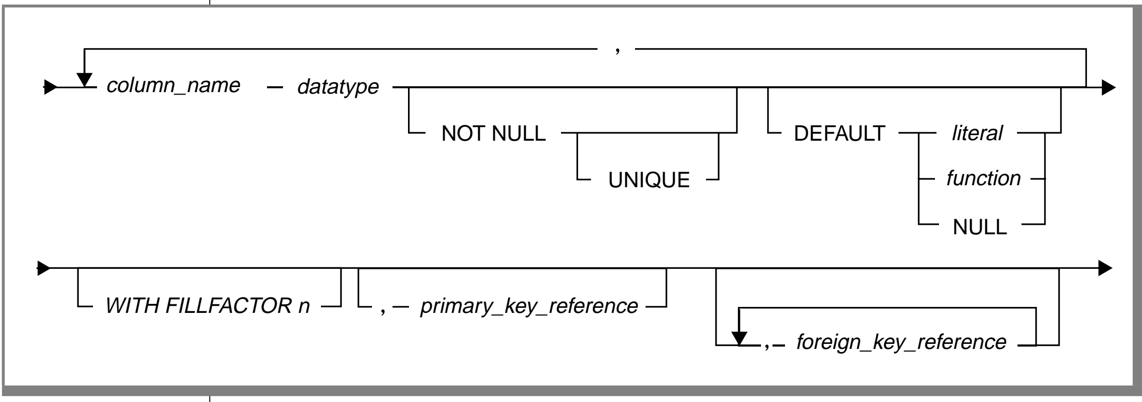
Each column in a database table must be completely defined in the CREATE TABLE statement with a column definition.

Syntax

The entire CREATE TABLE statement syntax diagram is repeated to provide a point of reference for the *column_definitions* syntax:



The following syntax diagram shows how to construct a *column_definition*:



column_name Specifies the name of a column in the table. The name must be unique within the table, must be a valid database identifier, and must not be qualified. A table can have a maximum of 7,280 columns.

A column can be specified as NOT NULL or NOT NULL and UNIQUE, and it can be specified to contain a default value.

datatype Specifies one of the following datatypes:

- CHARACTER or CHAR for fixed-length character string values
- VARCHAR for variable-length character string values.
- DATE, TIME, or TIMESTAMP for datetime values
- DECIMAL or DEC for signed decimal values (same as NUMERIC)
- FLOAT for signed floating-point values (same as DOUBLE PRECISION)
- REAL for signed floating-point values
- INTEGER or INT for signed integer values (between -2^{31} and $2^{31}-1$)
- SMALLINT for small signed integer values (between -2^{15} and $2^{15}-1$)
- TINYINT for signed integer values (between -2^{7} and $2^{7}-1$)

When you define a column of character (CHAR or VARCHAR) datatype, the value you specify for the length of the column is interpreted as the number of *bytes* that a column value can occupy, not the number of characters.

When a 7- or 8-bit ASCII character set is used, a character is always one byte long, so it makes no difference whether the length of the column is interpreted in bytes or characters. However, users of multibyte character sets must account for the possibility that a column might not be able to hold as many characters as it would if a single-byte character set were being used.

For additional information about datatypes, refer to page 2-18.

NOT NULL Declares that each row of the table must contain a value in the column: No missing or unknown values are allowed. If an INSERT or UPDATE statement violates this constraint for a given row, the operation is not performed and an error message is returned to the user.

UNIQUE Declares that duplicate values are not allowed in the column. Uniqueness is enforced only if a single-column B-TREE index is defined on the column.

DEFAULT

Assigns a default value to the column. The default value for a column is used for existing rows when a new column is added to a table with an ALTER TABLE statement and for new rows when the row to be added does not contain a value for the column. This value is used when rows are added with the INSERT command and when rows are automatically generated to satisfy referential integrity during a load operation. For information about automatic row generation, refer to the *Table Management Utility Reference Guide*.

If a literal value is specified, it must be compatible with the datatype for the column.

If a function is specified, the value returned from the function must be compatible with the datatype for the column. The returned value is inserted into the column during row insertion. One of the following functions can be specified as the default:

- CURRENT_USER (or USER)

  The CURRENT_USER function can be assigned as a default only to columns with a datatype of CHAR(128) or VARCHAR(128) or greater.
- CURRENT_DATE
- CURRENT_TIME (*precision*)
- CURRENT_TIMESTAMP (*precision*)

DEFAULT NULL can be specified to set the default of the column to NULL if no value is specified during row insertion.

You can specify both NOT NULL and DEFAULT NULL for the same column. This combination specifies that the column cannot accept a default value under any circumstances. Defining a column in this way effectively disables the use of automatic row generation for that table. For information about automatic row generation, refer to the *Table Management Utility Reference Guide*.

For more information about the use of default values, refer to examples of the INSERT statement starting on page 8-210

WITH FILLFACTOR n	This clause applies only to columns of the VARCHAR datatype. Specifies the percentage of the column length that a typical VARCHAR value is expected to take. This value is used to calculate the size of a row and the number of rows to allocate to a block. If WITH FILLFACTOR is not specified, the default column fill factor in the *rbw.config* file is used; if no column fill factor is defined there, the system default of 10 percent is used. For more information about fill factors, refer to the *Administrator's Guide*.

The CREATE TABLE statement below creates the table Prod_Basic. The default value *Unknown* is assigned to the Prodname column. When a row is later inserted into the Prod_Basic table, *Unknown* will be inserted into the Prodname column if no value is specified.

```
create table prod_basic (
    prodkey integer not null,
    prodname char(30) default 'Unknown',
    descript character(40),
    constraint prod_pkc primary key (prodkey) )
```

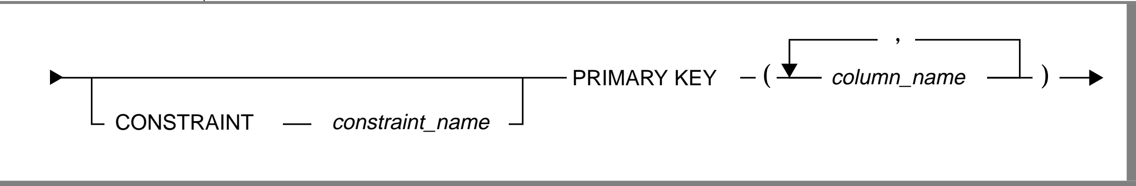
Primary-Key Reference

A table's primary key can consist of one or more columns. Primary keys are optional; however, it is recommended that only temporary tables be created without them. A table that does not have a primary key cannot be:

- Foreign key–referenced by another table.
- Checked for referential integrity.
- Loaded with the Table Management Utility UPDATE or MODIFY syntax.

Syntax

The following syntax diagram shows how to construct a *primary_key_reference*. To see how the PRIMARY KEY REFERENCES clause relates to the CREATE TABLE statement and column definitions, refer to page 8-135.



CONSTRAINT constraint_name

Specifies a name for the primary key constraint, which is then stored in the RBW_RELATIONSHIPS and RBW_CONSTRAINTS system tables. Primary key constraint names are optional. If they are not supplied, default names of the following format are automatically assigned:

```
tablename_PKEY_CONSTRAINT
```

For detailed information about constraint names, see "Primary-Key and Foreign-Key Constraint Names" on page 8-144.

PRIMARY KEY column_name

Specifies the primary key of the table. Any table can have a multi-column primary key.

Primary keys must conform to the following rules:

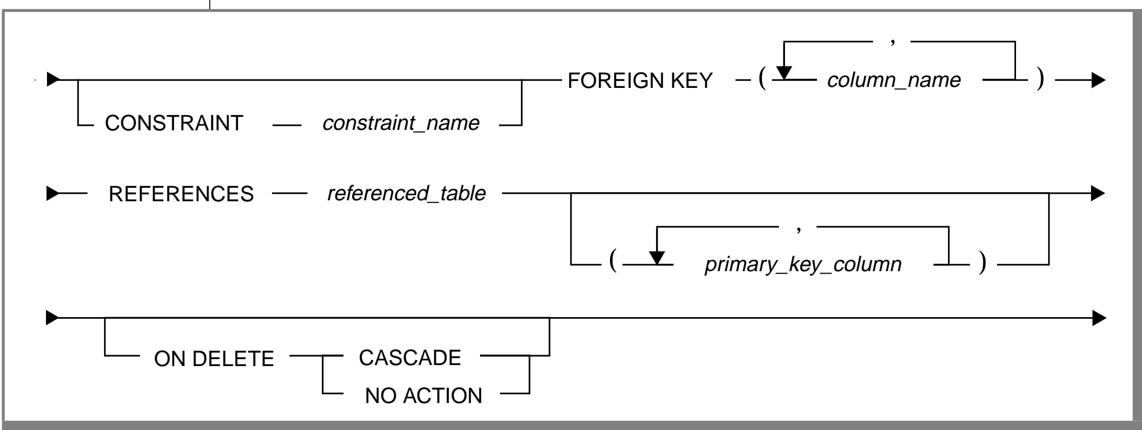
- A table must have only one primary key, but that key can consist of multiple columns.
- The column or columns declared as the primary key must each be declared NOT NULL.
- The values in a primary key must be unique: a primary key uniquely identifies a row of a table.

Foreign-Key Reference

When a column contains values selected only from the primary key values of another table, the column is said to make foreign key references.

Syntax

The following syntax diagram shows how to construct a *foreign_key_reference*. To see how the FOREIGN KEY REFERENCES clause relates to the CREATE TABLE statement and column definitions, refer to page 8-135.



CONSTRAINT constraint_name
: Specifies a name for each foreign key constraint, which is then stored in the RBW_RELATIONSHIPS and RBW_CONSTRAINTS system tables. Foreign key constraint names are optional. If they are not supplied, default names of the following format are automatically assigned:

```
tablename_FKEY#_CONSTRAINT
```

where # represents the ordinal position of the foreign key definition in the CREATE TABLE statement.

When a STAR index is created on a table that references a table with a multi-column primary key, the CREATE STAR INDEX statement must specify user-defined foreign key constraint names.

For detailed information about constraint names, see "Primary-Key and Foreign-Key Constraint Names" on page 8-144.

FOREIGN KEY
: Defines join paths between the table being created and the table named after the REFERENCES keyword. The paths are defined on the columns specified by *column_name* and *parent_column*. A maximum of 256 foreign keys can be declared for a given table.

Tables referenced by foreign keys must have a primary key.

You cannot define circular schema references. For example, you cannot add a constraint from *table1* to *table2* if there is already a constraint from *table2* to *table1*. This kind of table definition would produce a referential integrity cycle.

column_name
: Specifies the name of a column in the table being created. This column must be declared NOT NULL and can have the same name as *parent_column*.

referenced_table
: Specifies the name of a referenced table that has already been created.

primary_key_ column	Specifies the name of a primary key column in the *referenced_table*. If you do not specify the columns, the primary key of the referenced table is automatically used. If specified, the list must match the primary-key columns specified in the referenced table.
ON DELETE	Specifies how referential integrity is to be maintained when a delete operation occurs on the referenced table: either by deleting all referencing rows or by not deleting the referenced row. If this clause is omitted, the default is NO ACTION, and no row will be deleted if its deletion would cause a referential integrity violation. If NO ACTION is specified, neither the row in the referenced table nor the referencing row is deleted. This type of delete is referred to as a restricted delete. If CASCADE is specified, the row in the referenced table and all rows that reference that row will be deleted. After such a delete operation, a message is issued containing the number of rows deleted from each table. This type of delete is referred to as a cascaded delete. If a delete operation is mixed-mode (that is, if it involves both restricted and cascaded deletes), NO ACTION takes priority over CASCADE: No rows will be deleted. This priority is recursive through all the referencing tables. A synonym must have the same referential actions as its base table. If not, the CREATE TABLE statement generates an error message. The behavior specified by the ON DELETE clause can be overridden for a specific DELETE statement if that statement includes the OVERRIDE REFCHECK clause. The OVERRIDE REFCHECK clause should be used with extreme caution to avoid violating referential integrity. For examples illustrating the delete operation modes, see "DELETE" on page 8-166 and the *Informix Red Brick Decision Server Administrator's Guide*.

Primary-Key and Foreign-Key Constraint Names

In the CREATE TABLE statement, primary and foreign key references can be prefixed with *constraint names.*

A primary key constraint name is a means of uniquely identifying the primary key column or columns of the table being created. A foreign key constraint name is a means of uniquely identifying each foreign key reference that the table makes to some other column or columns in some other table.

For example, the CREATE TABLE statement for the Sales table specifies a primary key constraint and four foreign key constraints:

```
create table sales (
    perkey integer not null,
    classkey integer not null,
    prodkey integer not null,
    storekey integer not null,
    promokey integer not null,
    ...
    constraint sales_pkc primary key
        (perkey, classkey, prodkey, storekey, promokey),
    constraint sales_date_fkc foreign key (perkey)
        references period (perkey),
    constraint sales_product_fkc foreign key (classkey,
prodkey)
        references product (classkey, prodkey),
    constraint sales_store_fkc foreign key (storekey)
        references store (storekey),
    constraint sales_promo_fkc foreign key (promokey)
        references promotion (promokey))
    ...
```

When you create a table, constraint names are always optional. If you do not specify them, the system assigns default constraint names. Default constraint names are of the form

```
table_name_PKEY_CONSTRAINT
table_name_FKEY#_CONSTRAINT
```

where ***table_name*** is the name of the table being created and # represents the ordinal position of the foreign key definition in the CREATE TABLE statement. Whether system-defined or user-defined, constraint names are stored in the RBW_RELATIONSHIPS and RBW_CONSTRAINTS system tables.

Because the system tables track primary and foreign key constraints by constraint names rather than by column names, it is advisable to specify user-defined constraint names consistently in all your CREATE TABLE statements. User-defined constraints are also a mechanism for assigning meaningful names to primary and foreign key references.

Although constraint names are always optional in the CREATE TABLE syntax, they are sometimes required in CREATE STAR INDEX statements: When you create a STAR index on referencing table columns that form a multi-column primary key in the referenced table, you must specify the index key with a user-defined foreign key constraint name.

For example, the following statement attempts to create a STAR index on the Classkey and Prodkey columns of the Sales table:

```
create star index prod_sales_ix on sales (classkey, prodkey);
```

Because these columns form a multi-column primary key in the referenced Product table, the statement returns a syntax error. The only way to create this STAR index is by specifying the foreign key constraint name that identifies the two foreign key columns:

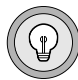
```
create star index prod_sales_ix on sales (sales_product_fkc);
```

Note that the *sales_product_fkc* constraint was defined in the CREATE TABLE statement for the Sales table. For more information, see page 8-144.

Segment Specification

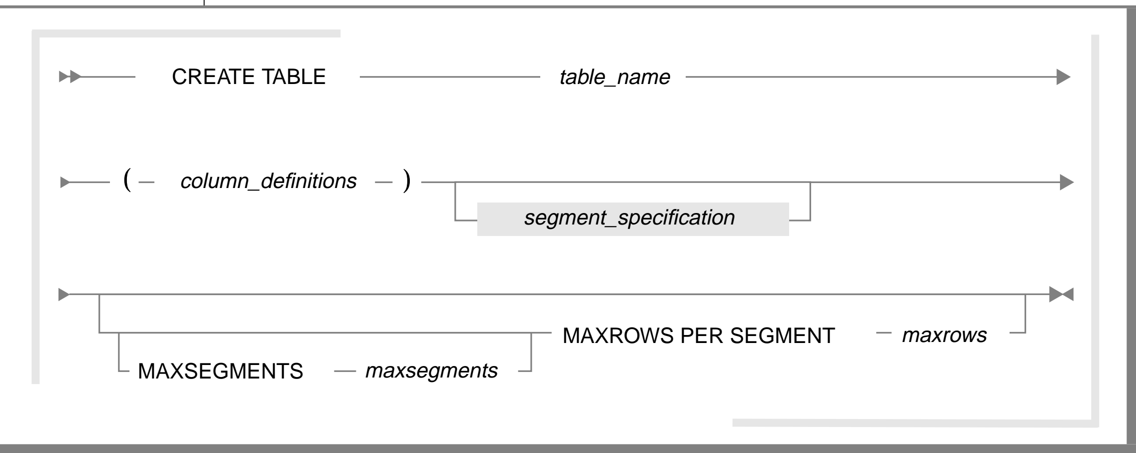
The segment specification assigns named segments and distribution ranges to table data and to the system-generated B-TREE index. For guidelines on when a segment should be specified for a database object, refer to the *Informix Red Brick Decision Server Administrator's Guide*.

You can divide all your tables into multiple segments. The data for each table that has a primary key resides, by default, in one segment, and its system-generated B-TREE index resides in another. The system assigns default segments if a segment is not specified for either the data or the index.

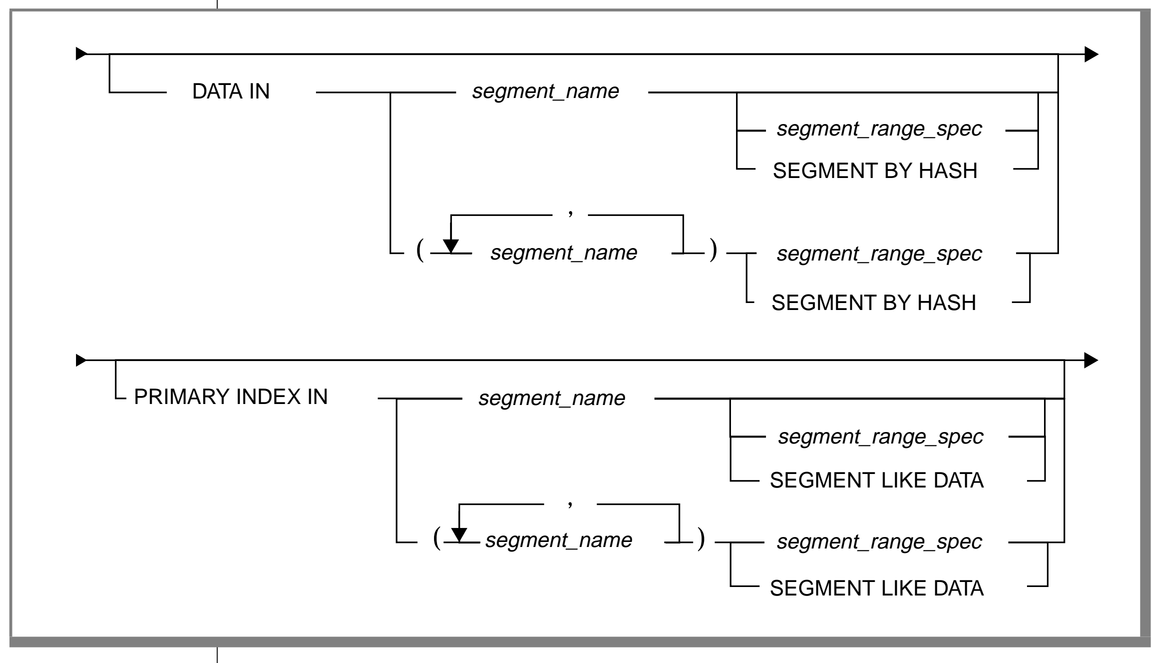


Tip: *If you are using Red Brick Decision Server for Workgroups, all tables and indexes are restricted to one segment each. Therefore, only one segment name can be specified for the data and the primary key index, and the segment range specification cannot be used.*

The entire CREATE TABLE statement syntax diagram is repeated to provide a point of reference for the ***segment_specification*** syntax:

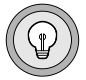


The following syntax diagram shows how to construct a segment specification.



DATA IN segment_name(s	Names one or more segments in which the data will reside. Any table can reside in multiple segments.

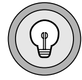
- If a single segment is named or created by default, the entire table resides in that segment.
- If multiple segments are specified, the data is distributed among them based on the segment range specification. The segment names must be separated by commas and the list enclosed in parentheses.
- If you do not specify a segment name, the system creates a default segment for the table, which consists of one physical storage unit (PSU).



Tip: *In Red Brick Decision Server for Workgroups databases, only one segment name can be specified.*

*PRIMARY INDEX IN segment_name**(s)***	Names one or more segments in which the primary key index will reside.

- If a single segment is named, the entire index resides in that segment.
- If multiple segments are named, the index is distributed among them based on the segment range specification. The segment names must be separated by commas and the list enclosed in parentheses.
- If no names are specified, the system creates a default segment for the primary key index, which consists of one PSU.



Tip: *In Red Brick Decision Server for Workgroups databases, only one segment name can be specified*

For information about creating indexes in named segments, see "CREATE INDEX" on page 8-96.

Examples—Using Default and Named Segments

The following statement creates a table that resides in default segments:

```
create table product(
    prodkey integer not null,
    classkey integer not null,
    product char(30),
    vendor character(40),
    constraint prod_pkc primary key (prodkey, classkey)
    constraint prod_fkc foreign key (classkey)
        references class (classkey))
```

The following statement creates a table that resides in named segments—the data in *dataseg* and the primary key index in *indexseg*.

```
create table market(
    mktkey integer not null,
    city char(40),
    state character(40),
    constraint mkt_pkc primary key (mktkey) )
    data in dataseg
    primary index in indexseg
```

segment_range_spec Assigns the segmentation scheme to the data or index. For the syntax of the segment range specification, refer to page 8-149.

SEGMENT BY HASH Distributes row data based on a hashing scheme that spreads data evenly among the segments. Rather than distributing data based on a value of a column, hashing uses all the values in each row to determine how the data is distributed. Hashing data results in random distribution and avoids the clustering of data in a single segment. This option cannot be used for indexes.

Segmenting with the hashing scheme requires no segmenting column or range specification.

A segment of a table created with the hashing scheme cannot be altered with the following options of the ALTER SEGMENT statement: DETACH, ATTACH, CHANGE RANGE, SEGMENT BY.

Example

In this example, the data will be distributed evenly among three segments.

```
data in (seg1, seg2, seg3)
    segment by hash
```

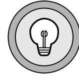
SEGMENT LIKE DATA Specifies that the segment range specification for a primary key index is identical to the segment range specification for the data. This option is valid only if:

- The data is segmented by values, not by hashing.
- The leading column of the primary key is the same as the segmenting column for the data.
- The same number of segments is specified for the primary key index and the data.

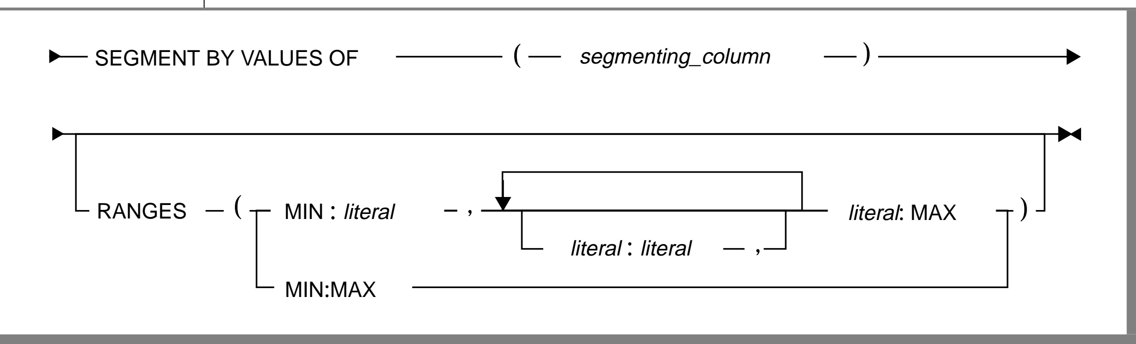
In Red Brick Decision Server for Workgroups databases, this clause cannot be used.

Segment Range Specification

The following syntax diagram shows how to construct a segment range specification. To see how the segment range specification relates to the segment specification, refer to page 8-145.



Tip: *In Red Brick Decision Server for Workgroups databases, the segment range specification cannot be used.*



SEGMENT BY VALUES OF segmenting_ column	Distributes table data or the primary key index among segments based on values in the *segmenting_column.* For data segmentation, the segmenting column must be declared NOT NULL. For primary key index segmentation, the segmenting column must be the first column of the primary key index.

Examples—Specifying a Segmenting Column

In this example, the Mktkey column is the segmenting column. The index is distributed among the segments *seg1_ix*, *seg2_ix*, and *seg3_ix* based on the values in the Mktkey column.

```
primary index in (seg1_ix, seg2_ix, seg3_ix)
    segment by values of (mktkey)
    ranges (...)
```

In this example, a segmenting column is specified even though only one segment is specified. Specifying a segmenting column for a single-segment table or index is useful if additional segments will later be attached with the ALTER SEGMENT command.

```
data in (seg1)
    segment by values of (prodkey)
    ranges (...)
```

RANGES (MIN...MAX) Specifies distribution of values in the segmenting column among the segments.

For both data and the primary key index, the ranges are based on values in the segmenting column. The range of values for the data or the primary key index depends upon the datatype of the segmenting column. If the segmenting column is an INTEGER, the range must be between –2,147,483,648 and 2,147,483,647. If the segmenting column is CHAR or VARCHAR, the ranges must be character values.

A range must be specified for each segment of a multi-segment data table or primary key index.

Each segment range must include a colon (:) and be separated by a comma. Ranges must start with the MIN keyword and end with the MAX keyword. They must be in ascending order and must not have any overlaps or gaps. The upper boundary of one segment must be the lower boundary of the next segment. The values entered into each segment are greater than or equal to the lower boundary and less than the upper boundary.

Examples

The following example illustrates a range for either data or a primary key index. The data (or index) is distributed among three segments based on values in the State column. Notice that the upper range (*AZ*) of the first segment is the lower range of the second segment. When data is inserted into the table or index, values from the lowest ASCII value to *AY* will be inserted into the first segment, values from *AZ* to *MN* into the second segment, and values from *MO* to the highest ASCII value into the third segment.

```
segment by values of (state)
ranges (min:'AZ', 'AZ':'MO', 'MO':max)
```

The next example specifies a range for either data or a primary key index. The data is distributed among four segments based on values of the Mktkey column. If a row containing a Mktkey value of 1000 is inserted into the table, it will be stored in the second segment.

```
segment by values of (mktkey)
ranges (min:1000, 1000:2000, 2000:3000, 3000:max)
```

The example in Figure 8-11 is incorrect because the first range overlaps the second range and a gap exists between the second and third:

```
/*Error*/

segment by values of (mktkey)
ranges (min:1000, 500:2000, 3000:4000, 4000:max)
                   ↑          ↑
```

Figure 8-11
Errors in Range Specifications

MIN, MAX Indicates the first and last values of a range. The first range value must be the MIN keyword, which indicates the minimum value in the segmenting column. The last range value must be the MAX keyword, which indicates the maximum value in the segmenting column.

Examples

The following example distributes either data or a primary key index among four segments. Mktkey is the segmenting column and is an INTEGER column. The MIN keyword specifies that all rows with Mktkey values below 500 will be stored in the first segment. The MAX keyword specifies that all rows with Mktkey values from 4,000 to 2,147,483,647 will be stored in the last segment.

```
segment by values of (mktkey)
ranges (min:500, 500:2000, 2000:4000, 4000:max)
```

literal Specifies an alphanumeric value that provides a range for a segment. For information about literal values, refer to page 2-10.

More CREATE TABLE Examples

The following statement creates a table that uses two default segments, one for the data and one for the primary key index.

```
create table sales  (
    mktkey integer not null,
    prodkey integer not null,
    perkey integer not null,
    punits numeric (9,2),
    units integer,
    dollars integer,
    constraint sales_pkc primary key (mktkey, prodkey,
        perkey),
    constraint sales_fkc1 foreign key (mktkey) references
        market (mktkey),
    constraint sales_fkc2 foreign key (prodkey) references
        product (prodkey),
    constraint sales_fkc3 foreign key (perkey) references
        period (perkey) )
```

The following statement creates a table that resides in named segments. The default value *No Name* is assigned to the Prodname column. When a row is later inserted into the Product_Exmpl table, *No Name* will be inserted into the Prodname column if no value is specified. If no value is specified for the Descript column, the column is set to NULL.

```
create table product_exmpl (
    prodkey integer not null,
    prodname char(30) default 'No Name',
    descript character(40),
    constraint prod_pkc primary key (prodkey) )
    data in dataseg
    primary index in indexseg
    maxsegments 2
    maxrows per segment 50000
```

The following statement creates a table whose data and primary key index both reside in multiple named segments.

```
create table orders (
    invoice integer not null,
    line_item integer not null,
    perkey date not null,
    prodkey integer not null,
    classkey integer not null,
    custkey integer not null,
    promokey integer not null,
    dollars integer,
    weight integer,
    constraint orders_pkc primary key (invoice, line_item),
```

```
constraint orders_fkc1 foreign key (perkey) references
    period (perkey),
constraint orders_fkc2 foreign key (prodkey, classkey)
    references product (prodkey, classkey),
constraint orders_fkc3 foreign key (custkey) references
    customer (custkey),
constraint orders_fkc4 foreign key (promokey) references
    promotion (promokey) )
data in (orders_data1, orders_data2, orders_data3)
    segment by values of (perkey)
    ranges (min:'04-01-1999',
        '04-01-1999':'07-01-1999',
        '07-01-1999':max)
primary index in (orders_ix1, orders_ix2, orders_ix3)
    segment by values of (invoice)
    ranges (min:1000, 1000:3000, 3000:max)
maxsegments 3
maxrows per segment 50000
```

CREATE TEMPORARY TABLE

A CREATE TEMPORARY TABLE statement defines a table that is accessible only during the session in which it was created. It exists only for the duration of that session or until it is dropped with a DROP TABLE command. A B-TREE index is automatically created on the table's primary key.

A temporary table has the following characteristics:

- It is exclusive to the SQL session in which it is created; it is not visible outside of the session and does not share its data with other sessions.
- It can be joined to any table in the database. Indexes and column default values can be defined for a temporary table and persist during the life of the table.
- It can have the same name as a permanent table created during a later session, but not the same name as a permanent table created during the same or a prior session.
- Any queries submitted by the user of a temporary table are automatically queried against that table. This is true even if a permanent table with the same name exists, or is subsequently created by a user in another session. Temporary tables always takes precedence over permanent tables.

- It does not have to be locked while it is being updated because other users cannot access the temporary table.
- To avoid system table contention, it is not permanently cataloged in the system tables. Information about a temporary table resides in memory and appears in the system tables during the user's session, but disappears when the session ends.
- It is automatically dropped at the end of the SQL session; however, it can also be dropped during the session with the DROP TABLE command.

Red Brick Decision Server temporary tables are consistent with the ANSI SQL-92 standard.

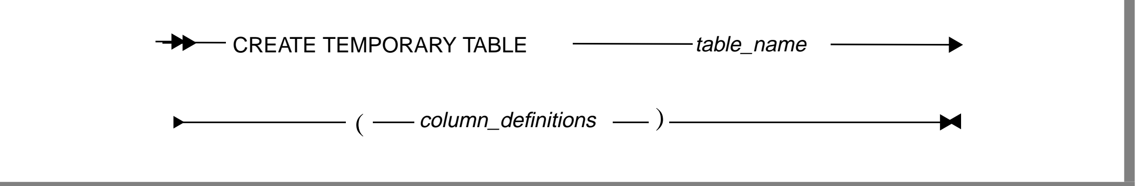
Authorization

To create a temporary table, a user must meet at least one of the following requirements:

- Be a member of the DBA or have RESOURCE system role authorization.
- Have CONNECT system role authorization through the GRANT_TEMP_RESOURCE_TO_ALL option.
- Have CREATE_ANY, CREATE_OWN, or TEMP_RESOURCE authorization either explicitly or through membership in a user-created role.

Syntax

The following syntax diagram shows how to construct a CREATE TEMPORARY TABLE statement:



table_name Specifies the name of the temporary table being created, which must be:

- Unique within each session. The table name must be different from that of temporary tables created during other sessions that access the same database.
- A valid database identifier

column_ definitions Specifies column definitions for each column in the table. The column_definitions within a CREATE TEMPORARY TABLE statement can contain a primary key definition but cannot contain foreign key references. Specification of MAXSEGMENTS or MAXROWS PER SEGMENT is not allowed. For a description of column definitions, refer to page 8-135.

Usage Notes

Note the following differences between a permanent table and a temporary table:

- A temporary table supports primary key definitions but does not support foreign key references.
- A temporary table does not support creation of a STAR index because it cannot reference other tables or be referenced by other tables.
- A temporary table does not support segment specification for data or index. They must each reside in a separate default segment.
- A temporary table does not allow specification of MAXSEGMENTS or MAXROWS PER SEGMENT.

- A temporary table cannot be loaded by the Table Management Utility (TMU).
- The following operations cannot be performed on a temporary table:
  - ALTER TABLE
  - ALTER INDEX
  - ALTER SEGMENT
  - CREATE VIEW
  - CREATE SYNONYM
  - CREATE STAR INDEX
  - GRANT PRIVILEGE
  - REVOKE PRIVILEGE

Example

The following example creates a temporary table:

```
create temporary table tea_list(
    name char(10) not null,
    type char(5) not null,
    stock_no int not null,
    constraint temptable_pkc primary key (col1,col3))
```

CREATE VIEW

The CREATE VIEW command creates a read-only table whose source data is a query expression that selects from existing tables or views.

The CREATE VIEW statement reads the query expression, expands any referenced macros, and then stores the expression in a more efficient "operational" format. Consequently, an existing view does not reflect any subsequent changes to the macros or tables it references. The original text is stored only for displaying in the RBW_VIEWTEXT table.

If a table or macro is modified after the view is created, the view must be dropped and re-created before it reflects changes to that table or macro reference.

A CREATE VIEW statement that contains a USING clause creates a precomputed view. A precomputed view is a view associated with a base table that contains the results of the query defined in the view. (The results of regular views are not precomputed in this way.) For more information about precomputed views, refer to the *Informix Vista User's Guide*.

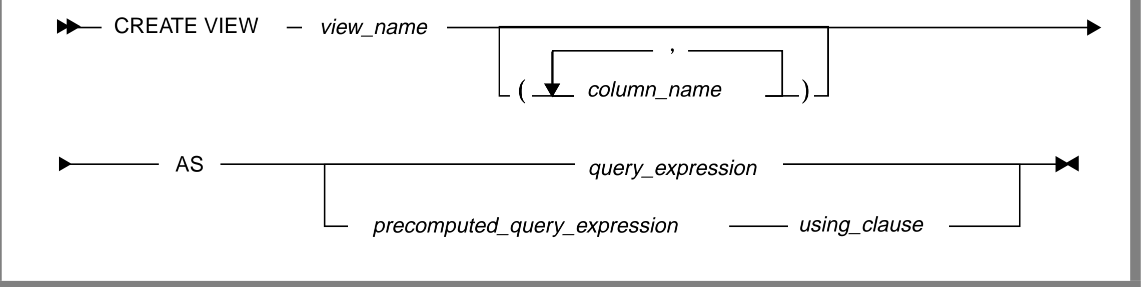
Authorization

To create a view, a user must meet at least one of the following requirements:

- Be a member of the DBA or RESOURCE system role.
- Have CREATE_ANY or CREATE_OWN authorization, either explicitly or through membership in a user-created role.

Syntax

The following syntax diagram shows how to construct a CREATE VIEW statement:



view_name Specifies the name of the view. The view name must be a database identifier that is different from the name given to any other view or table in the database. Views cannot be modified with INSERT, UPDATE, or DELETE statements.

column_name Specifies the column names of the view. These can be the same as or different from the column names of the base table. If a list of column names is specified, it must contain the same number of columns as the specified query expression would return if it were expressed as a SELECT statement.

If a list of column names is not specified, the view contains the same column names as the base table. A list of column names must be specified if the query expression:

- References tables that share column names; otherwise, the duplicate column names would result in ambiguous references.

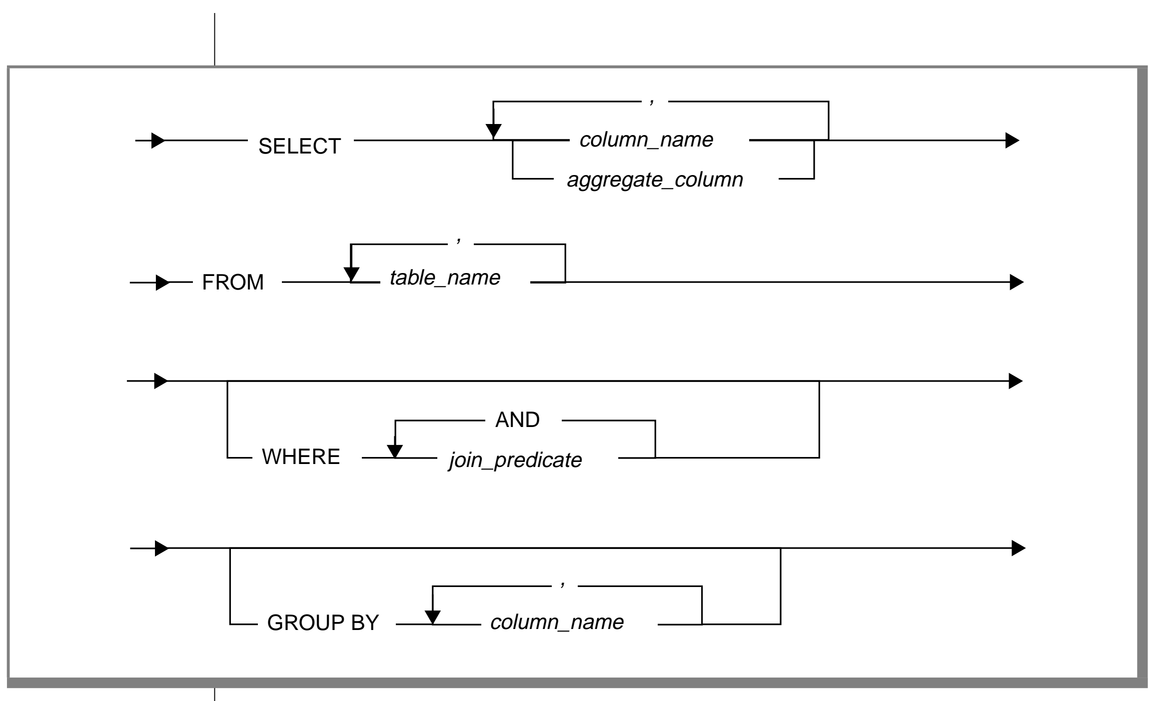
  For example, suppose the Sales table contains a Storekey column that also occurs in the Store table. If the query expression in the CREATE VIEW statement references the Sales and Store tables but does not include a list of column names, Storekey would be ambiguous.
- Includes unnamed columns. For example, if the query expression includes a set function such as

  ```
  sum(dollars)
  ```

  with no alias, the column is unnamed.

query_ expression Defines the contents of the view. Whenever the view is referenced, the result table that would be returned by a SELECT statement equivalent to the query expression is returned. Query expressions are defined and discussed in detail in Chapter 7, “Query Expressions.”

precomputed_ query_ expression Specifies a query expression that is more limited in scope than the query expression in a regular CREATE VIEW statement. It cannot contain a subquery, a HAVING clause, or a WHEN clause. The syntax for a precomputed query expression is as follows:



SELECT Introduces the columns chosen from the tables specified in the FROM clause.

column_name Specifies a column selected from a table. Each column name specified in the SELECT list (other than aggregate column names) must also be specified in the GROUP BY clause. Column aliases are allowed in the SELECT list.

aggregate_column Specifies a column of the form

```
set_function(expression)
```

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

- SUM
- MIN
- MAX
- COUNT

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

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

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

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

Note the following restrictions:

- Expressions used as arguments to the COUNT function must be simple expressions.
- Expressions that contain scalar functions, RISQL display functions, and subqueries are not supported.
- The expression for the SUM function must be numeric.
- The AVG set function cannot be used; however, AVG queries can be rewritten if the appropriate aggregate table contains SUM and COUNT values for the same column.

- The DISTINCT function can only be used as an argument to the COUNT function. SELECT DISTINCT queries cannot be used.
- For detailed information about set functions, refer to Chapter 4, “Set Functions”; for more detailed information about expressions, refer to Chapter 3, “Expressions and Conditions.”

FROM Introduces the tables from which the columns in the SELECT list are derived. Each table in the FROM clause can be referenced by only one other table in this clause using join predicates in the WHERE clause. Joins cannot be specified in the FROM clause; they must be specified in the WHERE clause.

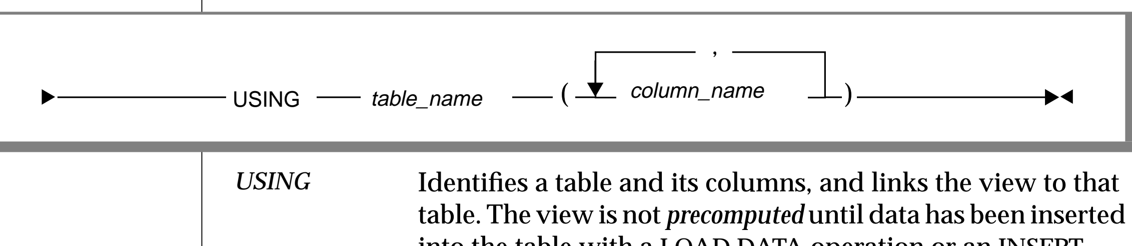
table_name Specifies a detail table or a synonym. System tables, views, tables derived from query expressions, and aggregate tables cannot be used. Table correlation names are allowed.

WHERE Contains the join predicates that join fact tables to dimension tables and outboard tables. In a query expression for a precomputed view, join predicates must join tables along foreign- key or primary-key relationships. These predicates must express equality conditions.

Outer join predicates are not allowed in precomputed view definitions.

GROUP BY Introduces the column names defined in the SELECT statement. All non-aggregated columns in the SELECT statement must be specified in the GROUP BY clause.

using_clause The following syntax diagram shows how to construct a using clause. To see how the using clause relates to the CREATE VIEW statement, refer to page 8-157.



USING Identifies a table and its columns, and links the view to that table. The view is not *precomputed* until data has been inserted into the table with a LOAD DATA operation or an INSERT statement.

table_name Specifies the name of the table associated with the view. Each precomputed view you create must use a different table.

column_name Specifies a column in the table that is mapped one-to-one with a column in the view. The list of column names is required.

Usage Notes for Precomputed Views

A CREATE VIEW...USING statement must meet the following validation constraints:

- The select list must contain at least one grouping column or one aggregation column.
- The grouping columns in the select list must match the columns listed in the GROUP BY clause.
- The join predicates must reflect primary key/foreign key relationships and equality constraints.
- The number of columns in the view must be the same as the number of columns specified for the table in the USING clause.
- The datatypes of the columns named in the table must be compatible with the datatypes of the columns in the view. Non-numeric datatypes must match exactly. For example, a CHAR(3) column in the table cannot map to a CHAR(4) column in the view.

  Numeric datatypes do not have to match exactly; however, if the column in the table might not be capable of storing the column values defined by the view, a warning message is displayed when you create the view. Refer to page 8-165 for an example.

Examples

The following view defines a logical table that contains only products that are classified as tea products (bulk or packaged):

```
create view tea_list
    as select product.classkey as c, product.prodkey as p,
        prod_name as name
    from product join class on product.classkey =
        class.classkey
    where class.classkey in (2, 5);
select * from tea_list;
C           P           NAME
2           0           Darjeeling Number 1
2           1           Darjeeling Special
2           10          Assam Grade A
2           11          Assam Gold Blend
2           12          Earl Grey
2           20          English Breakfast
2           21          Irish Breakfast
2           22          Special Tips
2           30          Gold Tips
2           31          Breakfast Blend
5           0           Darjeeling Number 1
5           1           Darjeeling Special
5           10          Assam Grade A
5           11          Assam Gold Blend
5           12          Earl Grey
5           20          English Breakfast
5           21          Irish Breakfast
5           22          Special Tips
5           30          Gold Tips
5           31          Breakfast Blend
```

The following view groups all stores by region and counts the number of stores per region:

```
create view regions (region, store_count)
    as select region, count(name)
    from market join store on market.mktkey = store.mktkey
    group by region;
select * from regions;
REGION                  STORE_COUNT
West                              7
South                             4
Central                           4
North                             4
```

The following statement creates a precomputed view associated with an aggregate table.

```
create view company_sales (period, class_no, product_no,
        units, amount) as
    select perkey, classkey, prodkey, sum (quantity) as units,
        sum (dollars) as amount
    from sales
    group by perkey, prodkey, classkey
using aggr_sales_table (perkey, classkey, prodkey, quantity,
    dollars)
```

The following statement creates a precomputed view associated with a derived dimension table; therefore, the query expression contains only grouping columns (no aggregation column). For information about derived dimensions, refer to the *Informix Vista User's Guide*.

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

The following example demonstrates the need for compatible datatypes in precomputed views and their associated tables. When a numeric column in the table is mapped to a numeric column in the view, a warning message is displayed if the two datatypes do not match exactly:

```
create table agg1 (perkey int, dollars dec(7,2))
create view agg1_view(perkey, dollars)
    as select period.perkey, sum(dollars)
    from sales, period
    where period.perkey = sales.perkey
    group by period.perkey
using agg1(perkey, dollars)
** WARNING ** (1931) Datatypes of related columns in the view
(AGG1_VIEW.DOLLARS, datatype DECIMAL (13,2)) and table
(AGG1.DOLLARS, datatype DECIMAL (7,2)) might cause precision
loss, overflow, or underflow.
```

This message is only a warning; the table and view are both created successfully.

A complete example that shows how to create and select from a view is presented in the *SQL Self-Study Guide*. Several more examples of precomputed views are presented in the *Informix Vista User's Guide*.

DELETE

A DELETE command deletes one or more rows from a specified base table.

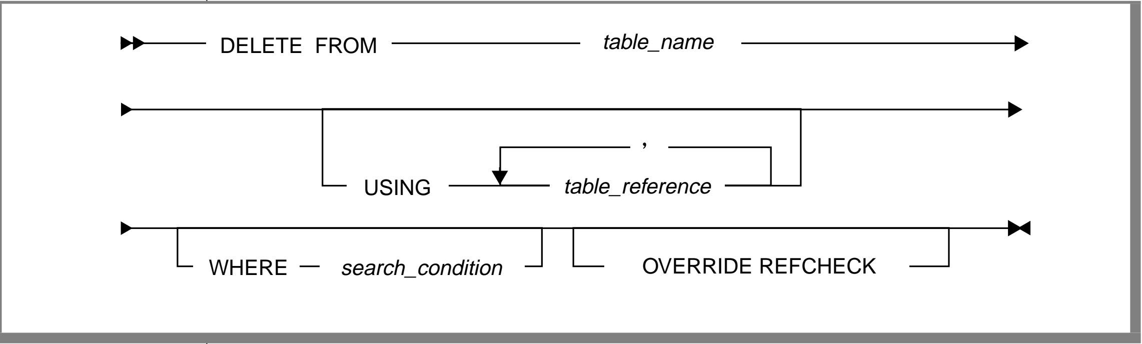
Authorization

To delete rows from a table, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have MODIFY_ANY authorization, either explicitly or through membership in a user-created role.
- Be the creator of the table.
- Have DELETE privilege on the table.

Syntax

The following syntax diagram shows how to construct a DELETE statement:



table_name Specifies the name of a table, a temporary table, or a synonym whose rows are to be deleted: *table_name* cannot be the name of a view.

USING table_reference
Specifies one or more tables to use as additional criteria for deleting rows. The tables specified by *table-reference* are joined, and then rows are deleted from *table-name* as specified by the WHERE clause. *table-reference* must also include *table-name* (see the example at the end of this section).

The USING *table-reference* clause is an alternative to using a subquery that provides improved performance.

The table or tables specified by *table-reference* must be base tables or synonyms. A USING clause is not allowed when updating a model table.

If a correlation name is assigned to the table from which rows are being deleted, that correlation name, not the actual table name, must be used to identify the table.

The *table_reference* clause is described in detail in "Table References" on page 7-13.

WHERE
Sets a search condition on which the DELETE statement acts. If a WHERE clause is not specified, the statement deletes all rows from the table; otherwise, it deletes only those rows that satisfy the search condition. The search condition can reference only the specified table. For additional information about conditions, refer to page 3-10; for more information about the WHERE clause, refer to page 7-26.

OVERRIDE REFCHECK
Directs the server not to validate referential integrity during the execution of the DELETE statement. The primary reason for overriding referential integrity checks is the case where several rows are being deleted from a table, and it is known that there are no rows that reference the rows to be deleted.

This clause overrides the delete mode set by the CREATE TABLE statement. Although omitting the referential integrity checks might result in a significant speed improvement, the database might be left in an inconsistent state and require a reorganization to ensure referential integrity once again.

The use of OVERRIDE REFCHECK can potentially result in incorrect query results and referential integrity violations.

Usage Notes

Referential integrity is maintained during delete operations by performing either a restricted delete (NO ACTION) or a cascaded delete (CASCADE), depending on the delete mode(s) specified when a table was created. If a delete operation involves multiple referencing tables and these tables were created with different delete modes, then the entire delete operation is performed as a restricted delete to ensure referential integrity. The priority of NO ACTION over CASCADE is recursive throughout the referenced and referencing tables.

When multiple DELETE statements are to be issued, access is improved if other users are restricted from access to the tables with LOCK table statements.

For information about LOCK table statements, refer to "LOCK Table" on page 8-212. For additional information about database reorganization and reloading and cascaded and restricted deletes, refer to the *Informix Red Brick Decision Server Administrator's Guide*.

Unlike the SELECT statement where outerjoins are supported but strongly discouraged, the DELETE statement does not support outer joins in the WHERE clause. If an outer join is required, it must be specified in the USING clause.

In the USING clause, the table being modified cannot be specified:

- as the left table specified in a right outer join
- as the right table specified in a left outer join
- anywhere in a full outer join

If a join operation is performed in a DELETE statement, the selection criteria specified in the statement might identify the same row in the target table more than once. If this occurs, the row is deleted the first time it is selected and all subsequent occurrences are ignored. If the row cannot be deleted because doing so would cause a referential integrity violation, each occurrence of the row is counted in the statistics of the number of rows not deleted that is reported at the end of the operation. In such a case, the sum of the number of rows not deleted and the number of rows deleted can exceed the number of rows in the table. A message is displayed describing this situation.

Examples: DELETE

The following DELETE statement removes any row from the Product table that satisfies the search condition specified in the WHERE clause:

```
delete from product
    where prod_name like '%Allspice%'
        and pkg_type like 'Sealed%'
```

The following example illustrates the USING clause. This example deletes all promotions that occurred before January 15, 1998.

```
delete from promotion
    using promotion, period, sales
    where period.perkey = sales.perkey
        and promotion.promokey = sales.promokey
        and period.date < DATE '1998-01-15'
```

The following example illustrates how delete operations maintain referential integrity based on the ON DELETE option in the CREATE TABLE statements for the referencing tables. Note that the tables Fact1 and Fact2 both reference table Dim1 with similar ON DELETE clauses, but they reference table Dim2 with different ON DELETE clauses.

```
create table dim1(
    pkey1 int not null,
    primary key (pkey1)
)

create table dim2(
    pkey2 int not null,
    primary key (pkey2)
)

create table fact1(
    pkey1 int not null,
    pkey2 int not null,
    primary key (pkey1, pkey2),
    foreign key (pkey1) references dim1(pkey1)
        on delete cascade,
    foreign key (pkey2) references dim2(pkey2)
        on delete cascade
)

create table fact2(
    pkey1 int not null,
    pkey2 int not null,
```

```
        primary key (pkey1, pkey2),
        foreign key (pkey1) references dim1(pkey1)
            on delete cascade,
        foreign key (pkey2) references dim2(pkey2)
            on delete no action
    )
```

In a delete operation on Dim1, the operation deletes the specified row in Dim1, plus all rows in Fact1 and Fact2 that reference the deleted row.

In a delete operation on Dim2, the operation is performed as if the ON DELETE clauses for the Dim2 references were both NO ACTION: No row is deleted from Dim2 that is referenced by Fact1 or Fact2, despite the cascaded delete mode specified for the Dim2 reference in Fact1.

The following example illustrates the priority of NO ACTION over CASCADE and the recursive nature of this priority:

```
    create table dim1_1(
        pkey1_1 int not null,
        primary key(pkey1_1)
    )

    create table dim1(
        pkey1 int not null,
        primary key (pkey1),
        fkey1 int not null,
        foreign key (fkey1) references dim1_1(pkey1_1)
            on delete cascade
    )

    create table dim2(
        pkey2 int not null,
        primary key (pkey2)
    )

    create table fact1(
        pkey1 int not null,
        pkey2 int not null,
        primary key (pkey1, pkey2),
        foreign key (pkey1) references dim1(pkey1)
            on delete cascade,
        foreign key (pkey2) references dim2(pkey2)
            on delete no action
    )
    create table fact2(
        pkey1 int not null,
        pkey2 int not null,
```

```
        primary key (pkey1, pkey2),
        foreign key (pkey1) references dim1(pkey1)
            on delete no action,
        foreign key (pkey2) references dim2(pkey2)
            on delete cascade
    )
```

In a delete operation on Dim1_1, the delete cascades into Dim1. However, because delete operations into Fact2 are NO ACTION, the entire delete operation is performed as if NO ACTION had been specified for all tables. That is, no row is deleted from Dim1_1 that is referenced from a row in Dim1 if the Dim1 row is in turn referenced by a row in Fact2. Furthermore, within the same delete operation, no row is deleted from Dim1_1 that is referenced from a row in Dim1 if the Dim1 row is in turn referenced by a row in Fact1. This is because restricted and cascaded deletes combined in one DELETE statement are all performed as restricted delete operations.

Any restricted (NO ACTION) delete in the primary key–foreign key dependencies implies restricted deletes for all dependencies.

DROP HIERARCHY

The DROP HIERARCHY command removes an existing hierarchy and all functional dependencies defined under that name.

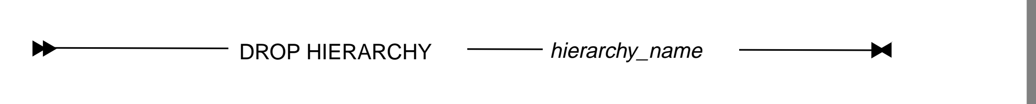
Authorization

To drop a hierarchy, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have DROP_ANY authorization, either explicitly or through membership in a user-created role.
- Be a member of the RESOURCE system role and be the creator of the hierarchy.

Syntax

The following syntax diagram shows how to construct a DROP HIERARCHY statement:



Example

The following DROP HIERARCHY statement drops the hierarchy named store_market_ relationship. Note that when this hierarchy is dropped, all functional dependencies within the hierarchy are removed.

```
drop hierarchy store_market_relationship;
```

For more information about defining and dropping hierarchies, refer to the *Informix Vista User's Guide*.

DROP INDEX

The DROP INDEX command deletes the specified index. This command can delete any index created with the CREATE INDEX command, as well as system-generated B-TREE indexes.

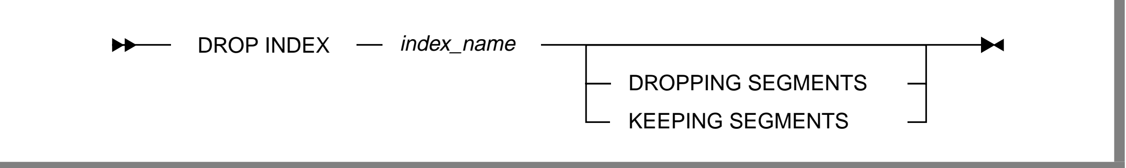
Authorization

To drop an index defined on a table, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have DROP_ANY authorization, either explicitly or through membership in a user-created role.
- Be a member of the RESOURCE system role and be the creator of the index.
- Have DROP_OWN authorization and be the creator of the index.

Syntax

The following syntax diagram shows how to construct a DROP INDEX statement:



index_name Specifies the name of the index to be dropped. You can drop any index created with the CREATE INDEX command, or any automatically created B-TREE index; see page 8-174.

DROPPING SEGMENTS Removes all named and default segments that are associated with the index. All physical storage units (PSUs) within the segments are deleted.

KEEPING SEGMENTS Retains all named segments that are associated with the index. The segments are detached from the index and are available for attachment to another database object. Default segments are always dropped.

Usage Notes

If neither DROPPING SEGMENTS nor KEEPING SEGMENTS is specified, the default behavior is used as specified in the OPTION SEGMENTS parameter of the *rbw.config* file. The default behavior originally specified in the *rbw.config* file is to keep named segments and drop default segments. The default behavior can be changed in the *rbw.config* file.

When you drop a table, all of its indexes are dropped automatically.

Examples

The following statement removes the *distribution_ix* index from the database and retains all of its named segments. The segments will be available for attachment to another index or table.

```
drop index distribution_ix
keeping segments
```

The next statement removes both the TARGET index *tgt_ix1* and its segment from the database. The segment will not be available for reuse.

```
drop index tgt_ix1
dropping segments
```

Dropping System-Generated B-TREE Indexes

When you use the DROP INDEX command to drop the system-generated B-TREE index, you will no longer be able to:

- Insert rows into the table on which the index was created.
- Insert rows into any other table that references the primary key of the table on which the index was created.

These restrictions on inserting data are imposed in order to preserve the uniqueness and referential integrity constraints of the data in the table. To make insert operations possible after dropping a system-generated index, you must explicitly create a new primary key index on the table.

Red Brick Decision Server does not automatically generate *simple star schemas*, whereby a STAR index is built on all of a table's foreign keys and those foreign keys constitute its primary key. To build such a schema, you must drop the table's system-generated B-TREE index, then issue a CREATE STAR INDEX statement that specifies all the foreign keys as the index key.

If you do create a simple star, inserts into the indexed table *are* allowed, despite the lack of a B-TREE index on the primary key. However, while the STAR index can maintain the uniqueness constraint, it cannot maintain referential integrity as well. Therefore, you will not be able to insert rows into any other table that references the primary key of the STAR-indexed table unless the referenced table has a B-TREE index on its primary key.

To summarize, rows cannot be inserted into:

- A *referenced* table whose primary key B-TREE index has been dropped. However, if a STAR index has been created on the primary key, rows can be inserted.
- A *referencing* table that references a table whose primary key B-TREE index has been dropped. Even if a STAR index has been created on the primary key of the referenced table, rows still cannot be inserted.

Order of Key Columns in Re-Created Indexes

When you create a B-TREE or STAR index on a multi-column primary key to replace a dropped system-generated B-TREE index, the order of the key columns in the new index is determined by the order specified in the CREATE INDEX statement. (For the system-generated index, the order is determined by the order listed in the CREATE TABLE statement.)

The order of the key columns in these "re-created" indexes is critical to performance. For information about indexing for performance, refer to the *Administrator's Guide*.

For detailed information about creating indexes, see "CREATE INDEX" on page 8-96.

DROP MACRO

The DROP MACRO command removes a macro name.

Authorization

To drop a PUBLIC macro, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have DROP_ANY or PUBLIC_MACROS authorization, either explicitly or through membership in a user-created role.

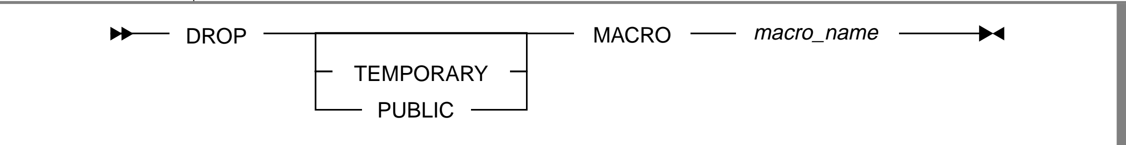
To drop a private macro, a user must be the creator of the macro and meet at least one of the following requirements:

- Be a member of the DBA or RESOURCE system role.
- Have DROP_ANY or DROP_OWN authorization, either explicitly or through membership in a user-created role.

Any user currently connected to a Red Brick Decision Server database can drop a TEMPORARY macro.

Syntax

The following syntax diagram shows how to construct a DROP MACRO statement:



Usage Note

When a view is created, any macros it references are expanded and the view SELECT statement is stored in a more efficient "operational" format. As a consequence, an existing view does not reflect any subsequent changes to the macros it references. If a macro is modified or dropped after the view is created, the view must be dropped and re-created before it reflects changes to that macro reference.

The keywords in the DROP MACRO statement must match the keywords that were used in the corresponding CREATE MACRO statement. For example, to drop a temporary macro, the TEMPORARY keyword must be used in the DROP MACRO statement, even if no other macro (public or private) has the same name.

DROP ROLE

The DROP ROLE command deletes the specified role.

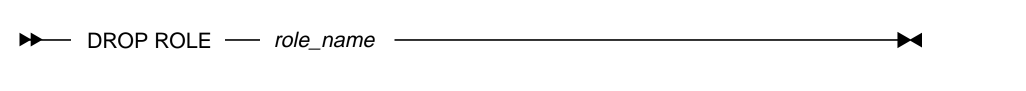
Authorization

To drop a role, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have ROLE_MANAGEMENT authorization either explicitly or through membership in a user-created role.

Syntax

The following syntax diagram shows how to construct a DROP ROLE statement:



role_name Specifies the role name to be dropped. System roles cannot be dropped.

Usage Notes

After a role is dropped, database users might still have some task authorizations or object privileges of the role if any of those authorizations or privileges were granted explicitly to the user or to a another role in which the user is a member.

Example

The following statement deletes the *temp* role:

```
drop role temp
```

DROP SEGMENT

The DROP SEGMENT command deletes the specified segment from the database as well as all physical storage units (PSUs) associated with it.

The DROP SEGMENT command cannot be used to drop the backup segment. Refer to "ALTER DATABASE" on page 8-6 for information about how to drop the backup segment.

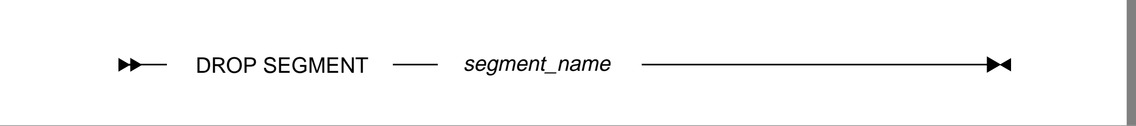
Authorization

To drop a segment, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have DROP_ANY authorization, either explicitly or through membership in a user-created role.
- Be a member of the RESOURCE system role and be the creator of the segment.
- Have DROP_OWN authorization and be the creator of the segment.

Syntax

The following syntax diagram shows how to construct a DROP SEGMENT statement:



Usage Note

Dropping a segment deletes all files in the segment. Before a segment can be dropped with the DROP SEGMENT command, it must be set to OFFLINE mode and detached.

A segment can be set to OFFLINE mode with the ALTER SEGMENT command. A segment can be detached:

- With the ALTER SEGMENT command.
- When a table or index is dropped, but the segment is not removed.

Examples

The following series of statements takes the *seg_market* segment offline, detaches it, and deletes it:

```
alter segment seg_market of table market offline
alter segment seg_market of table market detach
drop segment seg_market
```

The following statement deletes the *seg_market_idx* segment. This example assumes that the index residing in the segment was dropped, but the segment was not deleted. Therefore, the *seg_market_idx* segment is already detached.

```
drop segment seg_market_idx
```

DROP SYNONYM

The DROP SYNONYM command deletes the specified synonym for a base table; the table is not dropped.

Authorization

To drop a synonym defined on a table, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have DROP_ANY authorization, either explicitly or through membership in a user-created role.
- Be a member of the RESOURCE system role and be the creator of the synonym.
- Have DROP_OWN authorization and be the creator of the synonym.

Syntax

The following syntax diagram shows how to construct a DROP SYNONYM statement:

synonym_name Specifies the name of the synonym to be dropped. A synonym cannot be dropped if it is referenced by a view.

Usage Note

Dropping a synonym defined on a table has no effect on the base table.

DROP TABLE

The DROP TABLE command deletes the specified table from the database, deletes any indexes defined on the table, and removes any privileges or synonyms that reference the table.

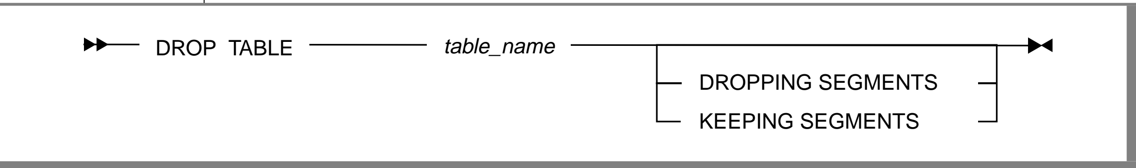
Authorization

To drop a table, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have DROP_ANY authorization, either explicitly or through membership in a user-created role.
- Be a member of the RESOURCE system role and be the creator of the table.
- Have DROP_OWN authorization and be the creator of the table.

Syntax

The following syntax diagram shows how to construct a DROP TABLE statement:



table_name The name of the table must be the name of a base table or a temporary table, not a synonym or a view. A table cannot be dropped if it is referenced by:

- A foreign key of another table
- A view

To drop a table referenced by a foreign key or by a view, drop the table or view containing the references first. (Alternatively, you can use the ALTER TABLE...DROP CONSTRAINT command to drop foreign key references, as described on page 8-79.)

DROPPING SEGMENTS Removes all default and named segments that are associated with the table. Segments associated with the table include all segments attached to the table and corresponding indexes. All physical storage units (PSUs) within the segments are deleted.

KEEPING SEGMENTS Retains all named segments that are associated with the table. The segments are detached from the table and are available to be assigned to another database object. Default segments are always dropped.

Usage Notes

If neither DROPPING SEGMENTS nor KEEPING SEGMENTS is specified, the default behavior is used as specified in the OPTION SEGMENTS parameter of the *rbw.config* file. If OPTION SEGMENTS is not specified, the default is to keep named segments and drop default segments. The default behavior can be changed in the *rbw.config* file.

Unlike user-defined segments, default segments are always dropped when the table is dropped.

If the table to be dropped contains one or more damaged segments (as indicated in the RBW_SEGMENTS system table), a DROP TABLE...KEEPING SEGMENTS command will fail; you must detach and drop the damaged segments before dropping the table. However, a DROP TABLE...DROPPING SEGMENTS command will succeed even when the table contains damaged segments.

Example

The following statement removes the Market_Temp table and its indexes from the database but retains all of its named segments. The segments will be available to attach to another table or to an index.

```
drop table market_temp
keeping segments
```

DROP VIEW

The DROP VIEW command removes the specified view from the database.

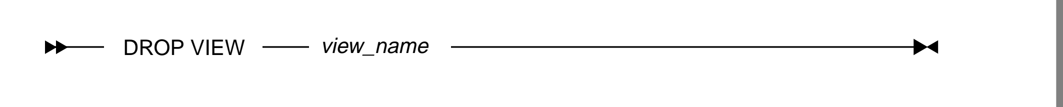
Authorization

To drop a view, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have DROP_ANY authorization, either explicitly or through membership in a user-created role.
- Be a member of the RESOURCE system role and be the creator of the view.
- Have DROP_OWN authorization and be the creator of the view.

Syntax

The following syntax diagram shows how to construct a DROP VIEW statement:



Usage Notes

Dropping a view from the database has no effect on its base table(s). Similarly, dropping a precomputed view associated with an aggregate table has no effect on the aggregate table.

A view referenced by other views cannot be dropped.

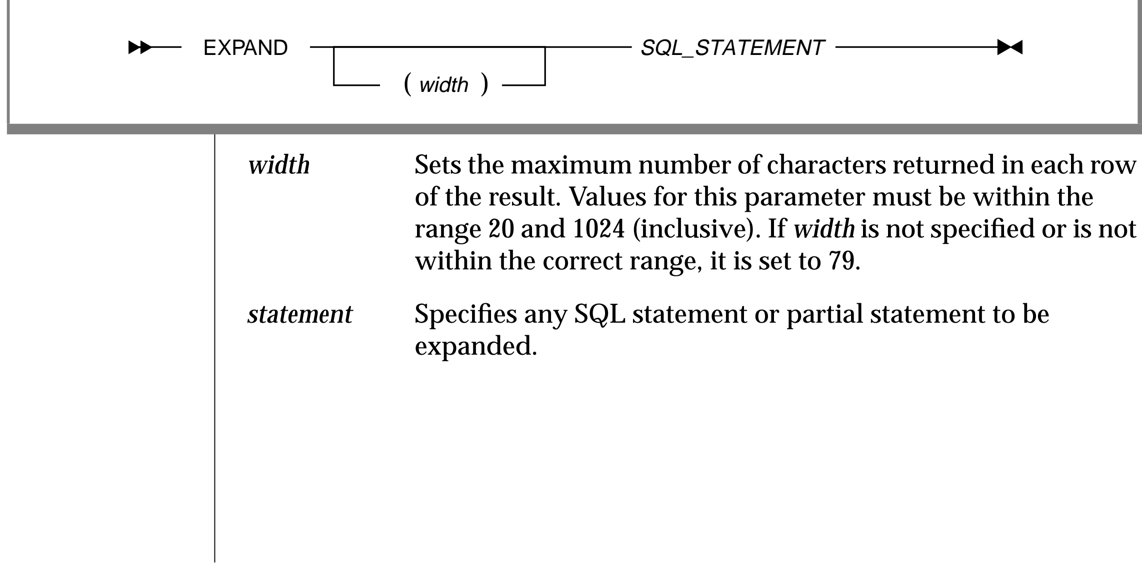
EXPAND

The EXPAND command displays a statement and expands any macro names contained in it.

When a statement is expanded, the server returns the statement with specific text substituted for all its macros (macro expansion). Although the syntax of the statement is not checked, its macros must be called with the correct number of arguments.

Syntax

The following syntax diagram shows how to construct an EXPAND statement:



width	Sets the maximum number of characters returned in each row of the result. Values for this parameter must be within the range 20 and 1024 (inclusive). If *width* is not specified or is not within the correct range, it is set to 79.
statement	Specifies any SQL statement or partial statement to be expanded.

Example

Define a single parameter macro as follows:

```
create temporary macro select_star(condition)
    as select *
    from market
    where condition
```

The following macro call retrieves rows from the Market table that have *South* in their Region column:

```
select_star(region='South')
```

The following EXPAND statement returns the SELECT statement after macro substitution has occurred:

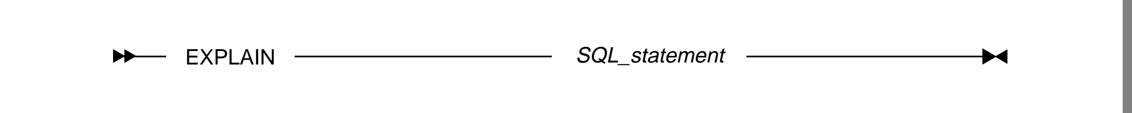
```
expand select_star(region='South')
STATEMENT
---------
SELECT * FROM MARKET WHERE REGION='South'
```

EXPLAIN

The EXPLAIN command displays internal query-processing information for a given query to help administrators tune performance. Unless otherwise specified, this information is displayed on the screen as standard output. Except for database object names, the output is always displayed in English. For details about the contents of the output, refer to the *Administrator's Guide*.

Syntax

The following syntax diagram shows how to construct an EXPLAIN statement:



SQL_statement Specifies the full text of the SQL statement. You can use the EXPLAIN command with any valid SQL query or INSERT, UPDATE, or DELETE statement.

Example

The following command will return information about all of the different operations involved in processing the query, including table joins, aggregations, and calculations.

```
explain
    select prod_name, sum(dollars) as prod_sales,
        rank(sum(dollars)) as prod_rank
    from product join sales on sales.classkey =
product.classkey
        and sales.prodkey = product.prodkey
    group by prod_name
```

If multiple processing plans are feasible for a query, the EXPLAIN command presents information about all the possibilities.

Usage Notes

This command is a useful tool for finding ways to improve query performance. For example, if a query takes advantage of a STAR index and uses STARjoin processing to join tables, it is likely to run faster. You can use the EXPLAIN command to determine what kind of join processing will be used for a query, then rewrite it to make better use of the indexes already available or add indexes to the schema.

Use the EXPLAIN command in conjunction with the SET STATS INFO command to compare the EXPLAIN output with the actual statistics and information generated when the query is executed. For information about SET STATS INFO, see "SET STATS" on page 9-52.

For examples of EXPLAIN output, refer to the *Administrator's Guide*.

EXPORT

The EXPORT command provides a means for data base administrators and application developers to efficiently export arbitrary result sets to a disk file. The export operation can move data from the database to an output file more efficiently than other methods because it bypasses the ODBC layer.

The user defines a result set with a query, specifies a format for the output file, and specifies an output file. The export statement then generates an output file plus a script that describes the output file (a TMU script) and a DDL script that describes a destination table. The user can also define a locale specification (optional). The generated DDL statement can be used to create a destination table, and the generated TMU script can be used to load the table with the exported data.

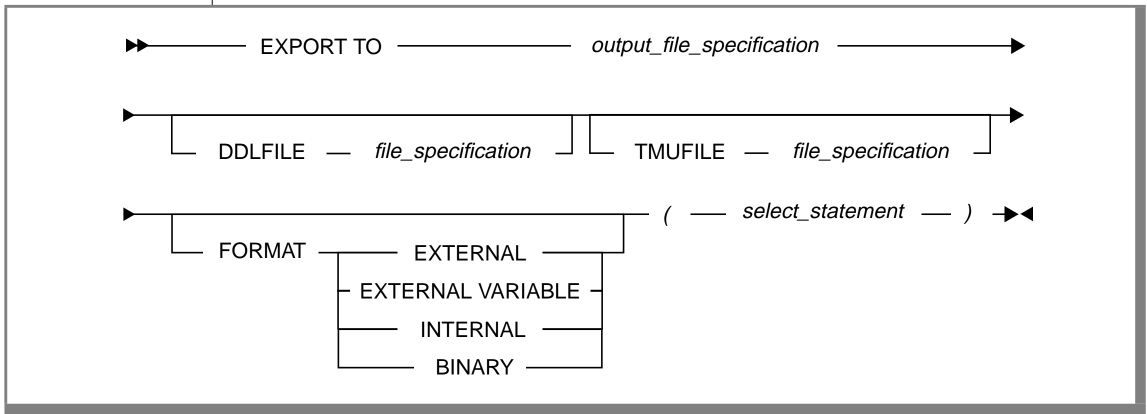
Authorization

To use the EXPORT command, the user must have either:

- DBA authorization
- EXPORT task authorization

Syntax

The following syntax diagram shows how to construct an EXPORT statement:



output_file_specification

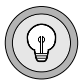
Specifies the name and path of the output file. The maximum length of a pathname is the maximum length allowed by the platform (generally 1000 bytes) minus 4 characters. This length permits the addition of .DDL or .TMU file extensions and allows creation of up to 999 overflow files. You must enclose the filename in single quotation marks.

The *output_file_specification* can be a fully qualified path and filename, a relative path and filename, or a pipe specification. If a file specification contains a relative path, it is relative to the path specified in *rbw.config* or by the path established by a SET EXPORT_DEFAULT_PATH command. If the first character of a file specification is a pipe symbol (|), the command writes the data to the pipe.

If the file specification contains a pipe symbol it *must* be the first character that occurs after the single quotation mark. If any other character, including a space, occurs before the pipe symbol, the command exports the data to a file instead of a pipe.

The specified path must exist and the *username* executing the command must have write permission to the file. The system creates the specified file for the export operation. For security reasons, this file can not be overwritten: if the filename already exists, an error message is returned and you must remove the existing file before you can execute the export statement.

If the EXPORT_MAX_FILE_SIZE option has been set, a three-digit suffix is appended to the name of each data file. You can create a maximum of 1000 such files.



Tip: By default, no limit is enforced for the size of the output file; overflow files are created only if the EXPORT_MAX_FILE_SIZE option is set. For more information about this option, see "SET EXPORT_MAX_FILE_SIZE" on page 9-16.

If the query returns no result set, the export operation does not generate output files.

DDLFILE file_specification

Specifies the name of the DDL file. This file is identical to the file created by the TMU UNLOAD or GENERATE command with one exception: the DDLFILE gives a unique column name to each column in the select-list column that defines the output result set.

If a *file_specification* is not supplied, the name generated is the name and path specified for the output file the suffix .DDL. If data is directed to a pipe, this specification is meaningless and references to it are ignored.

If a locale is specified, export operation creates the DDLFILE (and TMUFILE) in the same locale. Data translation is performed as needed.

The table in this file is named 'GENERATED_TABLE' and the columns are named after those used in the select list of the query except for cases where an expression or aggregate function occurs. In these cases, the columns have a generated name 'GENERATED_COLNAME_*XX*' where *XX* represents the column position in the select list.

TMUFILE file_specification

Specifies the name of the TMU file. The file is identical to the file generated by the TMU UNLOAD or GENERATE command with one exception: every select-list column in the output data is given a unique column name.

If the *file_specification* is not supplied, the generated name is the name and path specified for the output file with the suffix .TMU. If data is directed to a pipe, this specification is meaningless and references to it are ignored.

If a locale is specified, the export operation creates the TMUFILE (and DDLFILE) in the same locale. Data translation is performed as needed.

FORMAT	Specifies the format of the exported data. EXTERNAL format exports data to a file that can be reloaded with the TMU on the same or different platform. During the reload operation, trailing blanks are removed from the data. EXTERNAL VARIABLE format exports data to a file that can be reloaded using the TMU on the same or different platform. During the reload operation, trailing blanks are preserved and loaded into the destination table. INTERNAL format exports data to a binary file. This data can only be reloaded on a system of the same platform. For example, you cannot export data in this format from an HP 9000 and reload it on an AIX RISC System/6000. During the reload operation, trailing blanks are preserved and loaded into the destination table. BINARY format generates a file that has platform-native representations of integer, floating-point, and double precision datatypes and ASCII (character) representations of all others. This data is used to transfer data in a portable manner to other applications on the same platform. During the reload operation, trailing blanks are removed from the data. INTERNAL and EXTERNAL formats are identical to those supported by the TMU UNLOAD command. For more information about these formats, refer to the *Table Management Utility Reference Guide*.
select_ statement	Specifies any valid SELECT statement, as defined in Chapter 7. The statement must be enclosed with parentheses. The results of the query are exported to the output file in the specified format and translated, if necessary, to the client locale.

Usage Notes

The format of the output file can limit how and where the exported data can be reloaded.

- INTERNAL format can only be reloaded into an Informix Red Brick database that runs on the same kind of platform. Datatypes of the destination table must *exactly* match the datatypes of the exported columns. If a Dollars field in the exported data is DEC(7,2), then the destination column must be DEC(7,2), not DEC(7,4) or DEC(10,2). If you export INTERNAL from an HP 9000, then you cannot reload the data into a Red Brick database running on an AIX RISC/6000. Also,
- EXTERNAL format is more portable. When you need to migrate from one platform to another kind of platform, this is your choice. This format can also be used with third-party applications, although using NULL information with these applications can be more difficult.
- BINARY format is more compact than EXTERNAL and can be used with third-party applications but only on the same platform. This exported data also contains NULL markers.

Example

The following statement exports the data from a query to a file named */home/george/sum1*.

```
export to '/home/george/sum1'
    (select prod_name, sum(dollars),
    rank(sum(dollars)) as top_ten_99
    from sales natural join product
    natural join period
where year = 1999
group by prod_name when top_ten_99 <=10)
```

The generated file *home/george/sum1.DDL* contains the following information:

```
CREATE TABLE GENERATED_TABLE (
PROD_NAME CHARACTER(30),
GENERATED_COLNAME_2 DECIMAL(13,2),
TOP_TEN_99 INTEGER);
```

The generated file */home/george/sum1.TMU* contains the following information:

```
LOAD DATA INPUTFILE
'/home/george/sum1'
RECORDLEN 60
INSERT
NLS_LOCALE 'English_UnitedStates.US-ASCII@Binary'
INTO TABLE GENERATED_TABLE (
    PROD_NAME POSITION(2) CHARACTER(30) NULLIF(1)='%',
    GENERATED_COLNAME_2 POSITION(33) DECIMAL EXTERNAL(15)
    NULLIF(32)='%', TOP_TEN_99 POSITION(49) INTEGER
    EXTERNAL(11) NULLIF(48)='%');
```

GRANT Authorization and Role

The GRANT command can grant the DBA and RESOURCE system roles, user-created roles, or separate task authorizations to database users or roles.

A database user must be created and assigned a password with the GRANT CONNECT command before being granted an authorization or role.

Authorization

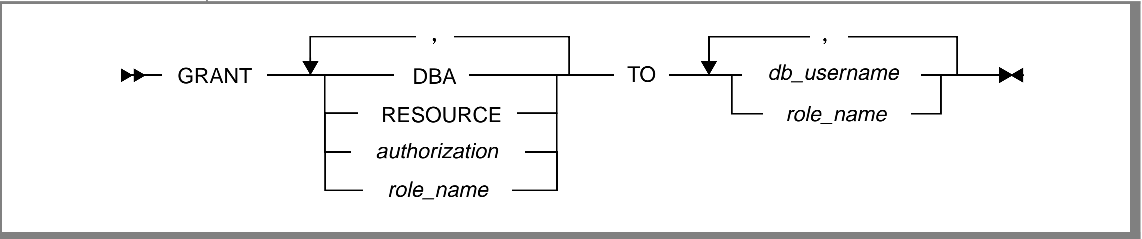
To use the GRANT command, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have the ROLE_MANAGEMENT authorization, either explicitly or through membership in a user-created role.

Be aware that the DBA and any user with the ROLE_MANAGEMENT task authorization can grant all task authorizations to themselves and others.

Syntax

The following syntax diagram shows how to construct a GRANT statement to grant authorizations and roles:



DBA Specifies the DBA system role is to be granted to the database users and user-created roles specified in the TO clause. These users and roles become members of the DBA system role and can perform all DBA tasks. For information about the DBA system role, see "System Roles and Task Authorizations" on page 8-194.

RESOURCE Specifies the RESOURCE system role is to be granted to the database users and user-created roles specified in the TO clause. These users and roles become members of the RESOURCE system role and can perform all RESOURCE tasks. For information about the RESOURCE system role, see "System Roles and Task Authorizations" on page 8-194.

authorization Specifies a task authorization to be granted to the database users and roles specified in the TO clause. Task authorizations are system-defined and provide the ability to perform operations within the database. For a list of task authorizations and their definitions, see "System Roles and Task Authorizations" on page 8-194.

role_name	Specifies a user-created role to be granted to the database users and roles specified in the TO clause. These users and roles receive all task authorizations defined for the granted role.
TO db_username	Specifies an existing database user to be granted the specified task authorizations and roles. Note that each database user can be a direct member of only 16 roles. However, there is no limit to the number of indirect role affiliations for each user.
TO role_name	Specifies a user-created role to be granted the specified task authorizations and roles. This role cannot be a system role (DBA or RESOURCE) because system roles cannot be altered.

System Roles and Task Authorizations

The system provides three system roles: CONNECT, RESOURCE, and DBA. The CONNECT system role allows a user to connect to the database and provides limited capabilities. The RESOURCE and DBA system roles allow a user to perform a set of tasks, as defined by the system.

The following table defines the task authorizations of the DBA system role. These task authorizations can be granted separately.

Task Authorization	Definition
ACCESS_ANY	Select data from all database objects and access private user information (such as private macros) in the system tables.
ACCESS_SYSINFO	Query the dynamic statistic tables for statistics about database activity. For information about the dynamic statistic tables, refer to the *Administrator's Guide*.
ALTER_ANY	Alter columns, indexes, macros, segments, synonyms, tables, and views.
ALTER_SYSTEM	Issue the ALTER SYSTEM command to perform database administration tasks.

(1 of 2)

Task Authorization	Definition
BACKUP_DATABASE	Back up the database. (*Informix Red Brick SQL*-BackTrack *Option*.)
CREATE_ANY	Create any object, including those that use another user's resources. For example, create an index on another user's table or create a table that reside in another user's segment.
DROP_ANY	Drop objects created by any user.
EXPORT	Export results of an arbitrary query to a data file.
GRANT_TABLE	Grant object privileges to database users and roles.
LOCK_DATABASE	Lock the database.
MODIFY_ANY	Insert, update, delete, and load any data.
OFFLINE_LOAD	Use any segment as a working segment for offline loads; synchronize segments after offline loads.
PUBLIC_MACROS	Create and drop PUBLIC macros.
REORG_ANY	Reorganize any table or index.
RESTORE_DATABASE	Restore the database. (*Informix Red Brick SQL*-BackTrack *Option*.)
ROLE_MANAGEMENT	Create, drop, grant, revoke, and alter roles.
UPGRADE_DATABASE	Upgrade the database.
USER_MANAGEMENT	Create database users with GRANT CONNECT. Drop database users with REVOKE CONNECT. Change passwords with GRANT CONNECT. Specify the default priority of a user's sessions with ALTER USER or GRANT CONNECT.

(2 of 2)

The following table defines the task authorizations of the RESOURCE system role. These task authorizations can be granted separately.

Task Authorization	Definition
ALTER_OWN	Alter own columns, indexes, macros, segments, synonyms, tables, and views.
ALTER_TABLE_INTO_ANY	Alter own tables into other users' segments.
CREATE_OWN	Create own objects (indexes, private macros, segments, synonyms, tables, and views).
DROP_OWN	Drop own objects.
GRANT_OWN	Grant object privileges on own objects to other users.

Usage Notes

- A database user becomes a direct member of a system role with a GRANT statement.
- System roles cannot be altered or dropped with GRANT and REVOKE commands.
- A database user becomes a direct member of a role with a GRANT statement.
- A database user can become a direct member of a role by being included in a CREATE ROLE statement.
- A database user or user-created role becomes an indirect member of a role when the user or role is a member of a one role that has been granted another role.

  For example, if Role1 is granted to Role2, members of Role2 are indirect members of Role1 and have all task authorizations and object privileges of Role1.

  A database user or user-created role can be an indirect member of an unlimited number of roles.
- Specific task authorizations, for example, CREATE_ANY, can be granted to database users and to user-created roles.

- A user or role can be a direct member of no more than 16 roles. If any user or role specified in the TO clause is already a member of 16 other roles, no user or role specified receives the grant.
- Database users can have access to task authorizations both directly and through role membership.
- User-defined roles, system roles, and task authorizations can be granted in a single GRANT statement.

Examples

The following statement grants the RESOURCE system role to *tommy*:

```
grant resource to tommy
```

The following statement grants the CREATE_OWN and ALTER_OWN task authorizations to *cody*, *daisy*, and the *temp* role. Users *cody* and *daisy* and all members of the *temp* role are now able to create and alter their own database objects.

```
grant create_own, alter_own to cody, daisy, temp
```

The following statement grants the *dba_junior* role to *sonia*:

```
grant dba_junior to sonia
```

The following statement grants the *temp* role to both *kathy* and the *dba_junior* role. The user *kathy* becomes a member of the *temp* role and has all task authorizations and object privileges that have been granted to that role. Members the of *dba_junior* role (*sonia*) are not direct members of the *temp* role, but indirectly have all task authorizations and object privileges of that role.

```
grant temp to kathy, dba_junior
```

To determine the roles that have been granted to each database user and role in the database, issue the following statement:

```
select username, rolename, indirect
from rbw_role_members
order by username, rolename
USERNAME     ROLENAME      INDI
----------   ----------    ----
DBA_JUNIOR   TEMP           N
KATHY        TEMP           N
SONIA        DBA_JUNIOR     N
SONIA        TEMP           Y
```

These results show that the *temp* role has been granted directly to the *dba_junior* role and to *kathy*. The *dba_junior* role has been granted directly to *sonia*. The *temp* role has been granted indirectly to *sonia* because she is a member of *dba_junior*, which has been granted the *temp* role.

The following statement grants the RESOURCE system role to the *management* role. The members of the *management* role become indirect members of the RESOURCE system role and have all task authorizations of the system role.

```
grant resource to management
```

GRANT CONNECT

The GRANT CONNECT command creates a database username, assigns or changes a password, and optionally grants the DBA or RESOURCE system role, user-created roles, or task authorizations to the user.

Authorization

To create a database name, assign a password, or assign a priority to a user, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have USER_MANAGEMENT authorization either explicitly or through membership in a user-created role.

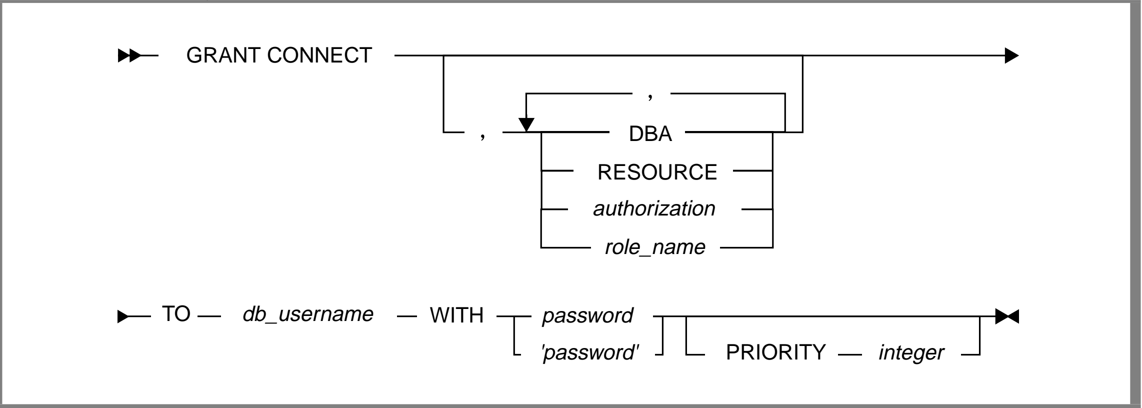
To grant a system role or a user-created role, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have ROLE_MANAGEMENT authorization and USER_MANAGEMENT authorization, either explicitly or through membership in a user-created role.

All database users can change their own passwords.

Syntax

The following syntax diagram shows how to construct a GRANT CONNECT statement:



CONNECT Either creates a new database username and assigns a password or changes the password of an existing database user. New users become members of the CONNECT system role and can:

- Connect to the database using their passwords.
- Retrieve data from any table that has been granted SELECT privilege to PUBLIC.
- Change their own passwords.
- Use PUBLIC tables and macros.

To access non-PUBLIC database objects, modify data, or create database objects, database users must be granted the appropriate object privilege, task authorization, or role.

For information about granting object privileges, see “GRANT Privilege” on page 202. For information about granting authorizations and roles, see “GRANT Authorization and Role” on page 8-192.

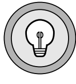
DBA	Specifies the DBA system role is granted to the specified database user. The database user becomes a member of the DBA system role and has all DBA task authorizations. For a list of DBA task authorizations, see “System Roles and Task Authorizations” on page 8-194.
RESOURCE	Specifies the RESOURCE system role is granted to the specified database user. The database user becomes a member of the RESOURCE system role and has all RESOURCE task authorizations. For a list of RESOURCE task authorizations, see “System Roles and Task Authorizations” on page 8-194.
authorization	Specifies the separate task authorization to be granted to the specified database user. Task authorizations are system-defined and provide the ability to perform operations within the database. For a list of task authorizations and their definitions, see “System Roles and Task Authorizations” on page 8-194.
role_name	Specifies a user-created role to which the specified database user will belong. The user becomes a member of the role and has all task authorizations and object privileges that have been granted to the role.
TO db_username	Specifies a database user. A database username must be a valid identifier and must be different from all other usernames.
WITH password	Specifies a database password. Database passwords must be different from all other passwords. If the password is not a valid SQL identifier, submit it as a string literal (in single quotes).

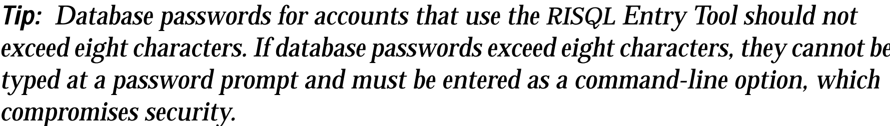
Users can change only their own passwords. If the username in a GRANT CONNECT statement is the same as the user executing the command, the user's password changes to the new value. Users with USER_MANAGEMENT task authorization, including those with the DBA system role, can change any user's password. Passwords are saved in encrypted form and are not accessible to any users, even those with the DBA system role.

Password security parameters are available with Red Brick Decision Server. These parameters are located in the *rbw.config* file and can restrict:

- The complexity and length of valid passwords.
- The frequency of password changes.
- Re-creation of old passwords to prevent users from repeatedly using the same password.

For more information about password security parameters, refer to the *Informix Red Brick Decision Server Administrator's Guide*.





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

PRIORITY integer	Specifies a default priority for the user's sessions; an integer in the range of 0 to 100, from highest to lowest. For example, if users have a priority of 0, their sessions take precedence for CPU time over users with a priority of 1 (or any number greater than 0). The default is 50. This clause does not affect current running sessions of an existing user. To change the priority of a running process, use the ALTER SYSTEM command. This clause is ignored if it is specified when users are changing their own passwords.

Examples

The following statement creates the database username *kathy* and assigns the password *dbexpert*:

```
grant connect to kathy with dbexpert
```

The following statement changes *kathy*'s password to *gumshoe*. Either *kathy* or a user with the DBA system role can execute this command.

```
grant connect to kathy with gumshoe
```

The following statement creates the database username *alison*, assigns the password *acrobat*, and grants the RESOURCE system role to *alison*:

```
grant connect, resource to alison with acrobat
```

GRANT Privilege

The GRANT command can assign object privileges on a specific table to one or more users or user-created roles.

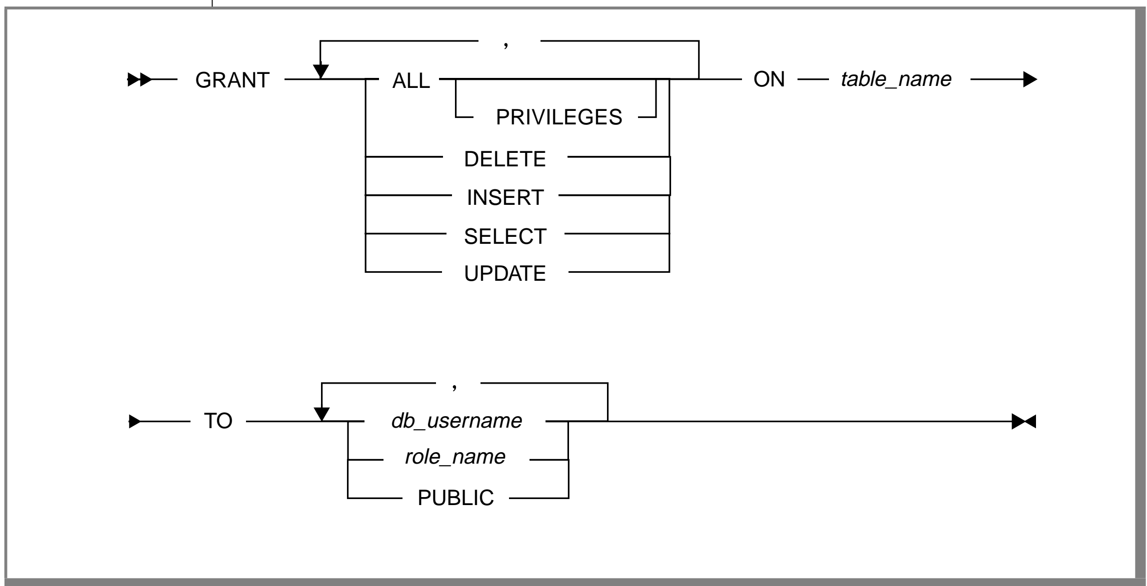
Authorization

To grant an object privilege on a table, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Be a member of the RESOURCE system role and be the creator of the table.
- Be the creator of the table and have the GRANT_OWN task authorization, either explicitly or through membership in a user-created role.
- Have the GRANT_TABLE task authorization, either explicitly or through membership in a user-created role.

Syntax

The following syntax diagram shows how to construct a GRANT privilege statement:



PRIVILEGES Object privileges can be granted to a specified user, a role, or to all users (to PUBLIC). A user must be granted CONNECT and assigned a password before being granted object privileges. A role must be created with the CREATE ROLE command before it can be granted object privileges.

System roles cannot be granted object privileges.

A user or role can be granted one or more of the following object privileges on a named table:

Object Privilege	Description
DELETE	Delete rows
INSERT	Insert rows
SELECT	Retrieve rows
UPDATE	Modify rows
ALL PRIVILEGES	All the above

Any user with INSERT privilege on a table must also have SELECT privilege on the table to insert rows.

A user who creates a table automatically has all object privileges on that table. These object privileges cannot be revoked from the table creator. A member of the DBA system role has all object privileges on any non-system table in the database.

table_name Specifies the table name. The named table must be in the database catalog or be a view; it cannot be a synonym for a table. If the named table is a view, only SELECT privilege can be granted. You cannot grant privileges on a temporary table.

TO db_username Grants all specified object privileges to a database user. A database username must exist before object privileges can be granted to it.

Grants all specified object privileges to a user-created role. All members of the role will have the object privilege. A role name must exist before object privileges can be granted to it. A role name cannot be a system role because system roles cannot be altered.

TO PUBLIC Grants all specified object privileges to all database users.

Examples

The following statement grants the SELECT privilege on the Product table to all database users.

```
grant select on product to public
```

The following statement grants SELECT, INSERT, DELETE, and UPDATE privileges on the Sales table to *alison*.

```
grant all privileges on sales to alison
```

The following statement grants the SELECT, INSERT, and DELETE privileges on the Market table to the *market_research* role. All members of the *market_research* role will be able perform these operations on the Market table.

```
grant select, insert, delete on market to market_research
```

INSERT

An INSERT command inserts one or more rows into a specified base table.

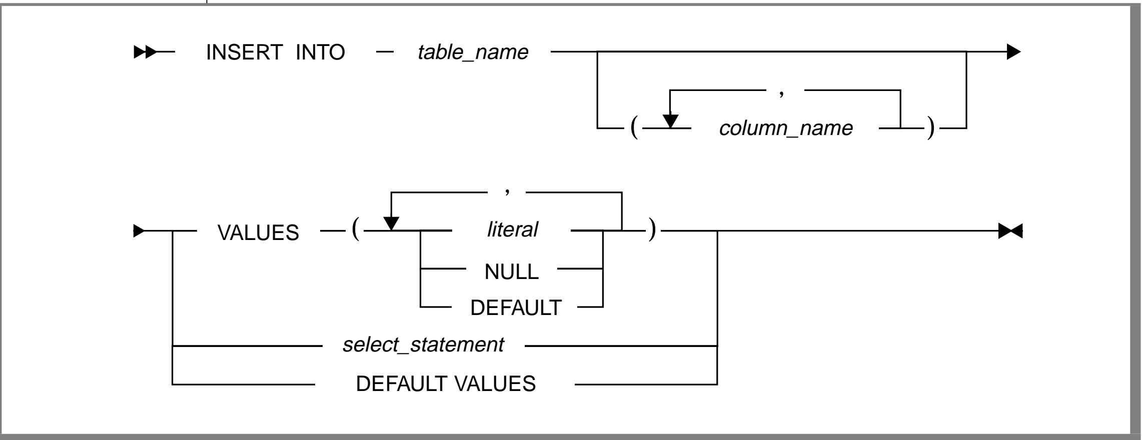
Authorization

To insert rows into a table, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have MODIFY_ANY authorization, either explicitly or through membership in a user-created role.
- Be the creator of the table.
- Have SELECT and INSERT privileges on the table.

Syntax

The following syntax diagram shows how to construct an INSERT statement:



table_name — Specifies the name or synonym of the table where rows are to be inserted. The *table_name* variable cannot name a view.

column_name — Specifies a column in the named table.

- The list of names can be in any order.
- A name can occur in the list only once.
- If a name is omitted, the column is assigned the default value (if a default value was not specified during table creation, the column is assigned NULL).

If no columns are specified, the command assumes that all columns have been specified and are ordered as they occur in the named table.

If a column is declared NOT NULL, then the INSERT statement must provide a non-null value for the column.

VALUES — If a VALUES clause is specified, it must contain the same number of entries as the column list. If a column list is not specified, the entries in the VALUES clause must correspond with the number of columns and order of columns in the named table.

The INSERT command inserts the first entry into the first column specified, the second value into the second column specified, and so on.

literal — A character or numeric constant.

NULL — Specifies the NULL value to be inserted into the column. If the column was defined as NOT NULL, the row is rejected.

DEFAULT Specifies that the default value of a column is inserted into the column. The default value is specified during creation of the table or when a column is added to the table with an ALTER TABLE command.

If the column was defined with no default and NULL values are allowed, NULL is inserted. If the column was defined with no default and as NOT NULL, the row is rejected.

select_statement Specifies a standard SQL SELECT statement, as defined on page 7-46, with the exception that the query cannot contain a BREAK BY subclause in its ORDER BY clause.

Sorting the result of a query that will be inserted into a table is useful with RISQL display functions, some of which are order-dependent. The ORDER BY clause controls the order of the intermediate query results, and, therefore, the value of columns containing RISQL display functions.

The result table of the query is inserted into the named table. The query can return one or more rows; if no rows are returned, the following message is returned:

```
** INFORMATION ** (209) Rows inserted: 0.
```

The number of columns returned by the query must be the same as the number of columns that occur in the column list. The value of the first column of the query's result table is inserted into the first column specified in the column list, the value of the second column is inserted into the second, and so on.

DEFAULT VALUES Specifies that a row inserted into the table use all default settings for the columns. A default value is inserted into each column of the table; you cannot use this DEFAULT VALUES with a ***column_name*** list.

An INSERT statement that contains the DEFAULT VALUES subclause is rejected if any column in the table has no default and is set to NOT NULL.

Usage Notes

In order to preserve referential integrity, an INSERT statement that contains a foreign key value and has no corresponding primary key value is rejected.

Rows cannot be inserted into:

- A *referenced* table whose primary key B-TREE index has been dropped. However, if a STAR index has been created on the primary key, rows can be inserted.
- A *referencing* table that references a table whose primary key B-TREE index has been dropped. Even if a STAR index has been created on the primary key of the referenced table, rows still cannot be inserted.

If the number of significant digits in a numeric constant exceeds the size of the numeric column, the server rejects the command and returns an error message. However, the server truncates the scale of a floating-point number whose precision exceeds the size of a numeric column.

Examples: INSERT command

This example shows how the results of a query can be inserted into a table. The results of a query that returns products sold in Los Gatos in the first quarter of 2000 is inserted into a table (Q1_00_Sales). The resulting sales data in Q1_00_Sales is a subset of the data in the Sales table. Queries can be issued on Q1_00_Sales to further constrain the data without repeating the City and Period constraints.

1. Create the Q1_00_Sales table.

```
create table Q1_00_Sales
    (product char(30), month char(5), dollars
dec(7,2));
```

2. Issue an INSERT statement that includes a query that returns sales data from Los Gatos in the first quarter of 2000. The results of the query are inserted into the Q1_00_Sales table.

```
insert into Q1_00_Sales (product, month, dollars)
    select prod_name, month, dollars
    from sales natural join product natural join period
        natural join store
    where qtr = 'Q1_00'
    and city like 'Los Gatos%';
** INFORMATION ** (209) Rows inserted: 390.
```

   A subset of the sales data is now stored in the Q1_00_Sales table. The table contains only sales data about products sold in Los Gatos in the first quarter of 2000.

3. Issue a query on the Q1_00_Sales table to further constrain the data to display information about the Veracruzano product:

```
select * from q1_00_sales
where product like 'Vera%';
PRODUCT                              MONTH  DOLLARS
Veracruzano                          JAN      330.00
Veracruzano                          JAN      285.00
Veracruzano                          JAN      262.50
...
```

A similar example of an INSERT INTO...SELECT statement is presented in the *SQL Self-Study Guide*.

The following INSERT statement defines a new class in the Class table by inserting one row:

```
insert into class
    (classkey, class_type, class_desc)
values (13, 'Music', 'Aroma collection of compact discs and
        cassettes')

CLASSKEYCLASS_TYPECLASS_DESC
13  MusicAroma collection of compact discs and
        cassettes
```

This statement can be re-written without listing the column names because it meets the following two conditions: A value is inserted for each column in the table and the values are inserted in the order in which the columns were defined in the table.

```
insert into class
values (13, 'Music', 'Aroma collection of compact discs and
    cassettes')
```

For the next example, assume that when the Store table was created, the default *CA* was set for the State column. The statement inserts *CA* into the State column.

```
insert into store
    (custkey, mktkey, store_type, name, street, city, state,
        zip)
values (20, 14, 'Small', 'Coffee Haven', '324 Ashby Avenue',
    'Berkeley', default, 94707)
```

For the next INSERT statement, assume that when the Market table was created, the default 1000 was set for Mktkey, the default *Atlanta* was set for Hq_City, and no default was set for the remaining columns, but NULL values are allowed in them:

```
insert into market
default values
MKTKEY  HQ_CITYHQ_STATEDISTRICTREGION
1000    AtlantaNULLNULLNULL
```

LOCK Table

The LOCK command blocks access by other users to a table.

Authorization

To lock a table, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have MODIFY_ANY authorization, either explicitly or through membership in a user-created role.
- Be the creator of the table.
- Have INSERT, DELETE, or UPDATE privilege on the table.

Syntax

The following syntax diagram shows how to construct a LOCK statement:

table_name Names the table to be locked. A user can set a lock on only one table at a time. The lock prevents access of any kind to the table by other users.

FOR DELETE Provides a cascading lock for use during delete operations. This option locks any table that contains a foreign key reference to the table named in the LOCK statement. The type of lock (read or write) on these other tables depends on the FOREIGN KEY... ON DELETE clause in their CREATE TABLE statements.

As each table is locked, this locking action also cascades to any tables that reference the locked table.

NO WAIT Specifies that a lock request fails if other users are accessing the table or have already locked the table. Control is immediately returned to the user (lock requestor). To lock the table, the lock requestor must re-issue the LOCK command when existing locks are released.

If the NO WAIT keywords are not specified, the server waits until existing locks are released and then locks the table with an exclusive lock. The server suspends the user until after the table has been locked. The length of the suspension depends on how many other users are accessing the table or whether another user already has the table locked. This (WAIT) behavior is the default behavior.

If waiting for existing locks to be released could result in a deadlock situation, the lock request is denied and control is returned immediately to the lock requestor.

NO WAIT and WAIT options can also be specified for the current session with a SET LOCK command. For information about the SET LOCK command, refer to page 9-32.

Usage Notes

When a user locks a table, other users cannot access the table until it is unlocked. A lock is released when the user that holds the lock submits an UNLOCK command or terminates the server session.

When you are performing multiple operations that modify a table or tables, you can use the LOCK command to improve access to those tables by locking out other users.

When a user locks a table and subsequently modifies the table or issues a CREATE INDEX statement, indexes are built while the lock is held.

Example

This example locks the Product table.

```
lock product no wait
```

The NO WAIT keywords are specified. If the Product table is locked or is being accessed by another user, the server denies the lock request and returns control to the lock requestor. If the Product table has not been locked or is not being accessed, the server locks the table with an exclusive lock.

LOCK DATABASE

The LOCK DATABASE command places a lock on each table in the database. The server suspends user access until each table in the database has been locked.

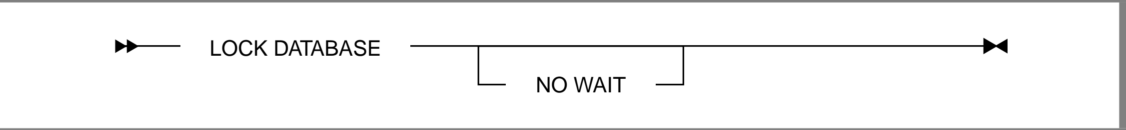
Authorization

To lock a database, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have LOCK_DATABASE authorization, either explicitly or through membership in a user-created role.

Syntax

The following syntax diagram shows how to construct a LOCK DATABASE statement:



NO WAIT Specifies that a database lock request always fails if at least one table is currently being accessed or is locked by another user. Control is immediately returned to the lock requestor, and the lock requestor must re-issue the LOCK command after existing locks are released.

If the NO WAIT keywords are not specified, the server waits until existing locks are released and then locks the database with an exclusive lock. If waiting for existing locks to be released could result in a deadlock situation, the lock request is denied and control is returned immediately to the lock requestor.

NO WAIT and WAIT options can also be specified for the current session with a SET LOCK command. For information about the SET LOCK command, refer to page 9-32.

Usage Notes

If the database is locked by a user (lock requestor) while another user (user2) is connected, user2 can quit from the database but can issue no other RISQL command until the database is unlocked.

A lock is released when the lock requestor:

- Submits an UNLOCK DATABASE command.
- Terminates the server session.

REVOKE Authorization and Role

The REVOKE command can remove the DBA and RESOURCE system roles, user-created roles, and separate task authorizations from database users and roles.

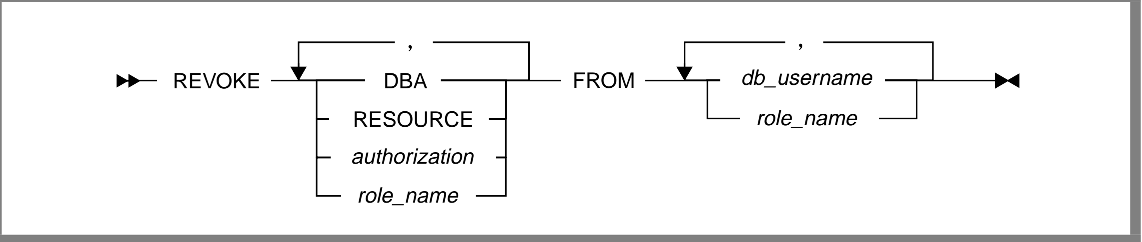
Authorization

To use the REVOKE command, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have ROLE_MANAGEMENT authorization, either explicitly or through membership in a user-created role.

Syntax

The following syntax diagram shows how to construct a REVOKE authorization statement:



DBA Specifies that the DBA system role is to be revoked from the specified database users and user-created roles.

RESOURCE Specifies that the RESOURCE system role is to be revoked from the specified database users and user-created roles.

authorization Specifies a separate task authorization to be revoked from the specified database users and roles.

role_name	Specifies a user-created role to be revoked from the specified database users and roles.
FROM db_username	Specifies a database username from whom the specified task authorizations and roles are to be revoked.
FROM role_name	Specifies a role name from which the specified task authorizations and roles are to be revoked. A system role cannot be specified because system roles cannot be altered.

Usage Note

Task authorizations cannot be revoked from system roles.

After revoking an authorization from a database user, the user might still have the authorization through a role. To remove an authorization from a database user, each role with that authorization must also be revoked from that database user.

Similarly, after revoking an authorization from a role, a member of the role might still have the authorization directly or through another role. To remove an authorization from all members of a role, revoke all occurrences of the authorization from each user.

Query the RBW_USERAUTH system table to determine if a user or role has an authorization explicitly, through a role, or through an indirect role.

Example

This example shows how an authorization can be revoked from a user, but the user still has access to the authorization.

Create the *temp* role:

```
create role temp
```

Create the user *tommy*, assign his password, and grant the *temp* role to him:

```
grant connect, temp to tommy with mysecret
```

Grant the CREATE_ANY task authorization to *tommy*:

```
grant create_any to tommy
```

Grant the CREATE_ANY task authorization to the *temp* role:

```
grant create_any to temp
```

Notice that *tommy* has been granted the CREATE_ANY task authorization explicitly and as a member of the *temp* role.

Revoke the CREATE_ANY task authorization from *tommy*:

```
revoke create_any from tommy
```

However, *tommy* can still create database objects because he is a member of temp. To prevent *tommy* from creating database objects, the CREATE_ANY task authorization must be revoked from the *temp* role or that role must be revoked from *tommy*:

```
revoke create_any from temp
revoke temp from tommy
```

REVOKE CONNECT

The REVOKE CONNECT command removes a username from the database.

Authorization

To drop a database username, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have USER_MANAGEMENT authorization, either explicitly or through membership in a user-created role.

Syntax

The following syntax diagram shows how to construct a REVOKE CONNECT statement:

```
▶▶── REVOKE CONNECT ── FROM ── db_username ──────────▶
```

CONNECT	Drops a database username from the database.
FROM db_username	Specifies a database username. That user can no longer connect to the database and has no task authorizations, object privileges, or roles associated with it.

Usage Note

To drop a role, use the DROP ROLE command.

REVOKE Privilege

A REVOKE command can remove specified object privileges on a named table from database users and user-created roles.

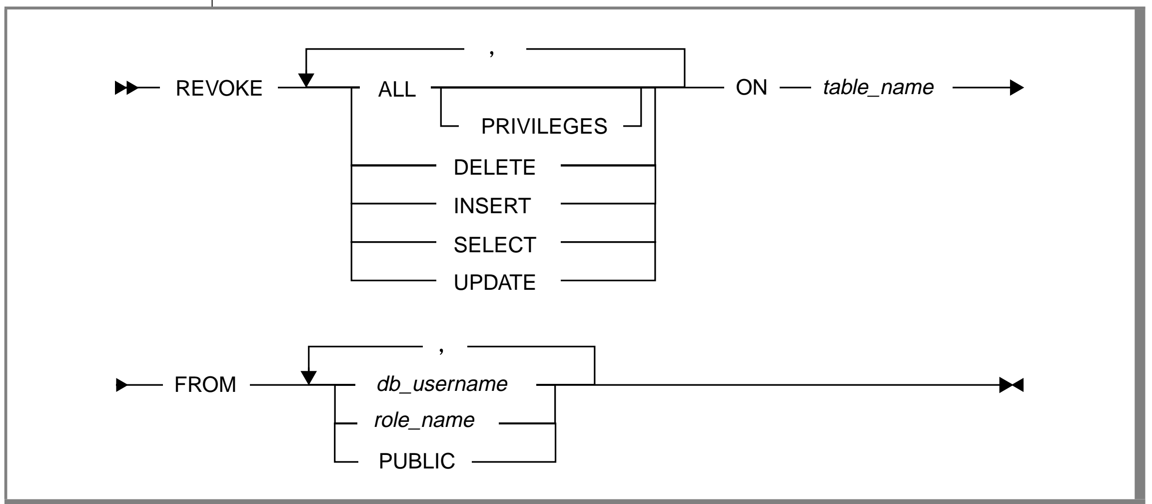
Authorization

To revoke an object privilege on a table, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Be a member of the RESOURCE system role and be the creator of the table.
- Be the creator of the table and have GRANT_OWN authorization, either explicitly or through membership in a user-created role.
- Have the GRANT_TABLE task authorization, either explicitly or through membership in a user-created role.

Syntax

The following syntax diagram shows how to construct a REVOKE privilege statement:



table_name	Specifies the table on which to revoke the specified object privileges. Privileges cannot be granted on temporary tables; therefore they cannot be revoked.
FROM db_username	Specifies a database user from whom to revoke all specified object privileges.
FROM role_name	Specifies a user-created role from which to revoke all specified object privileges.
PUBLIC	Specifies that all specified object privileges are to be revoked from all users.

Usage Note

After revoking an object privilege from a database user, the user might still have the object privilege through a user-created role. To remove an object privilege from a user, any roles with the object privilege must also be revoked from the user.

Similarly, after revoking an object privilege from a role, a member of the role might still have the object privilege directly or through another role. To remove an object privilege from all members of a role, revoke all occurrences of the object privilege from each user.

Query the RBW_TABAUTH system table to determine if a user or role has an object privilege for a given table explicitly, through a role, or through an indirect role.

Example

The following statement revokes the SELECT privilege on the Product table from all database users:

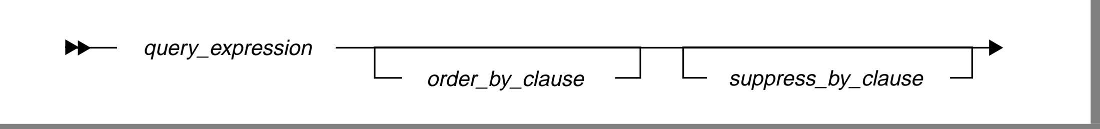
```
revoke select on product from public
```

Note that although the SELECT privilege has been revoked from all users (PUBLIC), a user can still select from the Product table if the user owns the table or has been granted any of the following:

- SELECT privilege on the table
- A role with SELECT privilege on the table
- ACCESS_ANY task authorization
- A role with ACCESS_ANY task authorization

SELECT

A SELECT statement retrieves rows of data from database tables. The following syntax diagram shows how to construct a SELECT statement:



query_expression Specifies any join or non-join query expression, as defined on page 7-3.

For detailed information about SELECT statements, refer to page 7-46.

UNLOCK Table

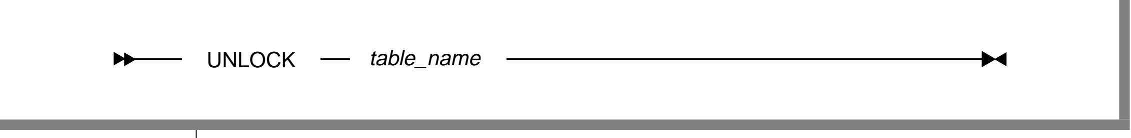
This command removes a lock previously set with a LOCK table command on a specified table.

Authorization

Only the user that holds the lock can unlock a table.

Syntax

The following syntax diagram shows how to construct an UNLOCK statement:



table_name Names the table to be unlocked.

Usage Notes

When a user locks a table and subsequently modifies the table or issues a CREATE INDEX statement, indexes are built while the lock is in place.

Example

The following example removes a lock on the Product table:

```
unlock product
```

UNLOCK DATABASE

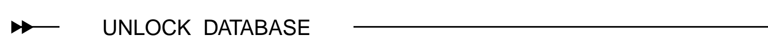
Removes a lock previously set by the user on the database.

Authorization

Only the user who set the database lock can unlock a database.

Syntax

The following syntax diagram shows how to construct an UNLOCK DATABASE statement:



Usage Notes

If a user locks the database, other users cannot access tables in the database until the database is unlocked. The database can be unlocked in two ways:

- The user can issue an UNLOCK DATABASE command.
- The user can terminate the session.

UPDATE

An UPDATE command modifies one or more rows of a specified table.

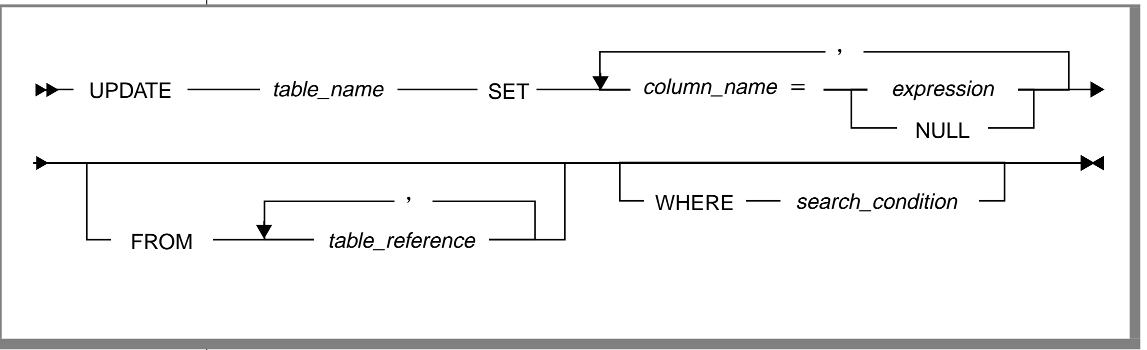
Authorization

To update rows of a table, a user must meet at least one of the following requirements:

- Be a member of the DBA system role.
- Have MODIFY_ANY authorization, either explicitly or through membership in a user-created role.
- Be the creator of the table.
- Have UPDATE privilege on the table.

Syntax

The following syntax diagram shows how to construct an UPDATE statement:



table_name Specifies the name of a table, temporary table, or synonym: *table_name* cannot name a view.

SET Contains one or more column names and expressions. The UPDATE command sets the values of the named column to *expression* or NULL.

The column names must:

- Already be defined in the named table.
- Occur once.
- Not be qualified with a table name or correlation name.

Aggregation functions (such as COUNT and SUM) and RISQL display functions (such as RANK and NTILE) are not allowed in the SET clause of an UPDATE statement.

FROM table_reference Specifies one or more additional tables from which values can be used to update rows. The tables specified by *table-reference* are joined, and then rows are updated in *table-name* as specified by the WHERE clause. *table-reference* must also include *table-name* (see the example at the end of this section).

The FROM *table-reference* clause is an alternative to using a subquery that provides improved performance.

The table(s) specified by *table-reference* must be base tables or synonyms. A FROM clause is not allowed when updating a model table.

If a correlation name is assigned to the updated table, the correlation name, not the actual table name, must be used to identify the updated table.

The FROM clause cannot be used to update a column by which the table being updated is segmented. For example, the following statement returns an error if table *t* is segmented by values of column *b*:

```
UPDATE t
  SET t.b = 7
  FROM t, s
    WHERE t.a = s.c
```

The *table_reference* clause is described in detail in "Table References" on page 7-13.

WHERE If the WHERE clause is omitted, all rows of the named table are updated; otherwise, those rows of the table that satisfy the search condition are updated. The search condition can contain a subquery.

For information about search conditions in general, refer to page 3-10; for information about the WHERE clause, refer to page 7-26.

Usage Notes

An UPDATE statement will be rejected if it attempts to:

- Update a primary-key column in a referenced table.
- Update a foreign-key column in a referencing table when the referenced table has no primary key index.
- Insert into a foreign key a value that has no corresponding primary-key value. (This action preserves referential integrity.)
- Set a primary key value to a value that already exists (primary keys must be unique).
- Set to NULL a column declared as NOT NULL.

When the above operations are attempted, the server returns an appropriate error message.

Unlike the SELECT statement, the UPDATE statement does not support outer joins in the WHERE clause. If an outer join is required, it must be specified in the FROM clause.

In the FROM clause, the table being modified cannot be specified:

- as the left table specified in a right outer join
- as the right table specified in a left outer join
- anywhere in a full outer join.

For example, the following UPDATE statement returns a syntax error because Sales, the table being updated, is specified on the right-hand side of the outer join:

```
update sales
    set dollars = dollars * 1.1
    from product left outer join sales
        on sales.prodkey = product.prodkey
    where product.prod_name = 'Veracruzano'
```

The following version of the UPDATE statement is valid:

```
update sales
    set dollars = dollars * 1.1
    from sales left outer join product
        on sales.prodkey = product.prodkey
    where product.prod_name = 'Veracruzano'
```

Examples

The following UPDATE statement updates rows of the Product_Promo table that satisfy its search condition:

```
update product_promo
    set descript = 'Espresso NO!',
        promo = 'March Wind',
        subpro = NULL
    where prod_id between 1020 and 1040
```

The following example illustrates the FROM clause. This example updates the sales table to increase revenue by 10 percent for the product "Veracruzano."

```
update sales
    set dollars = dollars * 1.1
    from sales, product
    where sales.prodkey = product.prodkey
        and sales.classkey = product.classkey
        and product.prod_name = 'Veracruzano'
```

Chapter

SET Commands

9

In This Chapter 9-5
SET ADVISOR LOGGING 9-6
SET ARITHIGNORE, ARITHABORT 9-7
SET AUTO INVALIDATE PRECOMPUTED VIEWS 9-8
SET COUNT RESULT 9-9
SET CROSS JOIN 9-10
SET DEFAULT DATA SEGMENT 9-11
SET DEFAULT INDEX SEGMENT 9-13
SET EXPORT_DEFAULT_PATH 9-15
SET EXPORT_MAX_FILE_SIZE 9-16
SET FIRST DAYOFWEEK 9-17
SET FORCE TASKS. 9-18
FORCE_SCAN_TASKS 9-20
FORCE_FETCH_TASKS and FORCE_JOIN_TASKS 9-21
FORCE_HASHJOIN_TASKS 9-23
FORCE_AGGREGATION_TASKS 9-24

SET IGNORE OPTICAL INDEXES 9-24
SET IGNORE PARTIAL INDEXES. 9-26
SET INDEX TEMPSPACE and SET QUERY TEMPSPACE 9-27
SET INFO MESSAGE LIMIT. 9-31
SET LOCK 9-32
SET OPTICAL AVAILABILITY 9-33
SET ORDER BY 9-36
SET PARALLEL_HASHJOIN 9-37
SET PARTIAL AVAILABILITY 9-38
SET PRECOMPUTED VIEW view_name 9-39
SET PRECOMPUTED VIEW QUERY REWRITE 9-40
SET PRECOMPUTED VIEWS FOR detail_table 9-41
SET QUERY MEMORY LIMIT 9-41
SET QUERYPROCS. 9-43
SET REPORT_INTERVAL 9-44
SET RESULT BUFFER and SET RESULT BUFFER FULL
ACTION 9-45
SET ROWCOUNT 9-46
SET ROWS_PER...TASK 9-48
ROWS_PER_SCAN_TASK 9-48
ROWS_PER_FETCH_TASK and ROWS_PER_JOIN_TASK 9-49

SET SEGMENTS 9-50
SET STATS 9-52
SET TEMPORARY SEGMENT STORAGE PATH 9-54
SET TRANSACTION ISOLATION LEVEL 9-55
SET UNIFORM PROBABILITY FOR ADVISOR 9-56
SET USE INVALID PRECOMPUTED VIEWS 9-57
SET USE LATEST REVISION 9-58
SET VERSIONING 9-59

In This Chapter

This chapter describes, in alphabetical order, the SET commands used to change the default behavior of the server during specific sessions. Global parameters that have the equivalent effect of these SET commands, but apply to all sessions, are specified in the *rbw.config* file. These parameters are documented in the *Administrator's Guide*.

SET commands that affect the behavior of the RISQL Entry Tool or RISQL Reporter and the Table Management Utility (TMU) are documented in the *RISQL Entry Tool and RISQL Reporter User's Guide* and the *Table Management Utility Reference Guide*, respectively.

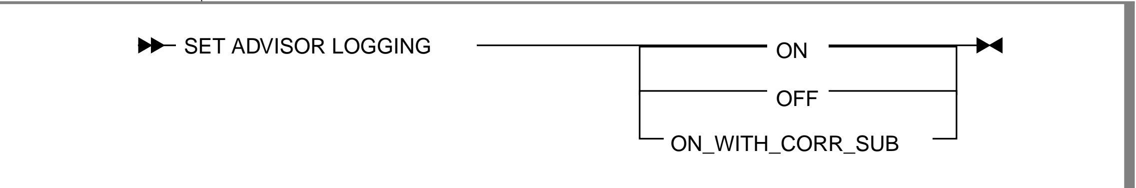
SET ADVISOR LOGGING

The SET ADVISOR LOGGING command enables or disables Informix Vista Advisor query logging for the current session. Advisor logging must be enabled, either with the ADMIN ADVISOR_LOGGING ON setting in the *rbw.config* file or with an ALTER SYSTEM START ADVISOR_LOGGING command, in order for the SET ADVISOR LOGGING command to take effect.

Use this command to control whether a particular query is or is not logged in the advisor log.

Syntax

The following syntax diagram shows how to construct a SET ADVISOR LOGGING statement:



When this parameter is set to ON_WITH_CORR_SUB, correlated subqueries, along with other queries that get rewritten, are logged. When it is set to ON, correlated subqueries are not logged. Use the OPTION ADVISOR_LOGGING *rbw.config* file parameter to set this parameter globally for all sessions. The default for the *rbw.config* file parameter is ON.

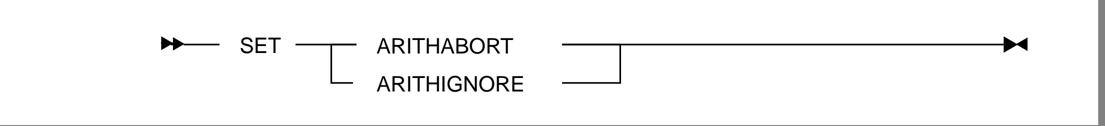
For more information about the Advisor, refer to the *Informix Vista User's Guide*.

SET ARITHIGNORE, ARITHABORT

The SET ARITHIGNORE command tells the server how to process queries when a divide-by-zero error occurs during the current server session.

Syntax

The following syntax diagram shows how to construct a SET ARITHIGNORE statement:



ARITHABORT	Instructs the server to terminate query processing and return an error message when a divide-by-zero error occurs during query execution.
ARITHIGNORE	Instructs the server to return NULL when a divide-by-zero error occurs during query execution.

Usage Note

This command overrides the default, which is ARITHABORT or as specified in the OPTION ARITHABORT or OPTION ARITHIGNORE parameter of the *rbw.config* file.

Example

The following statement instructs the server to return NULL when a divide-by-zero error occurs on queries executed during the current session:

```
set arithignore
```

SET AUTO INVALIDATE PRECOMPUTED VIEWS

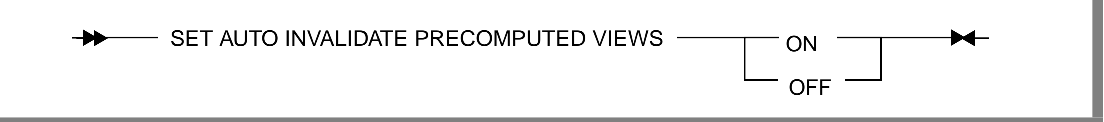
The SET AUTO INVALIDATE PRECOMPUTED VIEWS ON command automatically invalidates all the precomputed views that reference any detail table whose contents are modified with inserts, updates, and deletes or LOAD DATA operations after the views are created.

If this command is set to OFF, precomputed views must be marked invalid manually with the SET PRECOMPUTED VIEW *view_name* INVALID command.

The default setting is ON.

Syntax

The following syntax diagram shows how to construct a SET AUTO INVALIDATE PRECOMPUTED VIEWS statement:



This command also exists as the OPTION AUTO_ INVALIDATE_ PRECOMPUTED _VIEWS parameter.

For more information about precomputed views, refer to the *Informix Vista User's Guide*.

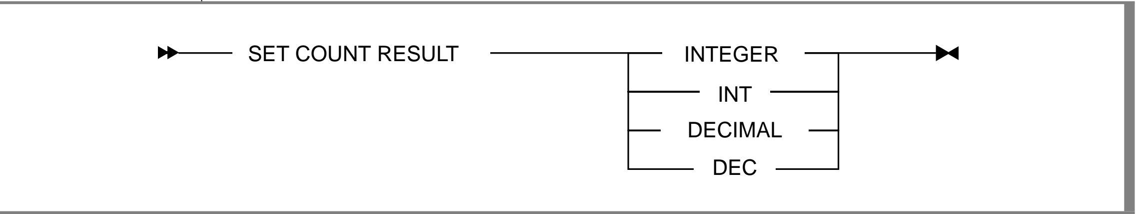
SET COUNT RESULT

The SET COUNT RESULT command specifies the datatype of results returned by the COUNT function during the current session. The default is INTEGER, which works for tables containing fewer than 2^{32} rows. However, for tables with 2^{32} or more rows, the COUNT RESULT parameter must be set to DECIMAL in order to get a correct count of those rows.

To specify the datatype for all server sessions, use the OPTION COUNT_RESULT parameter in the *rbw.config* file.

Syntax

The following syntax diagram shows how to construct a SET COUNT RESULT statement:



Usage Notes

When you set the COUNT RESULT parameter to DECIMAL or DEC, the result of the COUNT function is displayed as a DECIMAL(15,0) datatype. If nothing is specified, the result is displayed as an INTEGER datatype.

Example

The following SET COUNT RESULT statement sets the datatype for the results of the COUNT function to DECIMAL before a query that counts the rows of a large fact table is issued:

```
set count result dec
select count(*) from sales_us
```

SET CROSS JOIN

The SET CROSS JOIN command allows or disallows joins between tables that will produce the cross product (or Cartesian product) of the two tables. For example, if this command is set to ON, the query

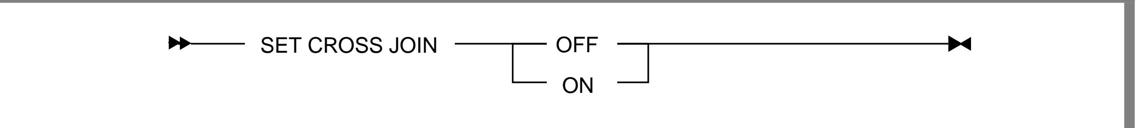
```
select * from market, product
```

will return every possible combination of rows from the Market and Product tables.

As a safeguard against the execution of unintended cross-join queries (resulting from an incorrect qualified-join specification, for example), the default behavior is to disallow such joins. If you want users to be able to use cross joins during a specific session, set this command to ON. Alternatively, you can reset the OPTION CROSS_JOIN parameter in the *rbw.config* file, which applies to all server sessions.

Syntax

The following syntax diagram shows how to construct a SET CROSS JOIN statement:



Usage Notes

Depending on the availability of indexes defined on the joining columns, cross-join processing is sometimes required for queries that contain non-equijoins—such as join predicates that express a less-than or greater-than relationship.

Using the Aroma database, the following query would require a cross join because the Date and Start_Date columns are not indexed:

```
select date
from period join promotion on date < start_date
```

However, the following similar query does not require a cross join because the Classkey columns are indexed (by default, as primary key columns).

```
select prod_name
from product join class on product.classkey > class.classkey
```

For more information about cross joins, refer to page 7-12.

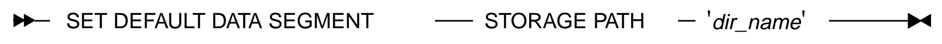
SET DEFAULT DATA SEGMENT

The SET DEFAULT DATA SEGMENT command specifies a directory location for the physical storage units (PSUs) of default data segments created during the current session.

To set the storage path for a temporary data segment, refer to "SET TEMPORARY SEGMENT STORAGE PATH" on page 9-54.

Syntax

The following syntax diagram shows how to construct a SET DEFAULT DATA SEGMENT statement:



STORAGE PATH ***dir_name*** Specifies the full pathname (***dir_name***) of the directory to contain the PSUs of default data segments. The directory can be created either before or after the SET DEFAULT DATA SEGMENT command is issued; however, it must be created before creating a table in a default segment.

Usage Notes

This command overrides the default, which is specified in the OPTION DEFAULT_DATA_SEGMENT parameter of the *rbw.config* file. If a default directory is not specified with the SET DEFAULT DATA SEGMENT command or in the *rbw.config* file, the PSUs of the default segments are stored in the database directory.

In a database server with multiple databases, ensure that the default directory is different for each database. If you use this command, be careful not to specify the default directory of another database. Two ways to ensure that each database points to a different directory are:

- Do not specify a directory with either the OPTION DEFAULT_DATA_SEGMENT parameter or with this command. PSUs will be created in the relevant database directory.
- Include this command in the *.rbwrc* file of each database. Each *.rbwrc* file should specify a different directory for default data segments.

Example

The following statement specifies the directory that will contain the PSUs of all default row data segments:

UNIX

```
set default data segment
    storage path '/default/dataseg_dir'
```

♦

WIN NT

```
set default data segment
    storage path 'c:\dsk1\dsegs'
```

♦

SET DEFAULT INDEX SEGMENT

The SET DEFAULT INDEX SEGMENT command specifies a directory location for the physical storage units (PSUs) of all default index segments created during the current session.

To set the storage path for a temporary index segment, refer to "SET TEMPORARY SEGMENT STORAGE PATH" on page 9-54.

Syntax

The following syntax diagram shows how to construct a SET DEFAULT INDEX SEGMENT statement:

```
►►─ SET DEFAULT INDEX SEGMENT ── STORAGE PATH ─ 'dir_name' ──►◄
```

STORAGE PATH ***dir_name*** Specifies the full pathname (***dir_name***) of the directory to contain the PSUs of default index segments. The directory can be created either before or after the SET DEFAULT INDEX SEGMENT command is issued; however, it must be created before creating an index in a default segment.

Usage Notes

This command overrides the default, which is specified in the OPTION DEFAULT_INDEX_SEGMENT parameter of the *rbw.config* file. If a default directory is not specified with the SET DEFAULT INDEX SEGMENT command or in the *rbw.config* file, the PSUs of the default segments are stored in the database directory.

In a server with multiple databases, ensure that the default directory is different for each database. If you use this command, be careful not to specify the default directory of another database. Two ways to ensure that each database points to a different directory are:

- Do not specify a directory with either the OPTION DEFAULT_INDEX_SEGMENT parameter or with this command. PSUs will be created in the relevant database directory.
- Include this set command in the *.rbwrc* file of each database. Each *.rbwrc* file should specify a different directory for default index segments.

Example

The following statement specifies the directory that will contain the PSUs of all default row data segments:

UNIX

```
Set default index segment
    storage path '/default/indexseg_dir'
```

♦

WIN NT

```
Set default index segment
    storage path 'c:\dsk1\ixsegs'
```

♦

SET EXPORT_DEFAULT_PATH

The SET EXPORT_DEFAULT_PATH command specifies the base path that is prepended to an unqualified file name specification for any OUTPUT file, TMUFILE, or DDLFILE file created by the EXPORT command.

Syntax

The following syntax diagram shows how to construct a SET EXPORT_DEFAULT_PATH statement:

```
►►── SET EXPORT_DEFAULT_PATH ──── path_specification ────►◄
```

path_ specification	Specifies the base pathname that is added to an unqualified file name specification. The default value is the same as that of the platform-specific spill directory specified for Red Brick Warehouse at installation.

Usage Notes

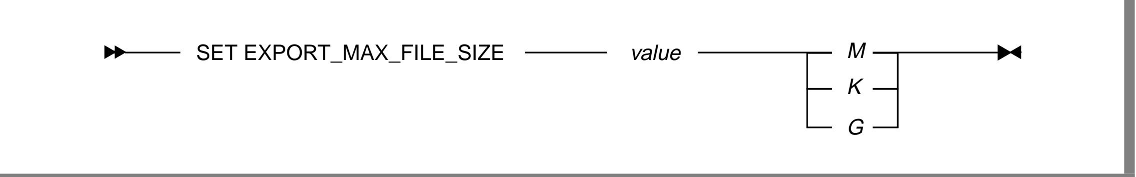
This command also exists as the OPTION EXPORT_DEFAULT_PATH parameter in the *rbw.config* file.

SET EXPORT_MAX_FILE_SIZE

The SET EXPORT_MAX_FILE_SIZE command sets the maximum size of the export data file used by the EXPORT command. If this value is exceeded by the amount of data, additional output files are created with '*.nnn*' appended to the filename, beginning with *nnn* = 000.

Syntax

The following syntax diagram shows how to construct a SET EXPORT_MAX_FILE_SIZE statement:



value Indicates a positive integer. The default value is 0, which indicates no Red Brick Warehouse enforced limit.

M, K, G Indicates the unit of measurement: Megabytes, Kilobytes, or Gigabytes. The default value is ***M***.

Usage Notes

As many as 999 additional files can be created, each one containing complete rows of data. (No partial row is written to a file; the overflow condition is detected and a new file is created.) No limits are applied when the EXPORT command is writing to a pipe.

This command also exists as the OPTION EXPORT_MAX_FILE_SIZE parameter in the *rbw.config* file.

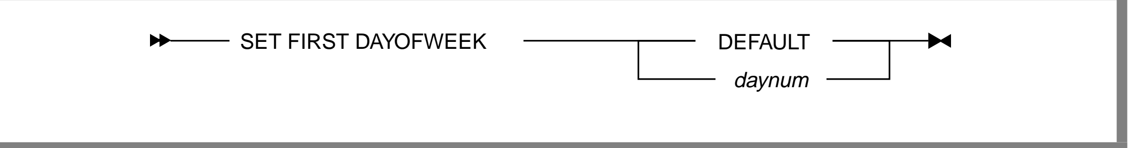
SET FIRST DAYOFWEEK

The SET FIRST DAYOFWEEK command overrides the default numbering system defined by the server locale for the days of the week. This command affects the results returned by the EXTRACT and DATENAME functions when the *weekday* argument is used. (For detailed information about these functions, refer to Chapter 5, "Scalar Functions.")

To specify the first day of the week for all server sessions, use the NLS_LOCALE FIRST_DAYOFWEEK parameter in the *rbw.config* file.

Syntax

The following syntax diagram shows how to construct a SET FIRST DAYOFWEEK statement:



DEFAULT Resets the first day of the week to its default value in the current server locale. For example, in a German database, the default value is 2 (Monday).

daynum Specifies an integer value from 1 to 7 that maps to the days of the week in the default U.S.-English locale: 1 = Sunday, 2 = Monday, and so on.

Usage Notes

This command has no effect on *day names*. Setting the first day of the week to a different number does not change how the server assigns day names to dates. For example, January 1, 1998, is *Thursday* in an English database and *Donnerstag* in a German database regardless of which day of the week is treated as the first.

The RBW_OPTIONS system table stores the current value of this parameter under the option name FIRST_DAYOFWEEK.

Example

The following statement sets the first day of the week to 7 (Saturday):

```
set first dayofweek 7
```

SET FORCE TASKS

This section describes four related SET commands:

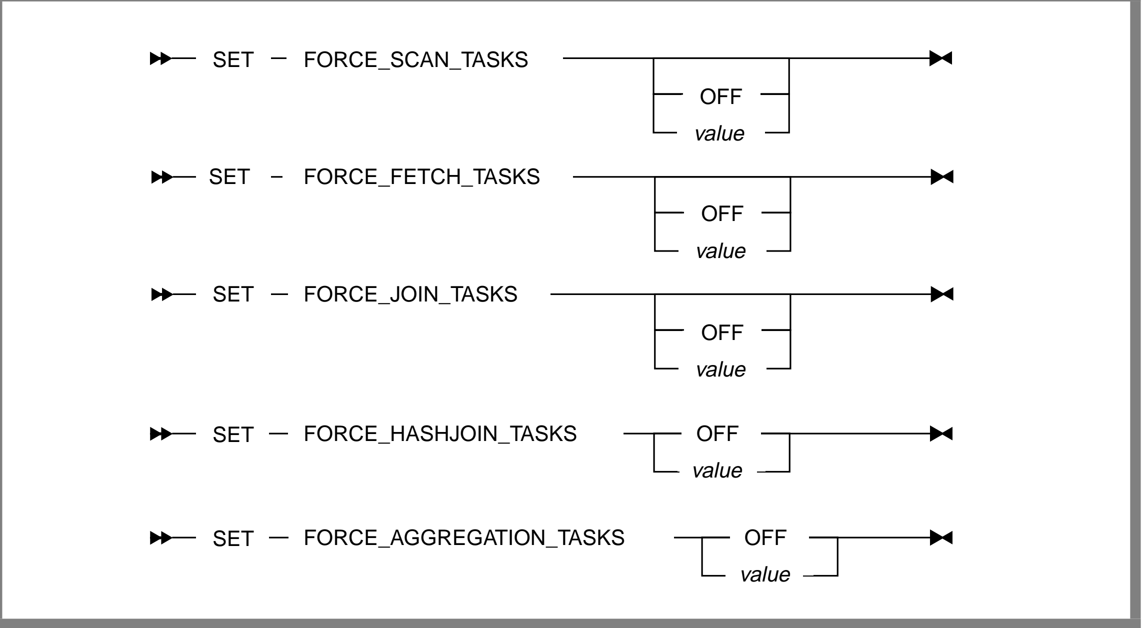
- SET FORCE_SCAN_TASKS
- SET FORCE_FETCH_TASKS
- SET FORCE_JOIN_TASKS
- SET FORCE_HASHJOIN_TASKS
- SET FORCE_AGGREGATION_TASKS

The first three commands behave as overrides to the ROWS_PER_TASK parameters and specify the number of parallel tasks (or processes) to be used for query processing regardless of row count. To control the extent of parallelism by specifying the number of rows per parallel task, see "SET ROWS_PER...TASK" on page 9-48.

For general information about parallel-query processing, refer to the *Administrator's Guide*.

Syntax

The SET commands must be set to OFF (the default) or a numeric value. OFF means that explicit control of parallelism is not enabled.



If you issue one of these SET commands without entering a numeric value or *OFF,* the system returns the current setting for that parameter:

```
set force_scan_tasks
** INFORMATION ** (1433) FORCE_SCAN_TASKS is currently set
to 6.
```

You can also enter these commands as TUNE parameters in the *rbw.config* file so they affect all server sessions.

FORCE_SCAN_TASKS

The value set for FORCE_SCAN_TASKS controls the number of parallel tasks for relation scans. However, the FORCE_SCAN_TASKS value does not guarantee that a certain number of parallel processes will be used. The actual number of processes used will be the *lowest* of these three values:

- The FORCE_SCAN_TASKS value.
- The number of PSUs over which the table is distributed.
- The number of processes that can be allocated from the QUERYPROCS and TOTALQUERYPROCS pool.

Also note the following points regarding task allocation for relation scans:

- If FORCE_SCAN_TASKS is set, the ROWS_PER_SCAN_TASK value is ignored.
- If FORCE_SCAN_TASKS is set to a value that is greater than the number of disk groups, some disk groups will simply be allocated more than one process. (When FORCE_SCAN_TASKS is set, the number of disk groups does not influence the behavior of parallel processing.)

  Disk groups are discussed in the *Administrator's Guide*.

Example

Assume the following settings:

FORCE_SCAN_TASKS	16
PSUs in table	18
QUERYPROCS	18
TOTALQUERYPROCS	24

Whether the FORCE_SCAN_TASKS value is used in this case depends on the number of processes *available* from the TOTALQUERYPROCS pool. If only 6 processes are already allocated, 18 processes will be available so the FORCE_SCAN_TASKS value of 16 will be used.

After 50 percent of the TOTALQUERYPROCS pool has been allocated, subsequent queries are allocated fewer processes per query.

FORCE_FETCH_TASKS and FORCE_JOIN_TASKS

The values set for FORCE_FETCH_TASKS and FORCE_JOIN_TASKS control the number of parallel tasks for fetching rows and joining tables in queries that use a STAR index. If either of these values is greater than or equal to 1, it will override the corresponding value set for ROWS_PER_FETCH_TASK or ROWS_PER_JOIN_TASK.

However, the FORCE_FETCH_TASKS and FORCE_JOIN_TASKS values do not guarantee that a certain number of parallel processes will be used. The actual number of processes used to fetch rows will be the *lowest* of these three values:

- The FORCE_FETCH_TASKS value.
- The number of PSUs over which the table is distributed.
- The number of processes available from the QUERYPROCS and TOTALQUERYPROCS pool.

The actual number of processes used to join tables will usually be the *lowest* of these two values:

- The FORCE_JOIN_TASKS value.
- The number of processes available from the QUERYPROCS and TOTALQUERYPROCS pool.

In rare cases, the FORCE_JOIN_TASKS value might be greater than the number of STAR index rows that match the constraints in the query; therefore, it will not be possible to logically divide and process the query by the specified number of tasks. Instead, the number of matching rows will be used to set the limit on parallel join tasks.

Also note the following points regarding task allocation for fetching rows and joining tables:

- You do not have to force both fetch and join tasks. For example, you can force join tasks but allow fetch tasks to be computed dynamically.
- If FORCE_FETCH_TASKS is set, the ROWS_PER_FETCH_TASK value is not used; similarly, if FORCE_JOIN_TASKS is set, the ROWS_PER_JOIN_TASK value is not used.

- Although the number of PSUs over which the STAR-indexed table is distributed affects the allocation of parallel fetch tasks, the number of disk groups does not.
- The number of PSUs used to partition the STAR index does not affect the allocation of parallel join tasks.
- For joins that involve more than one STAR-indexed table, one such table is selected to control the partitioning. If 10 PSUs are used to distribute the chosen table, 10 processes are available for fetch-task partitioning.
- If there are fewer than the requested number of processes available from the QUERYPROCS and TOTALQUERYPROCS pool and both FORCE options are set, the system tries to preserve the ratio of FORCE_JOIN_TASKS to FORCE_FETCH_TASKS values.

Examples

Assume the following settings:

FORCE_FETCH_TASKS	16
PSUs in table	18
QUERYPROCS	18
TOTALQUERYPROCS	24

In this case, whether the FORCE_FETCH_TASKS value will be used depends on the number of processes *available* from the TOTALQUERYPROCS pool. If only 6 processes are already allocated, 18 processes will be available and the FORCE_FETCH_TASKS value of 16 will be used.

Assume the following settings:

FORCE_JOIN_TASKS	8
QUERYPROCS	12
TOTALQUERYPROCS	30

If there are 9 or more processes available from the TOTALQUERYPROCS pool, the FORCE_JOIN_TASKS value will be used.

FORCE_HASHJOIN_TASKS

The value set for FORCE_HASHJOIN_TASKS controls the number of parallel tasks for hybrid hash joins. However, the FORCE_HASHJOIN_TASKS value does not guarantee that a certain number of parallel processes will be used. The actual number of processes used will be the *lowest* of the following values:

- The FORCE_HASHJOIN_TASKS value.
- The number of processes that can be allocated from the QUERYPROCS and TOTALQUERYPROCS pool.

Example

Assume the following settings:

FORCE_HASHJOIN_TASKS	8
QUERYPROCS	12
TOTALQUERYPROCS	30

If there are 10 or more processes available from the TOTALQUERYPROCS pool, the FORCE_HASHJOIN_TASKS value will be used.

Usage Notes

Also note the following points regarding task allocation for parallel hybrid hash joins:

- The PARALLEL_HASHJOIN option must be set to ON, either with a SET PARALLEL_HASHJOIN ON command or a TUNE PARALLEL_HASHJOIN ON parameter, in order to get any parallelism from hybrid hash joins.
- You must have at least 2 more processes available from the QUERYPROCS and TOTALQUERYPROCS pool than the value you specify in FORCE_HASHJOIN_TASKS in order to achieve that level of parallelism. For example, in order to get 8 parallel hash join processes, you must specify FORCE_HASHJOIN_TASKS to 8 and have at least 10 processes available from the QUERYPROCS and TOTALQUERYPROCS pool.

FORCE_AGGREGATION_TASKS

The value set for FORCE_AGGREGATION_TASKS controls the number of aggregation tasks for a single session. However, the FORCE_AGGREGATION_TASKS value does not guarantee the specified number. The actual number of processes used will have an upper bound of the lower of the following two values:

- The FORCE_AGGREGATION_TASKS value.
- The number of processes that can be allocated from the QUERYPROCS and TOTALQUERYPROCS pool.

SET IGNORE OPTICAL INDEXES

Whether indexes with PSUs that reside on optical storage are considered when the best index for a query is selected depends on the setting of the IGNORE_OPTICAL_INDEXES option. If a query is processed and an error or warning message is issued stating that an optical index was used, and you know that other fully available but less optimal indexes exist, you can set the IGNORE_OPTICAL_INDEXES option to force the use of an index not residing on optical storage.

In most cases, frequently used indexes will not reside in optical segments. Storing an index on slower optical devices defeats the purpose of the index.

Syntax

The following syntax diagram shows how to specify whether indexes stored in optical segments should be used for specific sessions:

OFF Specifies that all indexes, even those in optical segments, are to be considered in selecting the best index. If an index with an optical segment is determined to be the best choice, the setting for the OPTICAL AVAILABILITY option controls how the operation proceeds. For further information on this option see "SET OPTICAL AVAILABILITY" on page 9-33. The default is OFF.

ON Specifies that only indexes stored entirely on non-optical storage are to be considered in selecting the best index. If no applicable index meets this criterion, an error message is issued and the operation fails.

Example

To specify the use of only those indexes stored non-optically, enter:

```
set ignore optical indexes on
```

SET IGNORE PARTIAL INDEXES

The SET IGNORE PARTIAL INDEXES command specifies whether or not partially available indexes should be considered when the system chooses the best strategy for processing a query during the current session. A partially available index has one or more offline index segments.

Syntax

The following syntax diagram shows how to construct a SET IGNORE PARTIAL INDEXES statement:

OFF Specifies that all indexes, even partially available indexes, are to be considered in selecting the best index for a query. If a partially available index is determined to be the best choice, either the OPTION PARTIAL_AVAILABILITY parameter in the *rbw.config* file or the SET PARTIAL AVAILABILITY command defines how the query is processed.

ON Specifies that only fully available indexes are to be considered in selecting the best index for a query. If no index is fully available, an error message is issued and the query fails.

Usage Note

This command overrides the default, which is OFF or as specified in the OPTION IGNORE_PARTIAL_INDEXES parameter in the *rbw.config* file.

Examples

The following statement specifies that during the current session only fully available indexes are considered when an index is selected for a query:

```
set ignore partial indexes on
```

SET INDEX TEMPSPACE and SET QUERY TEMPSPACE

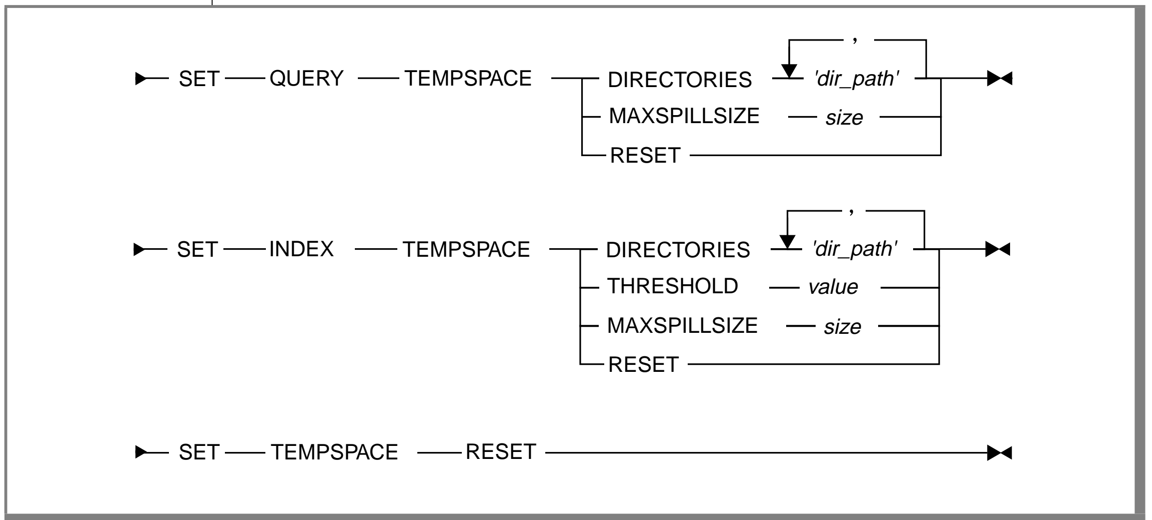
The SET INDEX TEMPSPACE command specifies the directory, threshold size, and maximum file size of spill files created for index building during the current session.

The SET QUERY TEMPSPACE command specifies the directory and maximum file size of spill files created for query processing during the current session.

Spill files reside on disk and contain the intermediate results of each index as it is being built or each query as it is being processed. The intermediate results initially reside in main memory, then spill to disk at the threshold value. For more information about spill files, refer to the *Informix Red Brick Decision Server Administrator's Guide*.

Syntax

The following syntax diagram shows how to construct SET INDEX TEMPSPACE and SET QUERY TEMPSPACE statements:



DIRECTORY dir_path, DIRECTORIES 'dir_path', ...	Specifies a directory or a set of directories to be used for temporary files; *dir_path* must be a full pathname. To define a set of directories using entries in the *rbw.config* file, enter multiple lines. The order in which the directories are specified has no effect because the order in which the temporary-space directories are used is random (determined internally) and no user control is possible.
	UNIX If no temporary-space directories are defined, the default directory is */tmp.* ♦
	WIN NT If no temporary-space directories are defined, the default directory is %TEMP%, or if not set, *c:\tmp.* ♦
THRESHOLD value	Specifies the amount of memory used before the intermediate results from index-building operations are written to disk. For operations involving multiple indexes, this threshold value is allocated equally among the indexes being built. The default value is 10 megabytes (10M).
	The size can be specified as kilobytes (K) or megabytes (M) by appending K or M to the number. Note that no space is allowed between the number and the unit identifier (K, M). For example: 1024K, 500M.
	The threshold value must be specified before the corresponding MAXSPILLSIZE value is specified; it must precede the MAXSPILLSIZE entry in the *rbw.config* file.
	A value of 0 causes files to be written to disk after the first 200 index entries.
	There is no THRESHOLD parameter for query-processing operations.

MAXSPILLSIZE size	Specifies the total maximum amount of temporary space per operation. For an index-building operation involving multiple indexes, this space is allocated equally among the indexes being built. For query operations, however, the entire value is allocated to each query and to each of its subqueries, if any. The size can be specified as kilobytes (K), megabytes (M), or gigabytes (G) by appending K, M, or G to the number. Note that no space is allowed between the number and the unit identifier (K, M, G). For example: 1024K, 500M, 8G. The default MAXSPILLSIZE value is 1 gigabyte. The maximum MAXSPILLSIZE value is 2047 gigabytes.
RESET	Resets the query or index TEMPSPACE parameters to the values specified in the *rbw.config* file. If neither QUERY nor INDEX is specified, all TEMPSPACE parameters are reset.

Examples

The following examples illustrate SET commands that can be used to change parameters for a specific session:

UNIX

```
set index tempspace directories '/disk1/itemp',
    '/disk2/itemp', '/disk3/itemp'
set index tempspace threshold 2M
set index tempspace maxspillsize 3G
```

♦

WIN NT

```
set index tempspace directories 'd:\itemp',
    'e:\itemp', 'f:\itemp'
set index tempspace threshold 2M
set index tempspace maxspillsize 3G
```

♦

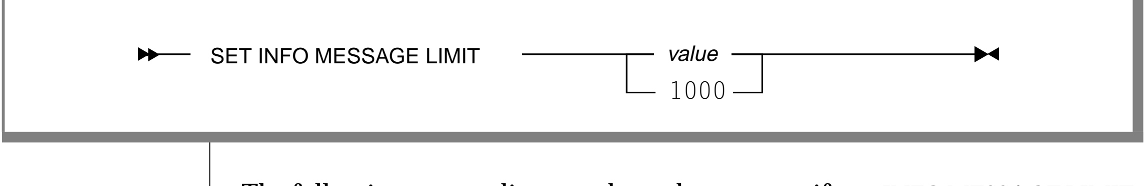
SET INFO MESSAGE LIMIT

The INFO_MESSAGE_LIMIT parameter is set by default to 1,000 to prevent the return of an excessive number of messages when the SET STATS INFO command is in use and correlated subqueries are issued. (See page -52.) A maximum of 1,000 informational messages (that is, messages labeled either "STATISTICS" or "INFORMATION") is returned per query; however, all queries are fully executed, and warning and error messages are returned as normal.

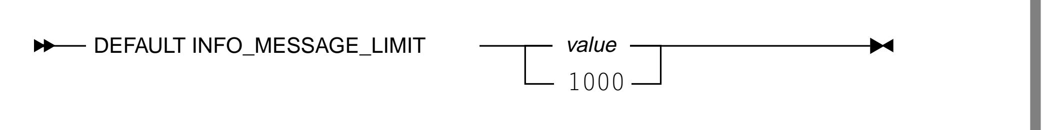
You can increase or decrease the message limit by either adding a DEFAULT INFO_MESSAGE_LIMIT value to the configuration file (*rbw.config*) or using a SET command. Informational messages are stored in memory while each query is being executed, so the extent to which you can safely increase the the message limit is a function of how much memory is available on your system.

Syntax

The following syntax diagram shows how to construct a SET INFO MESSAGE LIMIT statement:



The following syntax diagram shows how to specify an INFO MESSAGE LIMIT parameter in the configuration file:



In both cases, *value* must be an integer. If *value* is set to 0, there is no limit to the number of messages returned.

Example

If you set the message limit to 2000:

```
set info message limit 2000
```

and issue a correlated subquery that normally would return many more messages, only the first 2,000 informational messages are displayed, followed by an explanatory warning message:

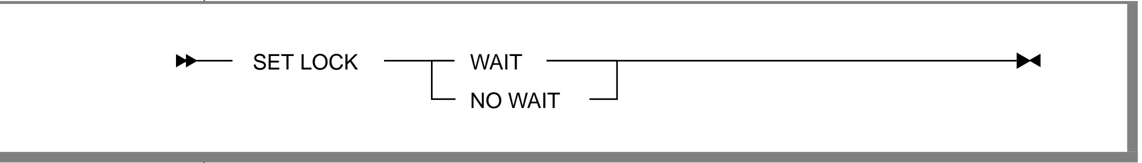
```
** WARNING ** (1443) No more informational messages will be
reported for this query due to the INFO_MESSAGE_LIMIT
constraint. Informational messages generated: 2000.
```

SET LOCK

The SET LOCK command specifies the behavior of table locks and database locks when another user is accessing a table or has already locked it.

Syntax

The following syntax diagram shows how to construct a SET LOCK statement:



WAIT Specifies that when a lock is needed the server session is suspended until existing locks are released and the new lock is acquired.

In situations that could result in deadlock, the lock request is denied and an error message is returned. (The WAIT setting is ignored.)

NO WAIT Specifies that when a lock is needed, the lock request fails and an error message is returned if the table(s) are locked or are being accessed by other users.

Usage Note

This command takes effect when a user issues a LOCK (table) or LOCK DATABASE command during the current session. The setting specified with this command can be overridden in a LOCK or LOCK DATABASE command query that contains the NO WAIT option.

SET OPTICAL AVAILABILITY

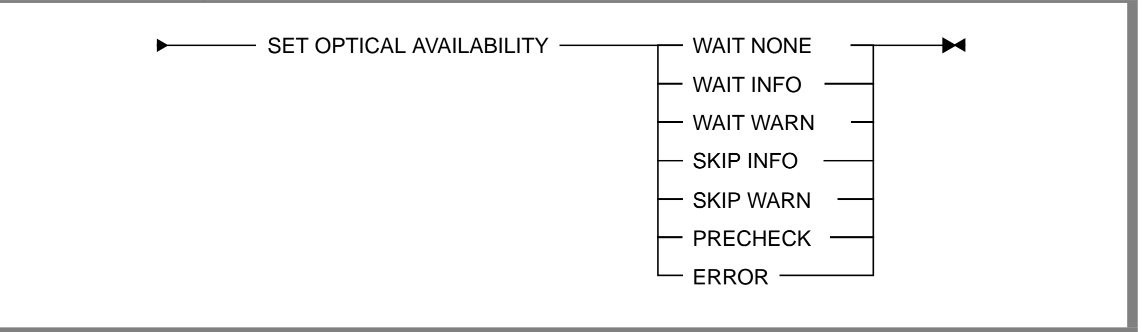
Access to data or indexes in optical segments depends on the setting of the SET OPTICAL_AVAILABILITY command and whether the commands used to access the data involve the following read and write operations:

- Read operations: SELECT statements and TMU UNLOAD statements
- Write operations:
  - ALTER TABLE and DROP TABLE statements
  - CREATE INDEX and ALTER INDEX statements
  - ALTER SEGMENT statements
  - INSERT, UPDATE, and DELETE statements
  - TMU LOAD DATA and REORG statements

The settings in the following syntax diagram apply as described to this list of commands only.

Syntax

To specify the availability of optically stored data and indexes for specific sessions, enter a SET command with the following syntax:



WAIT NONE Specifies that an operation is to wait for row data or indexes stored in optical segments. No message regarding optical storage access is issued. The default is WAIT_NONE.

WAIT INFO Specifies that an operation is to wait for row data or indexes stored in optical segments. An informational message is issued stating that optical storage is being accessed.

WAIT WARN Specifies that an operation is to wait for row data or indexes stored in optical segments. A warning message is issued stating that optical storage is being accessed.

SKIP INFO Specifies that for read operations any optical segments containing row data or indexes are to be skipped. An informational message is issued stating that such segments were skipped.

For write operations, this option is ignored and the command is processed as if PRECHECK were specified.

SKIP WARN Specifies that for read operations any optical segments containing row data or indexes are to be skipped. A warning message is issued stating that such segments were skipped.

For write operations, this option is ignored and the command is processed as if PRECHECK were specified.

PRECHECK Specifies that for any operation a check for optical segments is to be performed before the operation is performed. If any optical segments are encountered, an error message is issued and the processing terminates.

ERROR Specifies that for read operations the statement is to be processed, but if any optical segments are encountered, an error message is to be issued and the processing terminates. Note that an operation might process for a significant amount of time before encountering an optical segment.

For write operations, this option is ignored and the statement is processed as if PRECHECK were specified.

Example

The following example illustrates how to set one type of access behavior for optically stored data and indexes:

```
set optical availability error
```

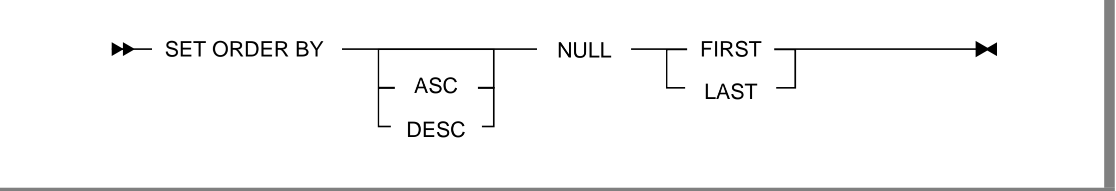
SET ORDER BY

The SET ORDER BY command modifies the server's placement of nulls in an ordered column.

The ORDER BY clause can override this default setting. For additional information about dynamic null placement, refer to page 7-47.

Syntax

The following syntax diagram shows how to construct a SET ORDER BY statement:



ASC, DESC Specifies an ascending or descending order.

FIRST, LAST Specifies the placement of nulls.

Usage Note

This command takes effect when a user issues a SELECT statement with an ORDER BY clause during the current session. The setting specified with this command can be overridden in a query that contains the NULL placement keyword in the ORDER BY clause.

Example

The following statement directs the server to place nulls last in a column that is in descending order:

```
set order by desc null last
```

SET PARALLEL_HASHJOIN

The SET PARALLEL HASHJOIN command determines whether parallel processing is allowed for hybrid hash joins during the current session. You can also enter this command as a TUNE parameter in the *rbw.config* file so it affects all server sessions.

Syntax

The following syntax diagram shows how to construct a SET PARALLEL HASHJOIN statement:

OFF, ON Specifies whether parallel processing is allowed for hybrid hash joins; the default is OFF.

To determine the current setting of this parameter, do not specify ON or OFF:

```
set parallel_hashjoin
** INFORMATION ** (1434) PARALLEL_HASHJOIN is currently set
to OFF.
```

Usage Note

This command must be set to ON if you want to use the SET FORCE_HASHJOIN_TASKS command to control the number of parallel processes used for hybrid hash joins.

Example

The following statement directs the server to allow parallel processing for hybrid hash joins:

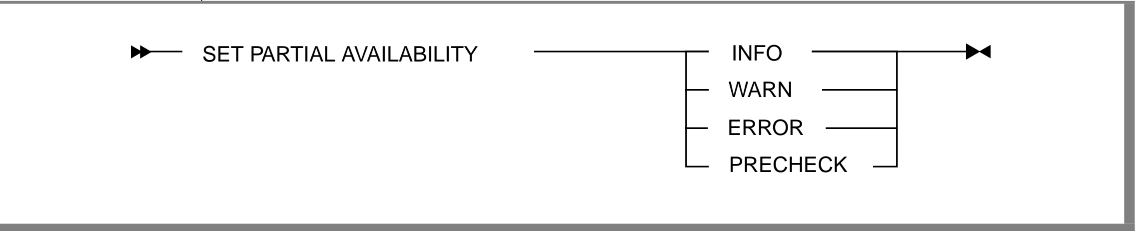
```
set parallel_hashjoin on
```

SET PARTIAL AVAILABILITY

The SET PARTIAL AVAILABILITY command specifies how queries behave against partially available tables during the current session. A partially available table has one or more offline row data segments or index segments for the index to be used for the query.

Syntax

The following syntax diagram shows how to construct a SET PARTIAL AVAILABILITY statement:



INFO Specifies that the query is to be processed even if a data or index segment that the query needs to access is unavailable. If the results would be different if the table were fully available, an informational message to this effect is issued along with the results.

WARN Specifies the same behavior as INFO, but the message returned is a warning, not an informational message.

ERROR Specifies the query is to be processed even if a row data or index segment of a table that the query needs to access is unavailable. If the results would be different if the table were fully available, no results are returned and an error message is issued.

PRECHECK Specifies that the availability of tables and indexes the query needs to access is to be checked before the query is processed. If a table is only partially available, the system issues an error message and the query is not processed.

Usage Notes

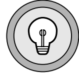
When a query contains multiple select expressions (a UNION operation) or a partial SELECT statement (subquery), the first select expression is previewed and processed, then the subsequent select expressions are previewed and processed. If the second select expression in a query results in an error because of a partially available table, the error message is not returned until the first select expression is processed. In other words, because the first select expression is processed before the second select expression is previewed, it might take longer than expected for the system to return the error.

The results and messages returned are based on all of the select expressions in the query and the behavior set for partial availability.

This command overrides the default, which is either PRECHECK or as specified in the OPTION PARTIAL_AVAILABILITY parameter of the *rbw.config* file.

SET PRECOMPUTED VIEW view_name

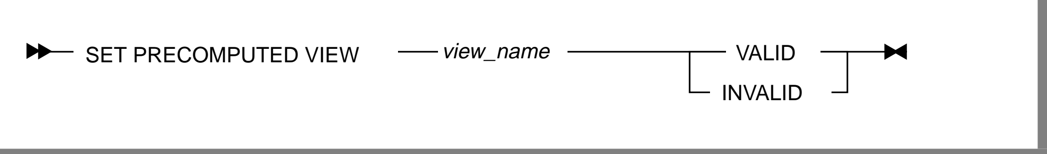
The SET PRECOMPUTED VIEW *view_name* command marks a precomputed view valid or invalid by updating the RBW_VIEWS table. There is no default for this command.



***Tip:** In Vista, the effects of SET commands apply to the database and persist across sessions.*

Syntax

The following syntax diagram shows how to construct a SET PRECOMPUTED VIEW *view_name* command:



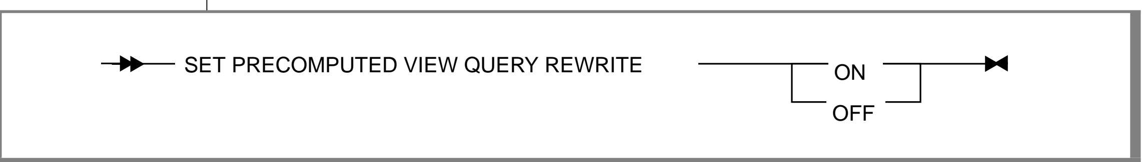
view_name Specifies the name of the precomputed view.

SET PRECOMPUTED VIEW QUERY REWRITE

The SET PRECOMPUTED VIEW QUERY REWRITE command turns the aggregate query rewrite system ON or OFF. The default setting is ON.

Syntax

The following syntax diagram shows how to construct a SET PRECOMPUTED VIEW QUERY REWRITE command:



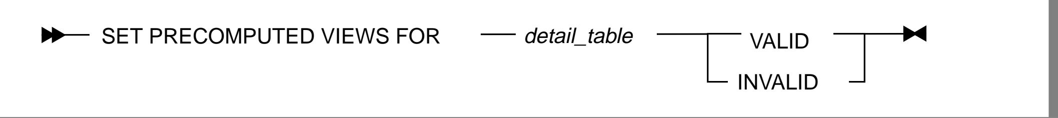
This command also exists as the OPTION PRECOMPUTED_VIEW_ QUERY_ REWRITE parameter in the *rbw.config* file.

SET PRECOMPUTED VIEWS FOR detail_table

The SET PRECOMPUTED VIEWS FOR detail_table command marks the data of all precomputed views for a detail table valid or invalid. There is no default for this command. However, upon creation, all precomputed views are marked invalid automatically

Syntax

The following syntax diagram shows how to construct a SET PRECOMPUTED VIEWS FOR *detail_table* command:



detail_table Specifies the name of the base table used in the creation of a precomputed view.

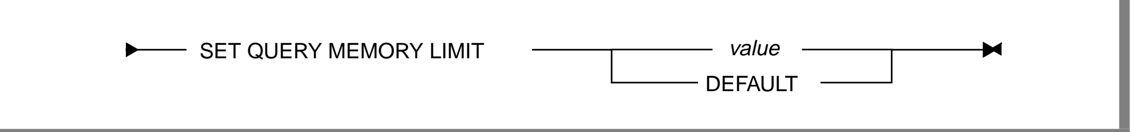
For more information about precomputed views, refer to the *Informix Vista User's Guide*.

SET QUERY MEMORY LIMIT

The SET QUERY MEMORY LIMIT command specifies a limit on the working memory available for the execution of a query or an INSERT, UPDATE, or DELETE statement during the current session. When the specified limit is reached, the memory spills to disk. For more information about memory-tuning parameters, refer to the *Administrator's Guide*.

Syntax

The following syntax diagram shows how to construct a SET QUERY MEMORY LIMIT statement:



value Specifies the memory limit in kilobytes (K), megabytes (M) or gigabytes (G). You must specify a *K*, an *M*, or a *G*, and the value must fall in the range of 2M (or 2048K) to 4G. No space is allowed between the number and the letter.

The DEFAULT keyword sets the limit to the value specified in the *rbw.config* file (or to the default of 50M if nothing is specified in that file).

Usage Notes

This command overrides the current setting of the TUNE QUERY_MEMORY_LIMIT parameter in the *rbw.config* file.

Example

The following script sets the memory limit for the query *sales_report99* to 100 megabytes, runs the query, then resets the limit to the default value:

```
set query memory limit 100M
run sales_report99
set query memory limit default
...
```

SET QUERYPROCS

The SET QUERYPROCS command specifies the limit on the total number of parallel processes available for individual queries executed during the current session.

Syntax

The following syntax diagram shows how to construct a SET QUERYPROCS statement:

num_per_query — An integer from 0 to 32767, inclusive.

A value of 0 or 1 effectively turns off parallel query processing.

Usage Notes

This command overrides the default, which is specified in the TUNE QUERYPROCS option of the *rbw.config* file. The server accepts any value from 0 to the value specified in the *rbw.config* file. A value greater than that specified in the *rbw.config* file is ignored, as are numbers less than 0.

Example

The following statement tells the server to use at most 5 processes for each query executed during the current server session:

```
set queryprocs 5
```

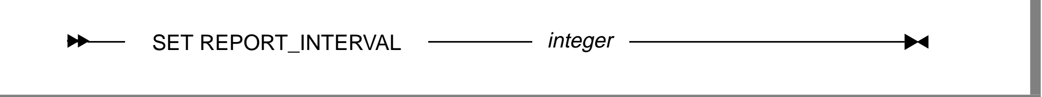
SET REPORT_INTERVAL

The SET REPORT_INTERVAL command specifies the maximum time interval that elapses before the session sends a report to the *rbwadmd* daemon. The *rbwadmd* daemon uses this information to update the dynamic statistic tables (DSTs).

A database event can trigger a report before this time interval elapses. Whenever a statement changes state or a session requests, acquires, or releases a lock, the server sends a report to the *rbwadmd* daemon process. The states of a statement include connecting, idle, executing, compiling, calculating, returning rows, sorting, building indexes, and inserting.

Syntax

The following syntax diagram shows how to construct a SET REPORT_INTERVAL statement:



integer A positive integer that specifies the maximum number of minutes between reports to the *rbwadmd* daemon.

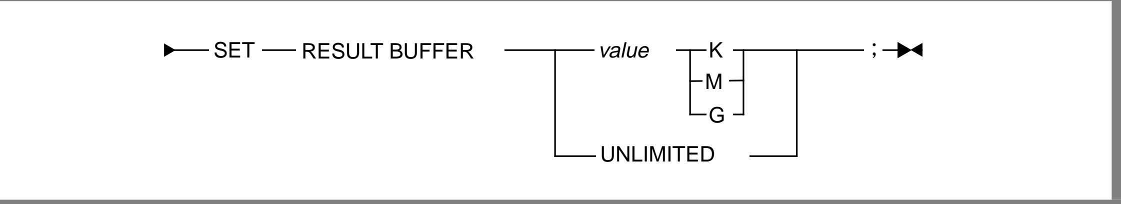
Usage Note

This command overrides the default report interval of one minute and the value specified by the ADMIN REPORT_INTERVAL parameter of the *rbw.config* file.

If the value `0` is supplied as the argument to this command, the session does not have a maximum interval between reports to the *rbwadmd* daemon process. The session simply waits until an event such as query completion occurs before sending a report.

SET RESULT BUFFER and SET RESULT BUFFER FULL ACTION

To specify the size of the buffer that holds query results until the client is ready to receive them, enter an SQL SET command with the following syntax:

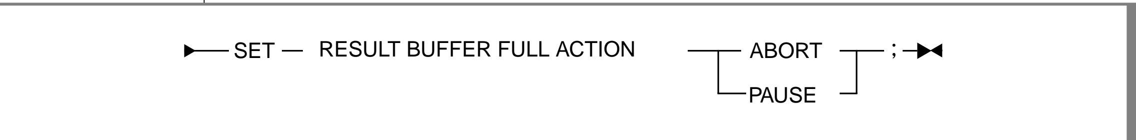


value Specifies an integer value, which must be followed by a *K* (kilobytes), an *M* (megabytes), or a *G* (gigabytes).

UNLIMITED Indicates that there is no limit on the amount buffered. Note that the buffer uses the same space allocated with the QUERY TEMPSPACE MAXSPILLSIZE parameter, so when the RESULT BUFFER parameter is set to unlimited, the buffer size is still limited by the QUERY TEMPSPACE MAXSPILLSIZE value.

Setting a value of 0 for the RESULT BUFFER parameter specifies that no results will be buffered.

To specify the behavior when the results buffer size specified with the SET RESULT BUFFER command is reached, enter a SET command with the following syntax:



The value ABORT indicates that the query will abort when the buffer size is reached. The value PAUSE indicates that when the buffer size is reached the query will pause until the client requests more data.

For a large result set, the read lock(s) on the table(s) remain until all of the results either leave the Red Brick Decision Server or are placed in the buffer. With client tools that require user input to receive more than a certain amount of data, the read locks will remain on the tables until all of the results are either delivered to the client or are placed in the buffer to wait for the client.

Example

The following SET commands specify a result buffer of 100 megabytes for the current session and force the query to abort when that buffer size is reached:

```
set result buffer 100M;
set result buffer full action abort;
```

SET ROWCOUNT

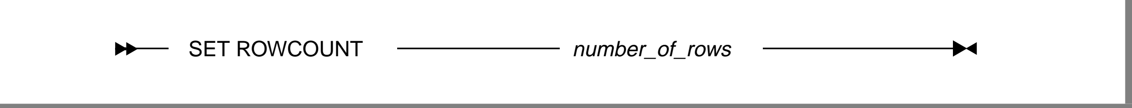
The SET ROWCOUNT command gives administrators some control over the execution of “runaway” queries issued by naive users of SQL. This command stops the execution of a query as soon as a specified number of rows has been retrieved. In this way, system resources are not wasted, but users can see at least a partial result set. An informational message is displayed after the result set, indicating that SET ROWCOUNT is in effect.

If a fully executed query happens to return the exact number of rows specified by the ROWCOUNT value, the query-termination message will still be displayed.

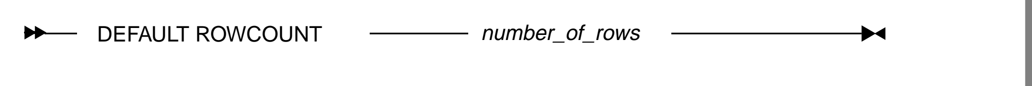
Syntax

You can set the row count for the current session by issuing a SET command. To enforce the row count for all server sessions, you can add a DEFAULT ROWCOUNT parameter to the configuration file (*rbw.config*).

The SET command syntax is as follows:



The syntax for the *rbw.config* file entry is as follows:



In both cases, the number of rows must be set to a positive number. The default setting is zero (0), which turns off the restriction on row retrieval. The current ROWCOUNT setting can be queried from the RBW_OPTIONS system table.

Example

In the following example, the query stops executing after returning only 10 rows.

```
set rowcount 10
select prod_name, dollars
from sales s join product p on s.prodkey = p.prodkey
    and s.classkey = p.classkey
    join period d on s.perkey = d.perkey
where year = 1998
    and month = 'JAN'
PROD_NAME              DOLLARS
Veracruzano               96.00
Veracruzano               17.25
Veracruzano               31.50
Veracruzano               40.00
Veracruzano               51.75
Veracruzano               69.00
Veracruzano              337.50
Veracruzano               69.00
Veracruzano               36.75
Veracruzano              135.00

** INFORMATION ** (1436) Query terminated because ROWCOUNT
number of rows have been fetched. Rows returned: 10.
```

SET ROWS_PER...TASK

This section describes three related SET commands:

- SET ROWS_PER_SCAN_TASK
- SET ROWS_PER_JOIN_TASK
- SET ROWS_PER_FETCH_TASK

These commands control the extent of parallelism in query processing by specifying the number of rows per parallel task. For information about the SET FORCE TASK commands, which behave as overrides to the ROWS_PER_TASK parameters and specify the number of parallel tasks to be used regardless of row count, refer to "SET FORCE TASKS" on page -18.

For general information about parallel-query processing, refer to the *Administrator's Guide*.

ROWS_PER_SCAN_TASK

The ROWS_PER_SCAN_TASK parameter sets a lower limit to the number of rows each scan process must return in order to justify its existence, thus limiting the number of parallel processes initiated for a relation scan. This limit affects queries that use no index but scan an entire table.

For detailed explanations and examples of the equations involved in computing values for this parameter, refer to the *Administrator's Guide*.

Syntax

The following syntax diagram shows how to construct a SET ROWS_PER_SCAN_TASK statement:

rows_per_process An integer in the range of 1 to 2^{31}. A higher value provides less parallelism in returning rows from the queried table, and a lower value, more parallelism. (Informix recommends you set this number to at least 5,000.)

The number of processes actually used is also bounded by the values set for the TOTALQUERYPROCS and QUERYPROCS parameters.

ROWS_PER_FETCH_TASK and ROWS_PER_JOIN_TASK

The ROWS_PER_FETCH_TASK and ROWS_PER_JOIN_TASK parameters determine how many parallel processes are used to process queries that use a STAR index. Because queries vary in the amount of work done during the index-probing phase and the row-data-processing phase, you can set different limits for each phase. For example, if your queries tend to require a lot of processing after each row is fetched (GROUP BY, SUM, MIN, and so on), you should assign fewer rows per process for the fetch phase than for the join phase so that more processes are used for the fetch phase.

For detailed explanations and examples of the equations involved in computing values for these parameters, refer to the *Administrator's Guide*.

Syntax

The following syntax diagram shows how to construct SET ROWS_PER_JOIN_TASK and SET ROWS_PER_FETCH_TASK statements:

```
►── SET ── ROWS_PER_JOIN_TASK   ── rows_per_process ──►◄
►── SET ── ROWS_PER_FETCH_TASK  ── rows_per_process ──►◄
```

rows_per_process Integers in the range of 1 to 2^{31}. A higher value provides less parallelism, and a lower value, more parallelism. In no case will the system run a query in parallel if the number of processes given by the preceding equations is less than 2.

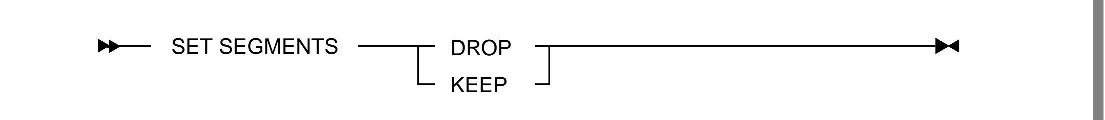
ROWS_PER_JOIN_TASK and ROWS_PER_FETCH _TASK should each be at least 5,000 to justify the use of parallel processes.

SET SEGMENTS

The SET SEGMENTS command specifies whether segments are dropped or retained when the table or index to which the segments are attached is dropped. This specification applies only to named segments. Default segments are always dropped.

Syntax

The following syntax diagram shows how to construct a SET SEGMENTS statement:



DROP Specifies that segments are dropped when the table or index to which the segments are attached is dropped.

KEEP Specifies that named segments remain available for re-use when the table or index to which the segments are attached is dropped. The physical storage units assigned to the segments remain intact and the segments can be attached to another table or index with the ALTER SEGMENT command.

Usage Note

This command overrides the default, which is either to drop named segments or as specified in the OPTION SEGMENTS parameter of the *rbw.config* file. All settings can be overridden by the DROP TABLE and DROP INDEX commands.

Example

The following statement specifies that when a table or index is dropped in the current session, the segment(s) attached to the table or index are also dropped. The segments cannot be reused with another table or index.

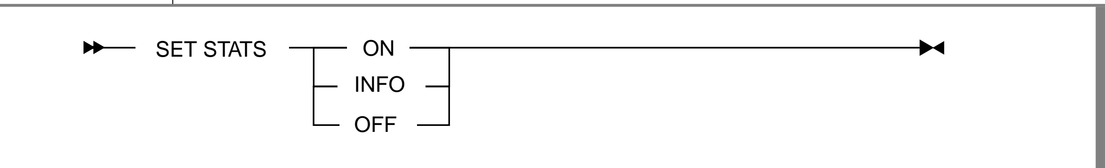
```
set segments drop
```

SET STATS

The SET STATS command turns on statistics reporting for the current session.

Syntax

The following syntax diagram shows how to construct a SET STATS statement:



ON Returns a summary of statistical information for each query. Statistics reporting varies from statement to statement and from platform to platform.

The following statistical information is typically supported:

- Number of data rows returned
- Elapsed time during statement processing
- CPU time spent on statement execution
- Logical I/O counts

INFO The INFO setting returns the same statistics as the ON setting, along with additional information about how the query was executed, such as query compilation time, the choice of query plan, which STAR index was used (if any), and whether the query was rewritten.

The IDs and operators referred to in the statistics messages correspond to information about query compilation and processing that can be retrieved with the EXPLAIN command. For a simple example of this command, refer to page 8-185 of this document. For detailed information about its output, refer to the *Administrator's Guide*.

An SQL query can contain any number of subqueries. To prevent the return of an excessive number of messages when complex queries are issued, the INFO_MESSAGE_LIMIT parameter defaults to a maximum of 1,000 messages returned per query. For more details about this parameter, see "SET INFO MESSAGE LIMIT" on page 9-31.

OFF Turns off statistics reporting. OFF is the default setting.

Example

To receive summary statistics for queries issued during the current session, enter:

```
set stats on
```

Statistics are then displayed in standard output after the result:

```
select promo_desc, sum(dollars)
from sales natural join promotion natural join period
where year = 2000
group by promo_desc
PROMO_DESC
No promotion                                       769209.70
Temporary price reduction                           14398.75
Aroma catalog coupon                                 8440.00
Monthly coffee special                               8420.45
Store display                                        6921.50
** STATISTICS ** (500) Time = 00:00:01.64 cp time, 00:00:01.49
time, Logical IO count=125
** INFORMATION ** (256) 5 rows returned.
```

If you issue the SET STATS INFO command before running the same query, more detailed statistics are displayed:

```
** STATISTICS ** (500) Compilation = 00:00:00.12 cp time,
00:00:00.12 time, Logical IO count=82
** STATISTICS ** (1458) CHOOSE PLAN (ID:1) chose Choice: 1.
** STATISTICS ** (1459) CHOOSE PLAN (ID:1) doing STARjoin on
1 tables.
** STATISTICS ** (1460) CHOOSE PLAN (ID:1) using Index
SALES_STAR_IDX of Table SALES for STARjoin.
** STATISTICS ** (1457) EXCHANGE (ID:5) used parallelism of 0.
** STATISTICS ** (1457) EXCHANGE (ID:9) used parallelism of 0.
** STATISTICS ** (500) Time = 00:00:01.29 cp time, 00:00:01.08
time, Logical IO count=125
** INFORMATION ** (256) 5 rows returned.
```

SET TEMPORARY SEGMENT STORAGE PATH

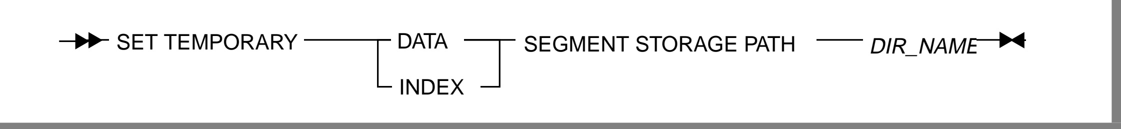
The SET TEMPORARY (DATA | INDEX) SEGMENT STORAGE PATH statement allows the user to specify the directory that stores the physical storage units (PSUs) of the default temporary data or temporary index segments. If this directory is not specified, the PSUs of the temporary segments are stored in the database directory.

If a large number of temporary tables or several large temporary tables are created simultaneously, the temporary storage segment might run out of storage space.

This command also exists as the OPTION TEMPORARY_DATA_SEGMENT and OPTION TEMPORARY_INDEX_SEGMENT parameters in the rbw.config file.

Syntax

The following syntax diagram shows how to construct a SET TEMPORARY (DATA | INDEX) SEGMENT STORAGE PATH statement:



dir_name Specifies the full pathname (dir_name) of the directory in which all default row data segments or all default index segments are to be stored. If a default directory is not specified, all default segments are stored in the database directory, as defined in the rbw.config file.

SET TRANSACTION ISOLATION LEVEL

The SET TRANSACTION ISOLATION LEVEL command controls the type of read locks used by an SQL statement that reads one table and changes another table, where no key relationships occur between the two tables. This command applies to versioned databases only.

Syntax

The following syntax diagram shows how to construct a SET TRANSACTION ISOLATION LEVEL statement:

SERIALIZABLE Specifies that an RD (Read Data) lock is used for a transaction that reads one table to modify another table. The RD lock does not permit simultaneous versioning modification on a table being read by another versioning transaction; the later transaction waits for the earlier one to complete before it can read the table. This is the most restrictive mode.

REPEATABLE READ Specifies that an RO (Read Only) lock is used. The RO lock allows a versioning modification of the table concurrent with a read operation. This allows a transaction to read an older version of a table which is currently being modified in order to modify another table (for example, an INSERT INTO...SELECT statement). This mode is less restricted than SERIALIZABLE.

This command also exists as the OPTION TRANSACTION_ISOLATION_LEVEL parameter in the *rbw.config* file. For more information about this parameter and examples of the differences between the two settings, refer to the *Administrator's Guide*.

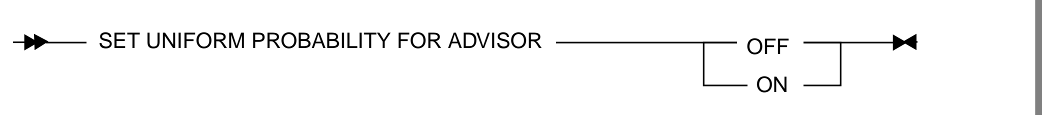
SET UNIFORM PROBABILITY FOR ADVISOR

The SET UNIFORM PROBABILITY FOR ADVISOR command determines whether the log file is scanned in order to compute the reference count for each view when the RBW_PRECOMPVIEW_UTILIZATION and RBW_PRECOMPVIEW_CANDIDATES Advisor system tables are queried. When it is set to ON, it is assumed that all of the views on a base table are referenced the same number of times. The default setting is OFF.

This command also exists as the OPTION UNIFORM_PROBABILITY_FOR_ADVISOR parameter in the rbw.config file.

Syntax

The following syntax diagram shows how to construct a SET UNIFORM PROBABILITY FOR ADVISOR command:



For more information about the Advisor and precomputed views, refer to the *Informix Vista User's Guide*.

SET USE INVALID PRECOMPUTED VIEWS

The SET USE INVALID PRECOMPUTED VIEWS ON command allows all precomputed views to be used for query rewriting regardless of their validity.

When this command is set to OFF, views must be marked valid in order to be considered for use in query rewrites or to be included in Advisor queries of the RBW_PRECOMPVIEW_UTILIZATION table.

Syntax

The following syntax diagram shows how to construct a SET USE INVALID PRECOMPUTED VIEWS statement:

```
SET USE INVALID PRECOMPUTED VIEWS --+-- OFF --+--
                                    +-- ON  --+
```

This command also exists as the OPTION USE_ INVALID_PRECOMPUTED _VIEWS parameter.

For more information about the Advisor and precomputed views, refer to the *Informix Vista User's Guide*.

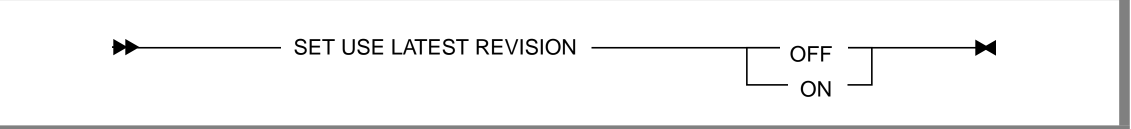
SET USE LATEST REVISION

The SET USE LATEST REVISION command specifies whether a RISQL session should see the current revision or the default query revision when a query revision has been set.

If no query revision has been set (by issuing an ALTER DATABASE FREEZE QUERY REVISION command) or if the query revision was turned off using the ALTER DATABASE UNFREEZE QUERY REVISION command, the session uses the latest revision regardless of the value chosen in the SET USE LATEST REVISION statement.

Syntax

The following syntax diagram shows how to construct a SET USE LATEST REVISION statement:



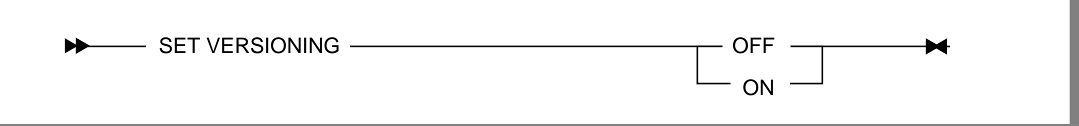
For more information about versioned databases, refer to the *Administrator's Guide*.

SET VERSIONING

The SET VERSIONING command determines whether database transactions are run as blocking transactions or versioning transactions.

Syntax

The following syntax diagram shows how to construct a SET VERSIONING statement:



OFF Specifies that statements are to be run as blocking transactions.

ON Specifies that statements are to be run as versioning transactions.

If the connected database does not have a version log, then a SET VERSIONING ON command causes transactions that modify the database (INSERT, UPDATE, and DELETE statements) to fail with an error.

If the connected database has a version log but versioning is currently stopped (ALTER DATABASE STOP VERSIONING), transactions that modify the database (INSERT, UPDATE, and DELETE statements) that would be run as versioning transactions fail with an error.

The SET VERSIONING command does not have an effect on DML operations for models and temporary tables; those operations are always performed as non-versioned (blocking) transactions.

This command also exists as the OPTION VERSIONING parameter in the *rbw.config* file. For more information about this command, refer to the *Administrator's Guide*.

Appendix

A

Syntax Summary

This appendix summarizes the SQL commands and RISQL extensions that are described in detail in Chapters 8 and 9; the syntax summaries are presented here for quick reference.

The summaries are grouped into five logical sections:

- Database Control Commands
- Data Definition Commands
- Data Manipulation Commands
- Miscellaneous Commands
- SET Commands

Database Control Commands

The commands listed in this section perform the following tasks:

- Control logging and accounting
- Control user activity
- Control user priority
- Create database usernames and passwords
- Grant and revoke database authorizations
- Grant and revoke privileges defined on database objects

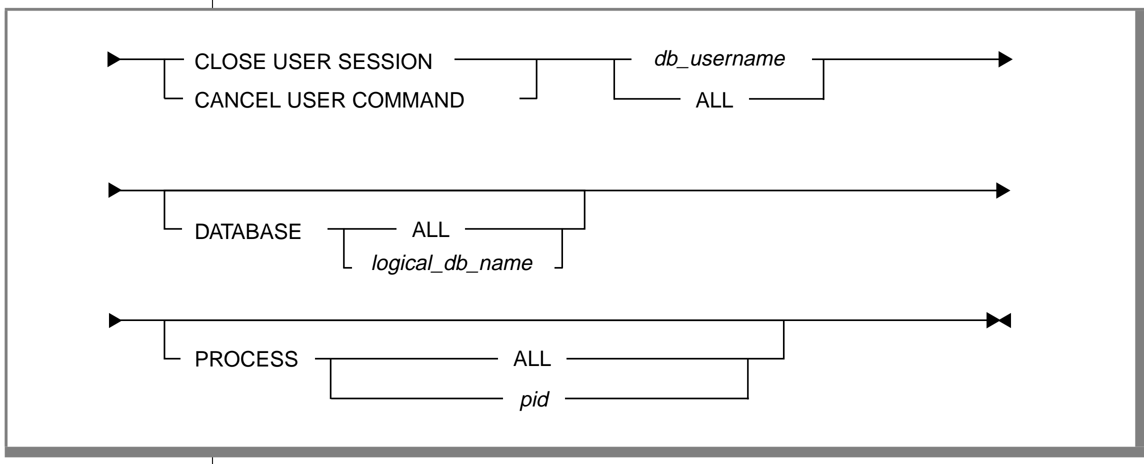
ALTER DATABASE

```
ALTER DATABASE ─┬─ CREATE BACKUP DATA IN segment_name ─┬─
                └─ DROP BACKUP DATA ───────────────────┘
```

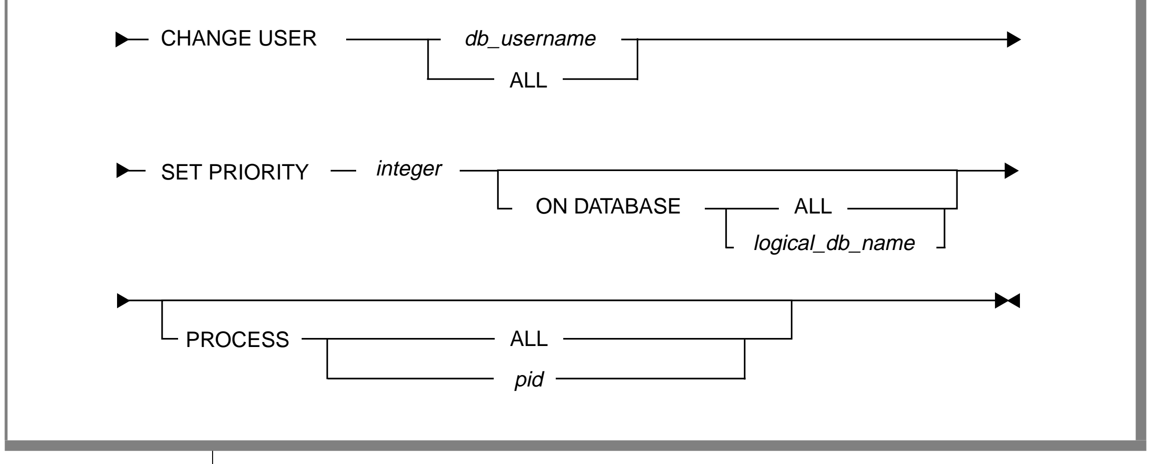
ALTER SYSTEM

```
ALTER SYSTEM ─┬─ RESET STATISTICS ─┬──────────────────────────────────────┬─
              │                    └─ DATABASE ─┬─ ALL ─────────────┬─────┘
              │                                 └─ logical_db_name ─┘
              ├─ QUIESCE ─┬──────────────────────────────────────┬─
              │           └─ DATABASE ─┬─ ALL ─────────────┬─────┘
              │                        └─ logical_db_name ─┘
              ├─ RESUME ─┬──────────────────────────────────────┬─
              │          └─ DATABASE ─┬─ ALL ─────────────┬─────┘
              │                       └─ logical_db_name ─┘
              ├─ TERMINATE ─┬─ ADMIN ──────────┬─ DAEMON ─
              │             └─ ADMINISTRATION ─┘
              ├─ alter_user_activity ─
              ├─ alter_user_priority ─
              ├─ alter_logging ─
              └─ alter_accounting ─
```

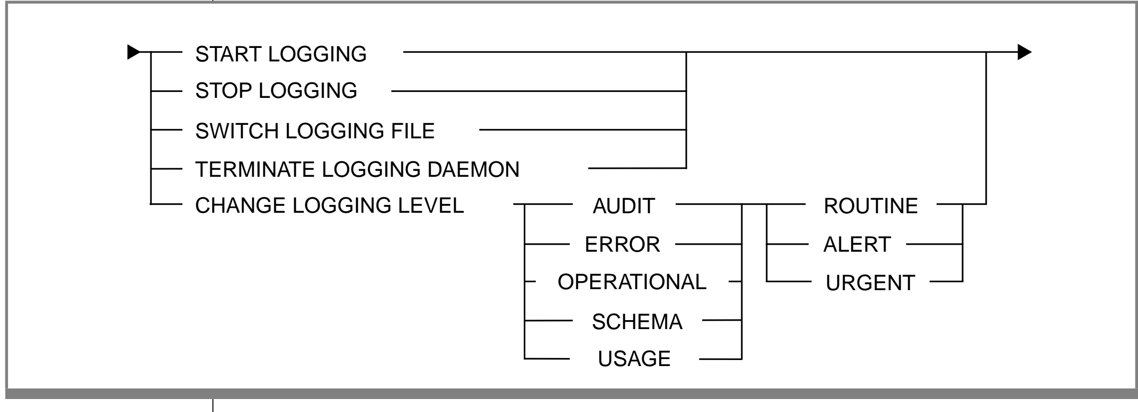
alter_user_activity specification



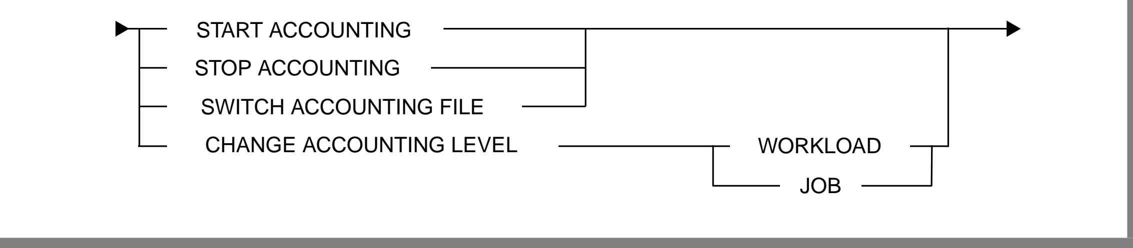
alter_user_priority specification



alter_logging specification

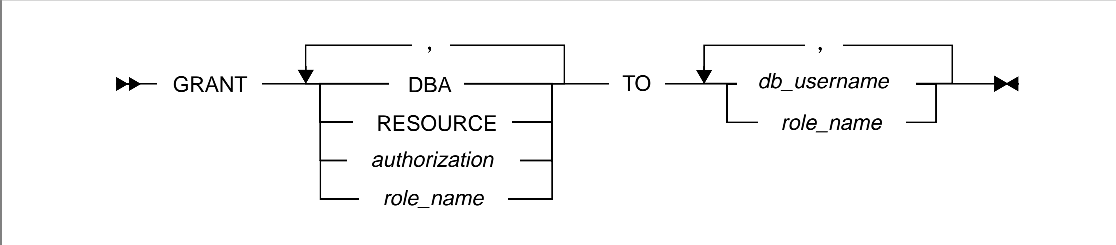


alter_accounting specification

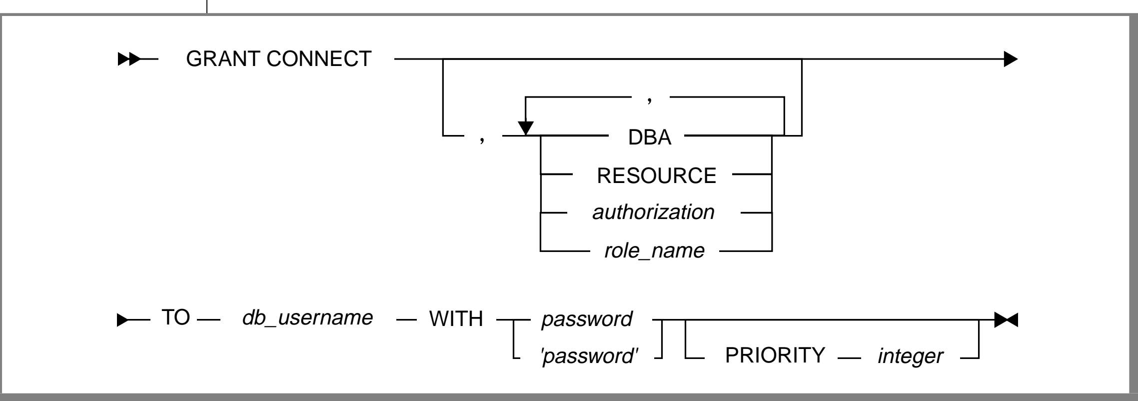


ALTER USER

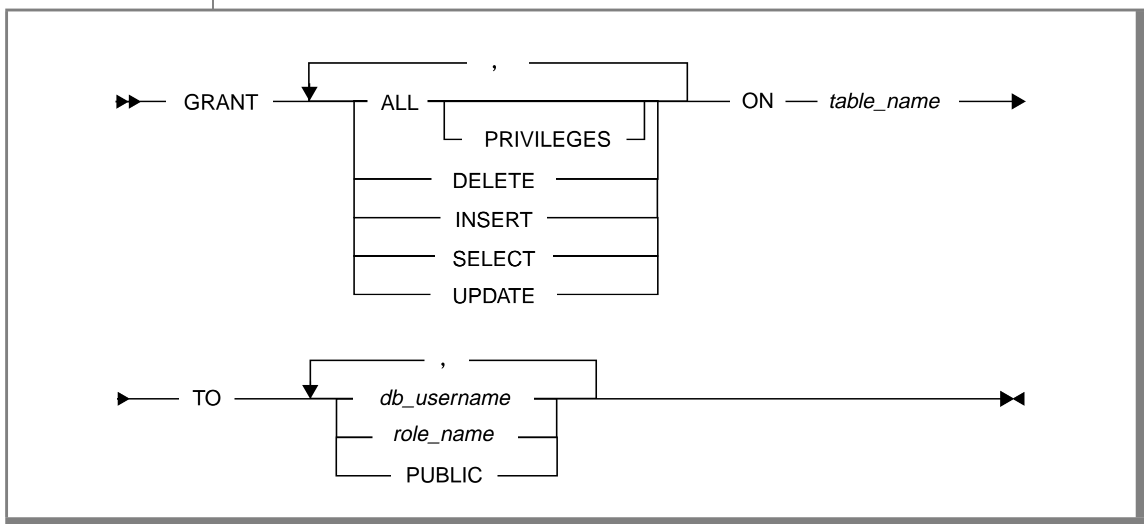
GRANT Authorization and Role



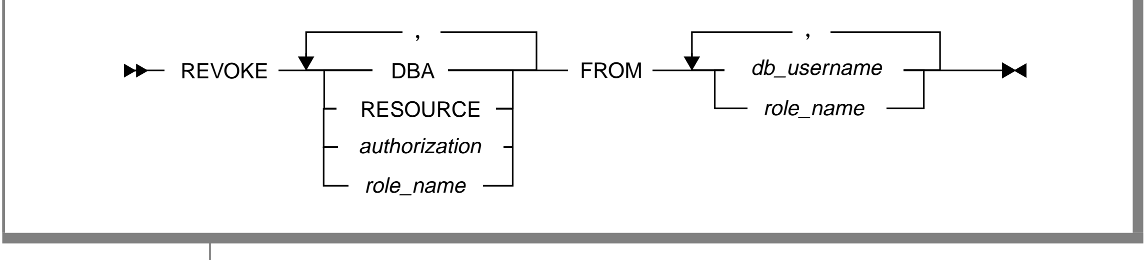
GRANT CONNECT



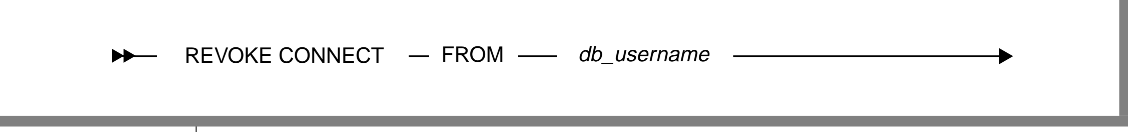
GRANT Privilege



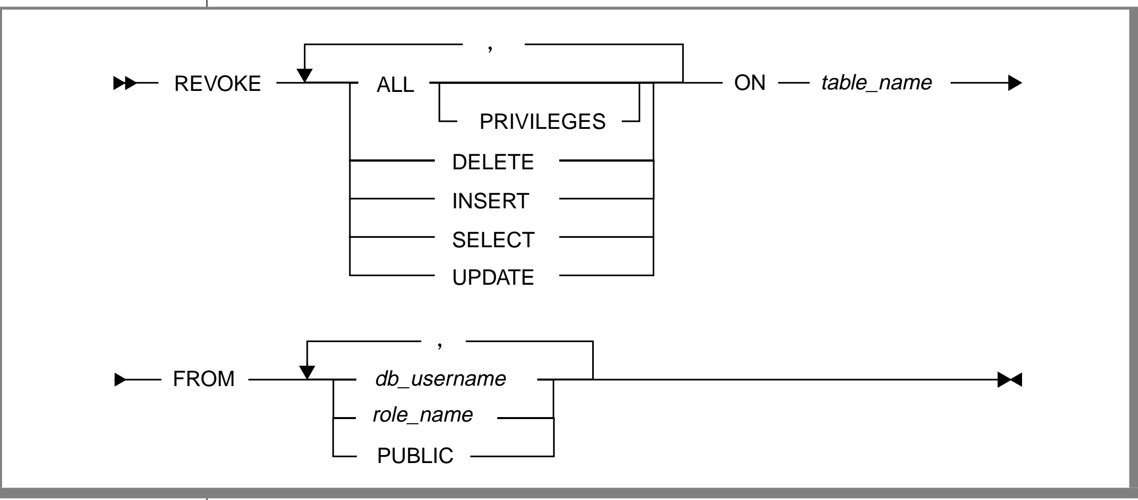
REVOKE Authorization and Role



REVOKE CONNECT



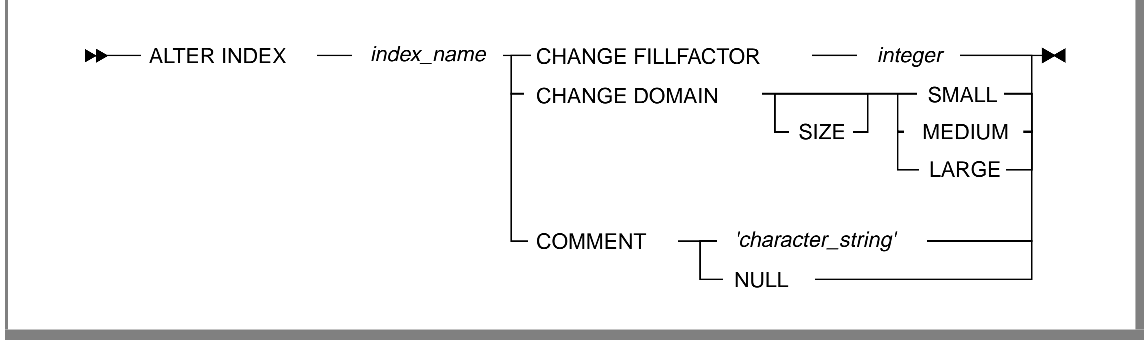
REVOKE Privilege



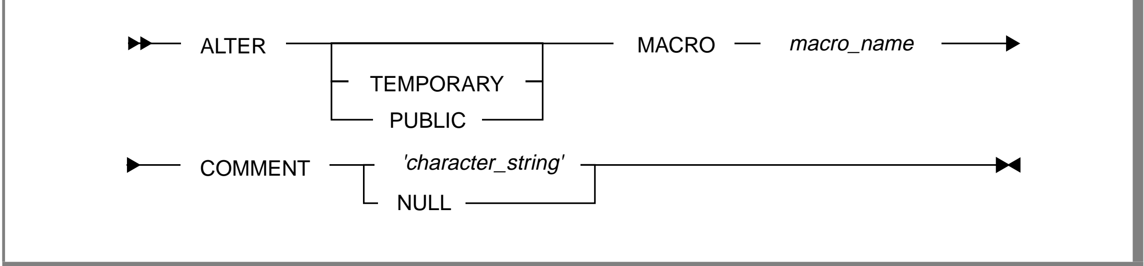
Data Definition Commands

The following commands are used to alter, create, and drop the following database objects: indexes, macros, roles, segments, synonyms, tables and views.

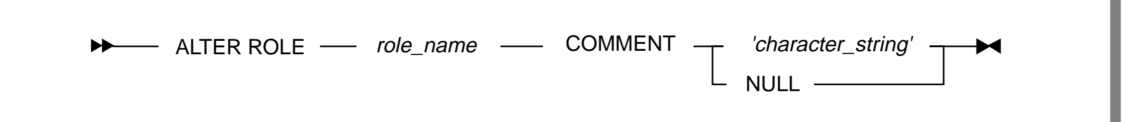
ALTER INDEX



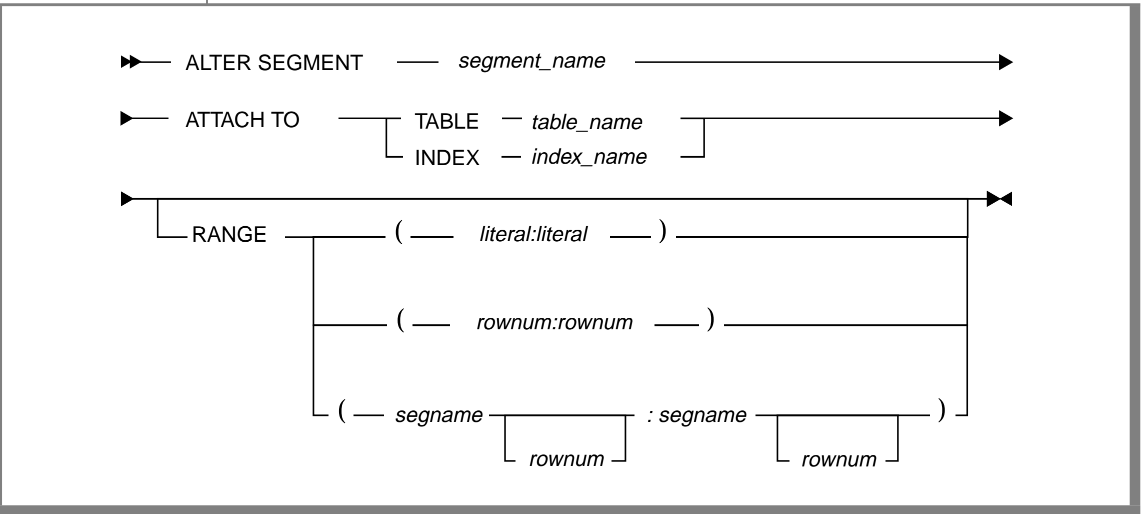
ALTER MACRO



ALTER ROLE

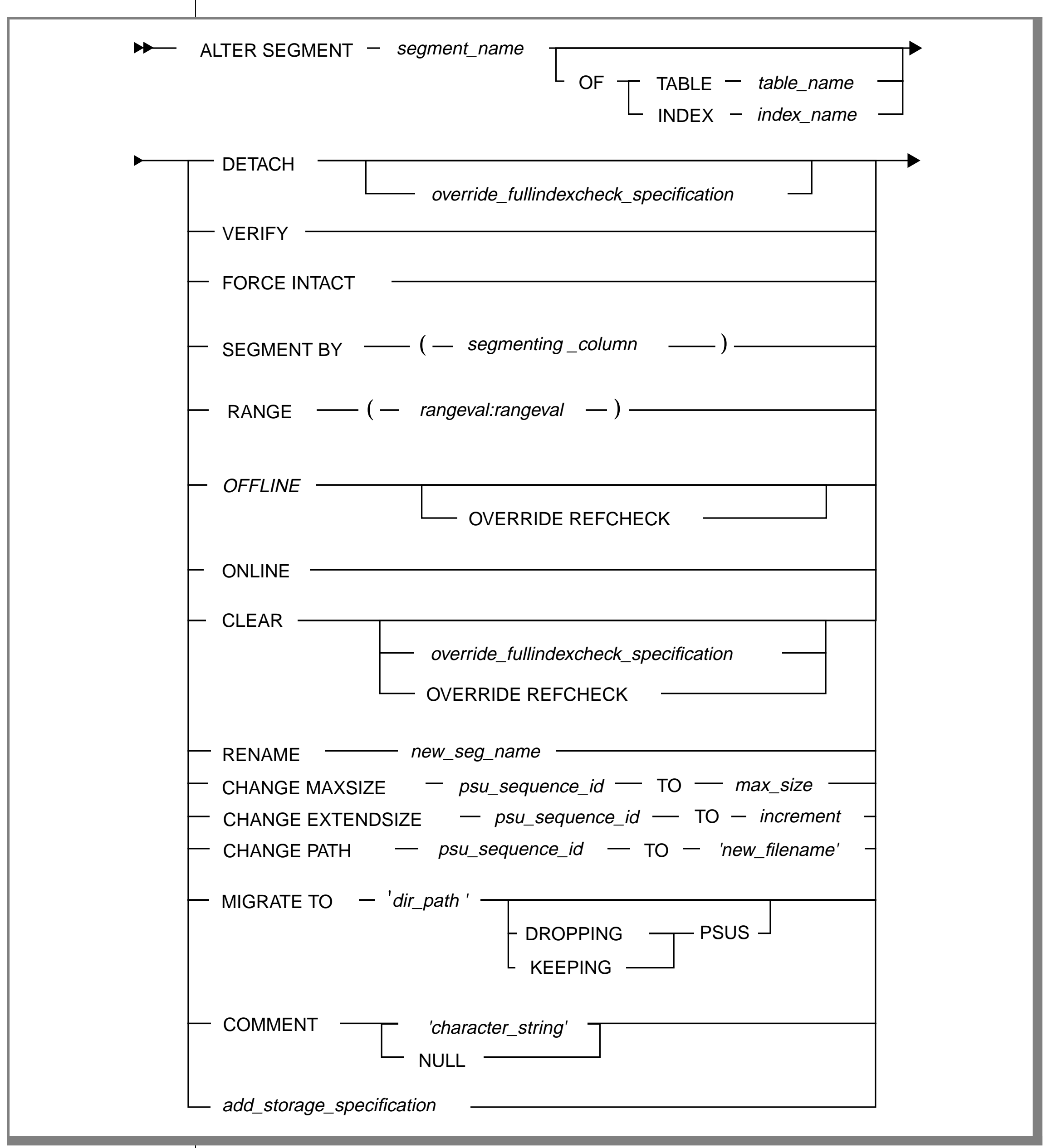


ALTER SEGMENT—Attach Clause

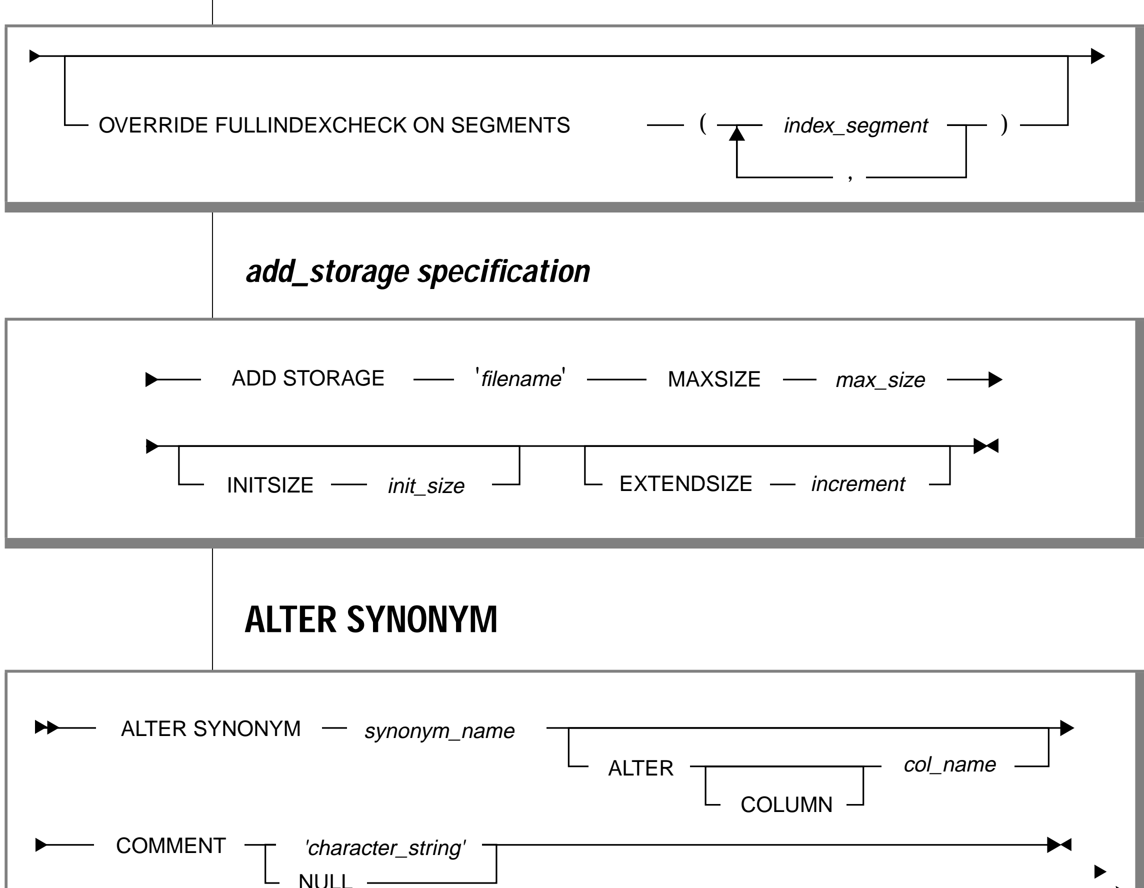


In all the variations of the RANGE specification, MIN and MAX are also valid values on the left and right sides of the colon, respectively.

ALTER SEGMENT—Other Clauses



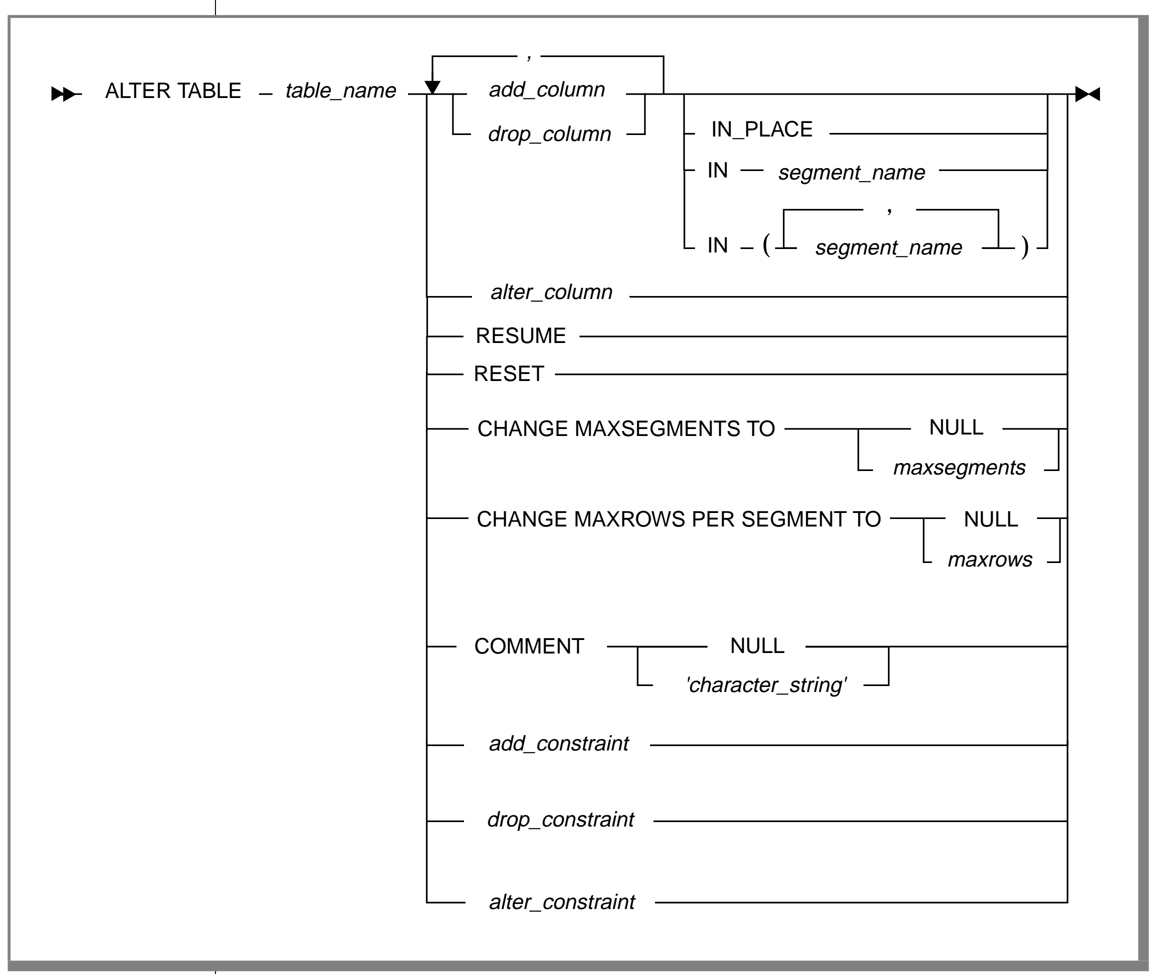
override_fullindexcheck_specification



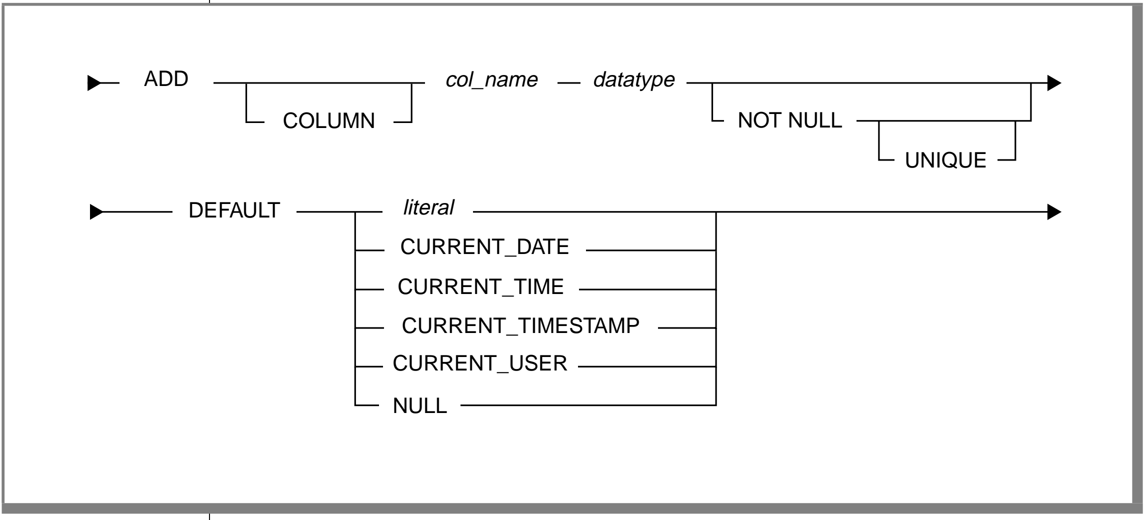
add_storage specification

ALTER SYNONYM

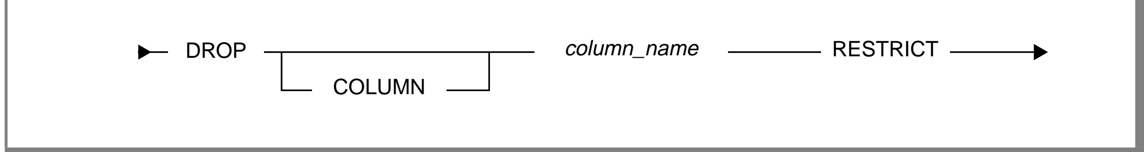
ALTER TABLE



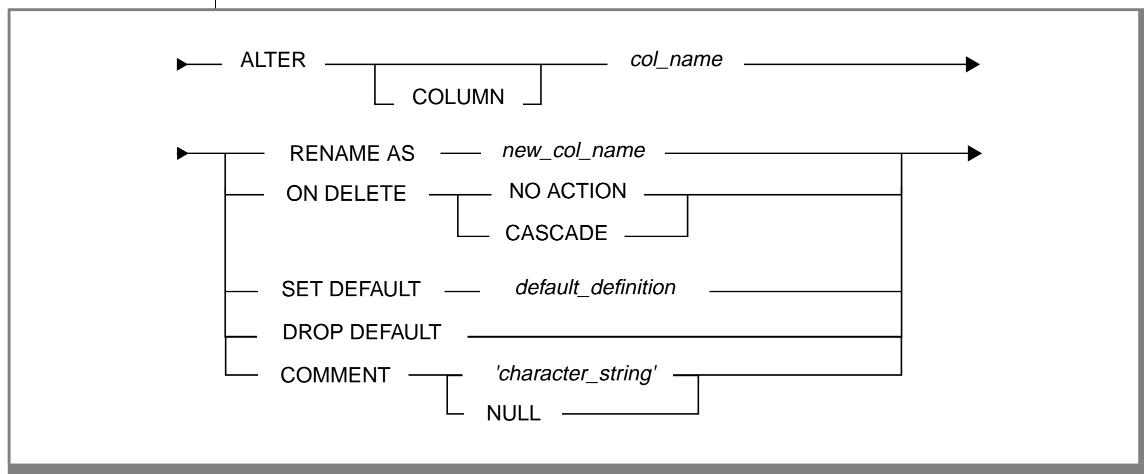
add_column specification



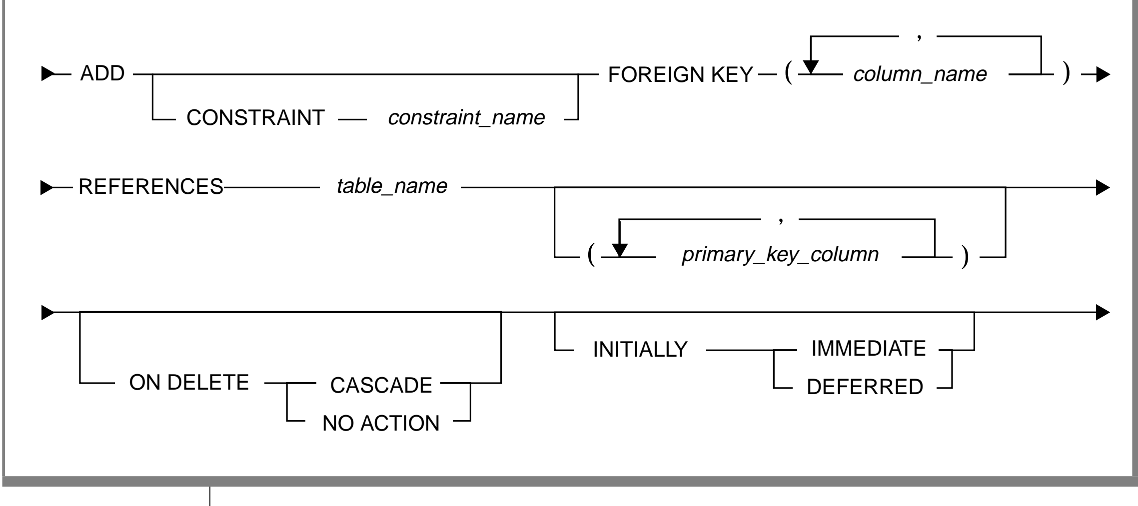
drop_column specification



alter_column specification



add_constraint specification



drop_constraint specification

DROP CONSTRAINT — *constraint_name*

alter_constraint specification

ALTER CONSTRAINT — *constraint_name*

REFERENCES — *referenced_table_name*

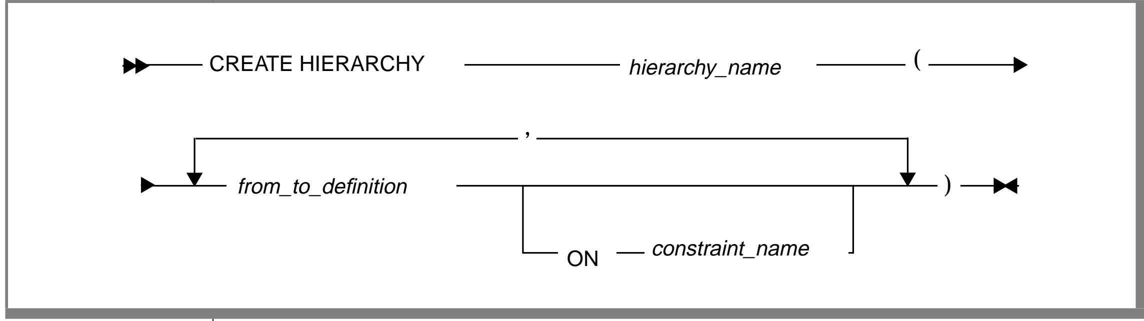
ON DELETE — CASCADE | NO ACTION

ALTER VIEW

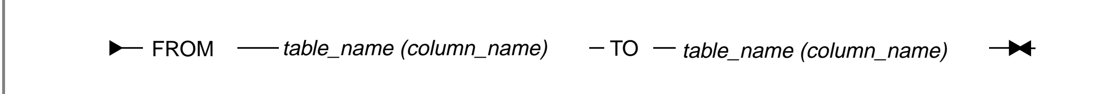
ALTER VIEW — *view_name* [ALTER [COLUMN] *col_name*]

COMMENT — *'character_string'* | NULL

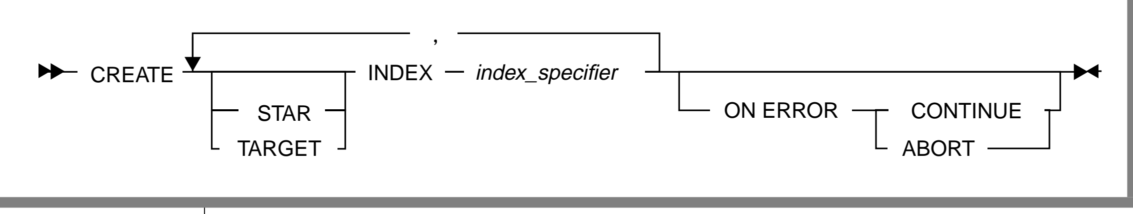
CREATE HIERARCHY



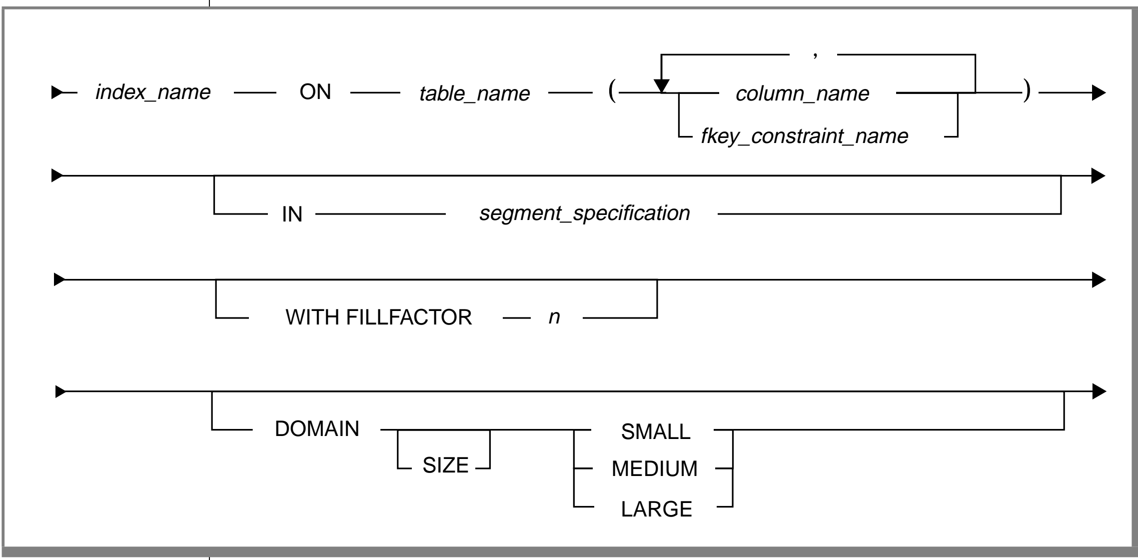
from_to_definition clause



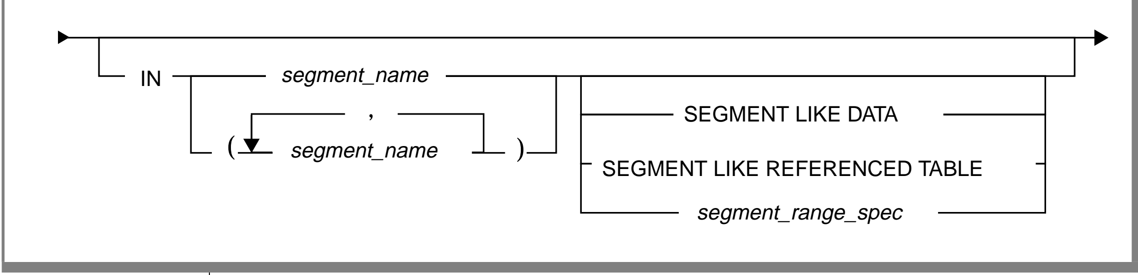
CREATE INDEX



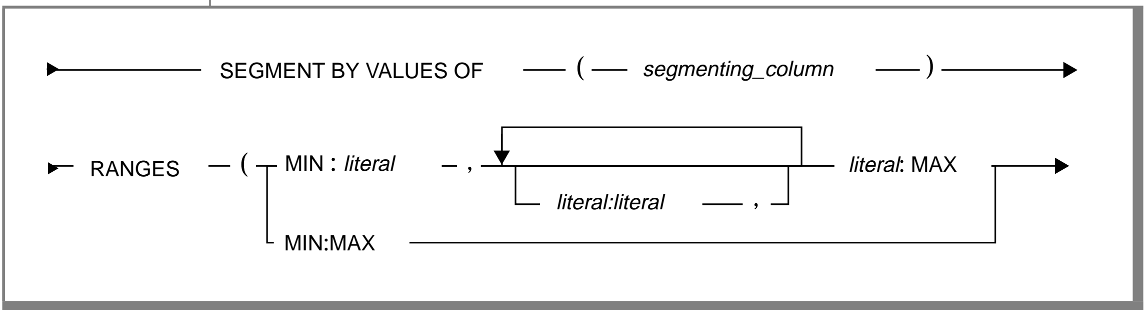
index_specifier



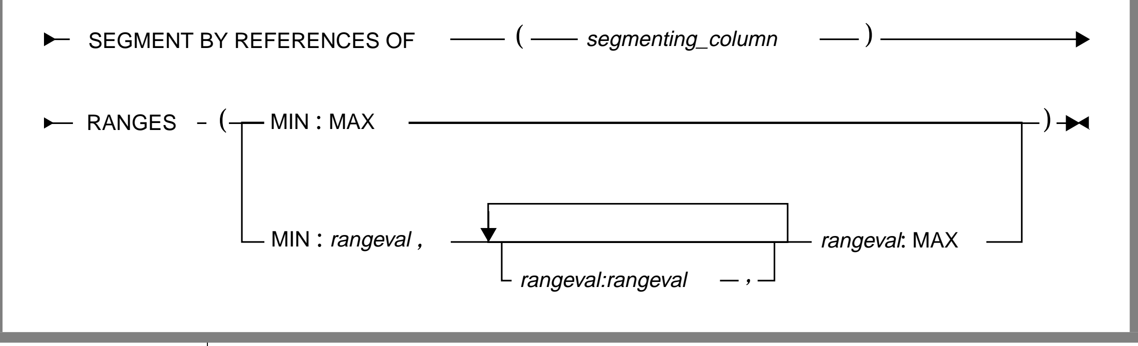
segment_specification



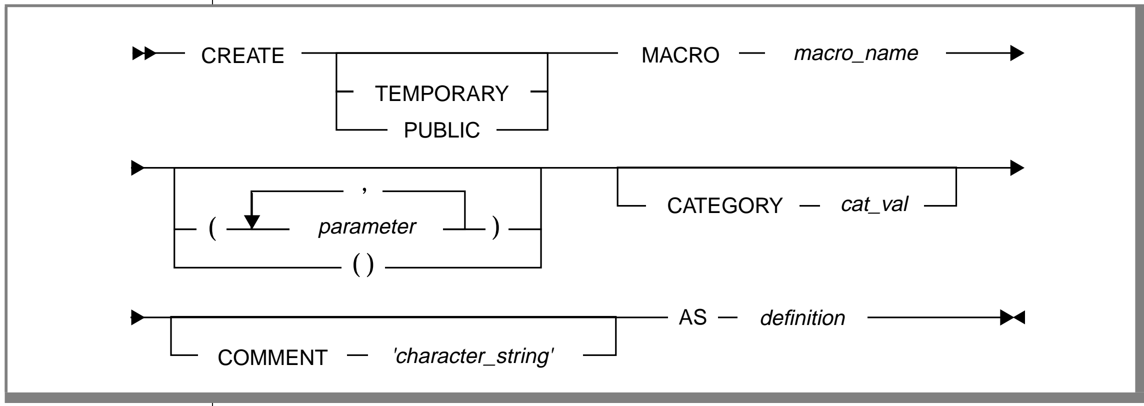
segment_range_spec—B-TREE and TARGET indexes



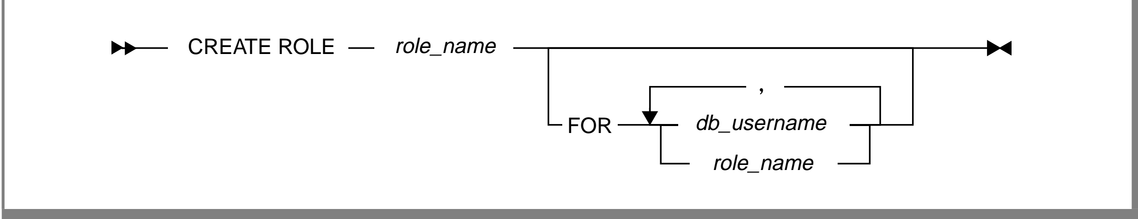
segment_range_spec—STAR indexes



CREATE MACRO



CREATE ROLE



CREATE SEGMENT

```
                                              ┌──────── , ────────┐
                                              ↓                   │
►► CREATE SEGMENT ── segment_name ─────────────── storage_specification ─┴──►◄
```

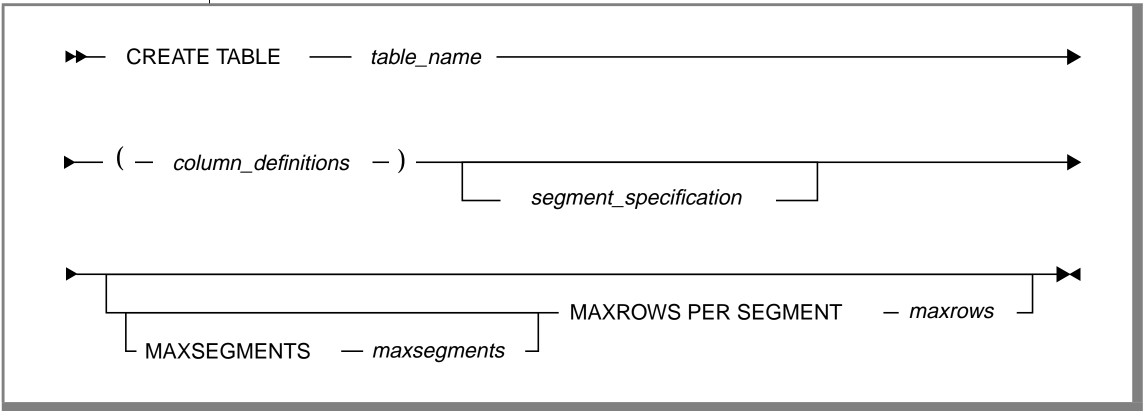
storage_specification

```
►── STORAGE ── 'filename' ──── MAXSIZE ── max_size ───►

►─┬──────────────────────────┬─┬──────────────────────────────┬─►◄
  └─ INITSIZE ── init_size ──┘ └─ EXTENDSIZE ── increment ─────┘
```

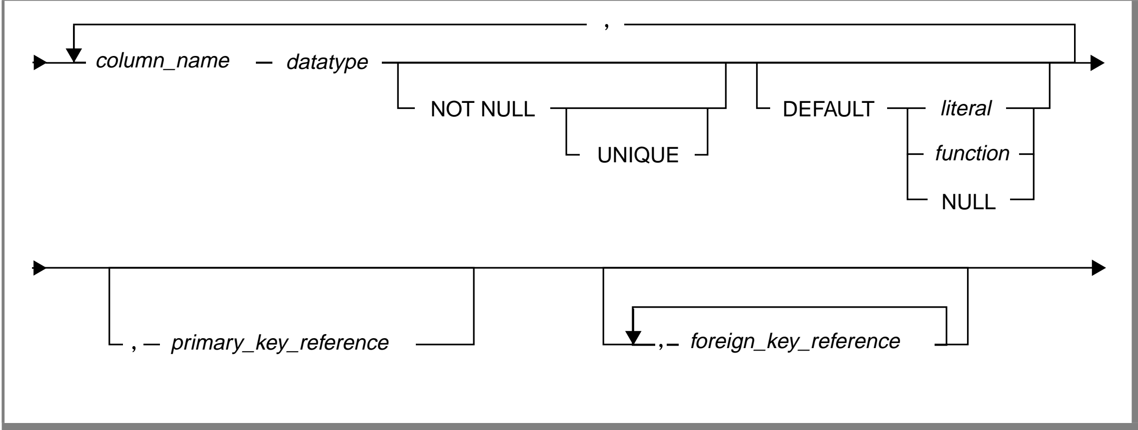
CREATE SYNONYM

```
►► CREATE SYNONYM ── synonym_name ── FOR ── table_name ──►◄
```

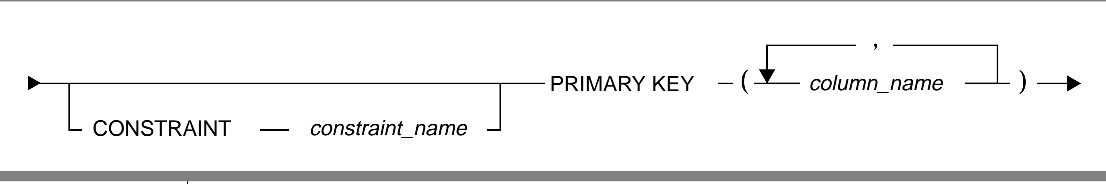
CREATE TABLE



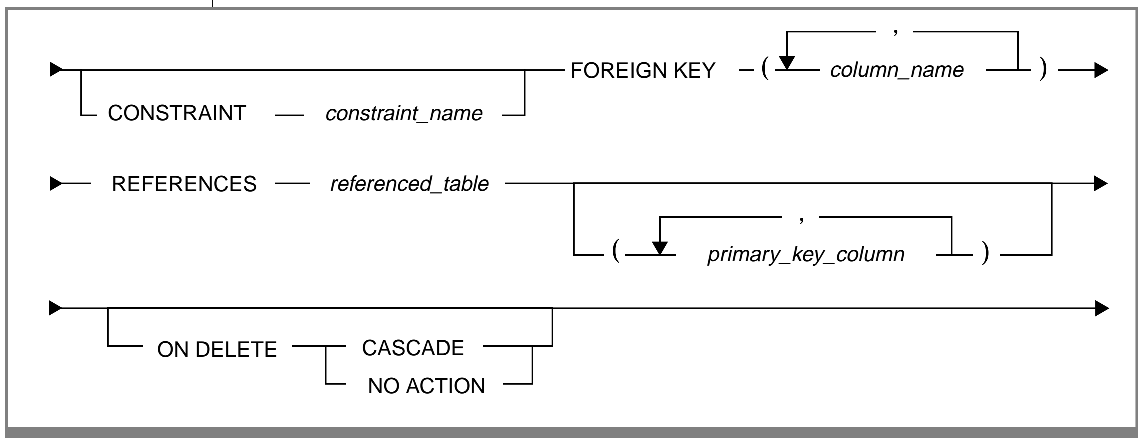
column_definitions



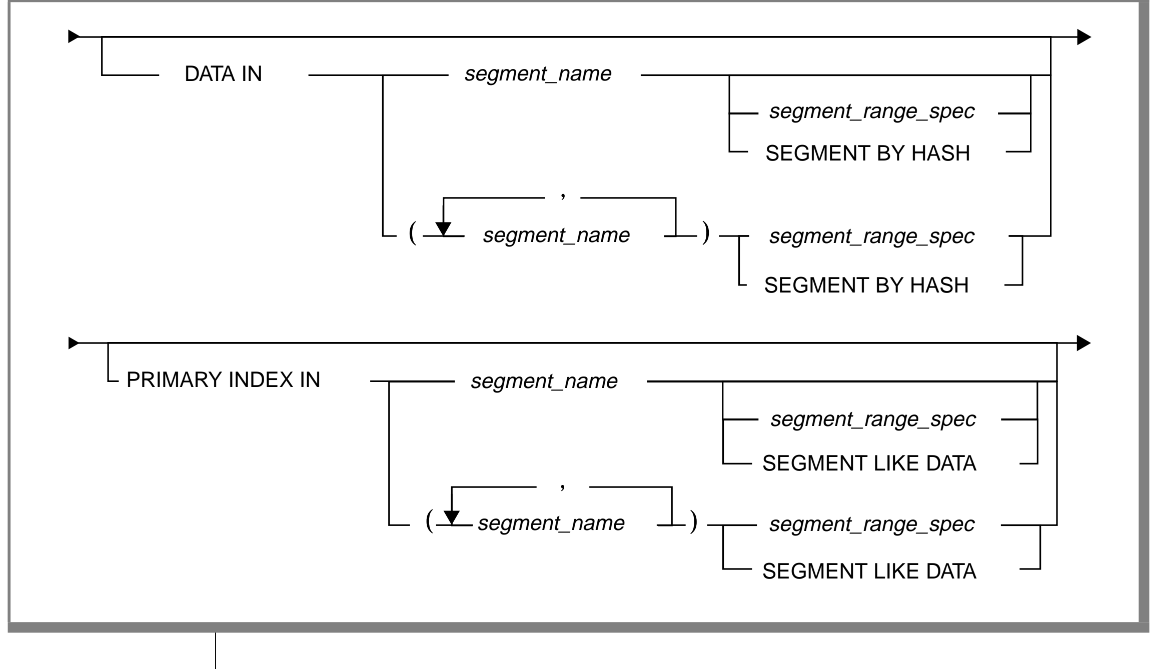
primary_key_reference



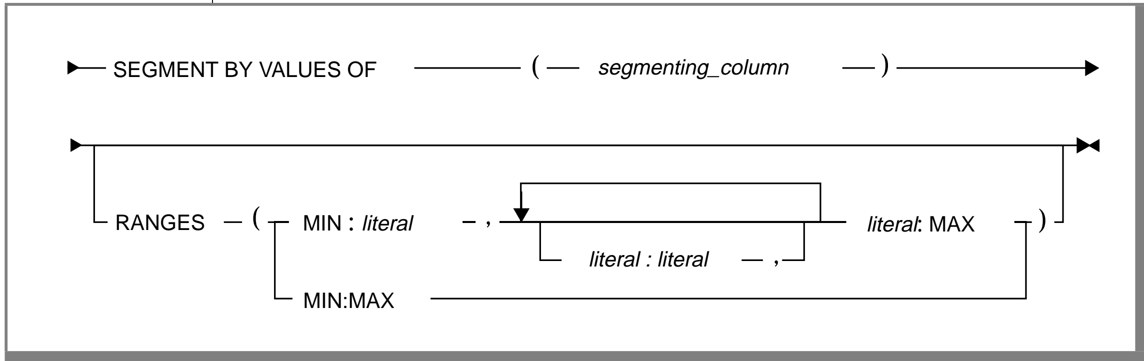
foreign_key_reference



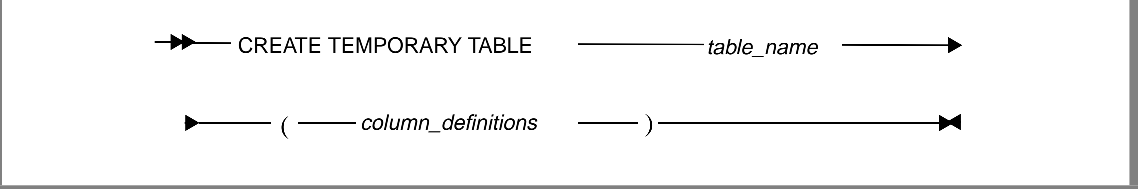
segment_specification



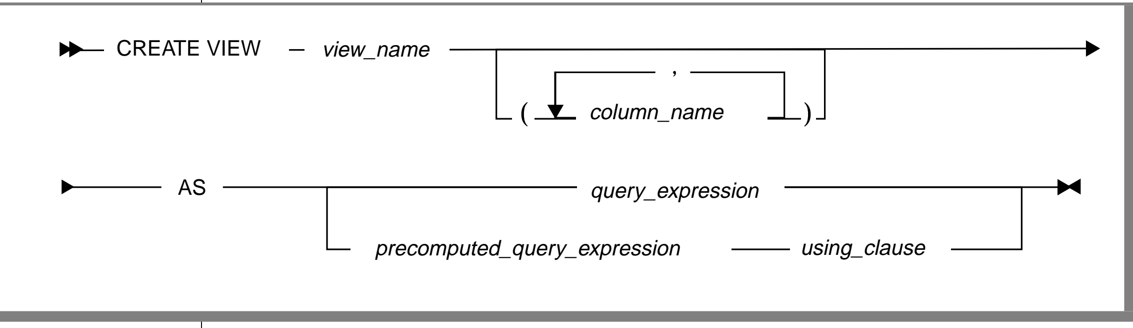
segment_range_spec

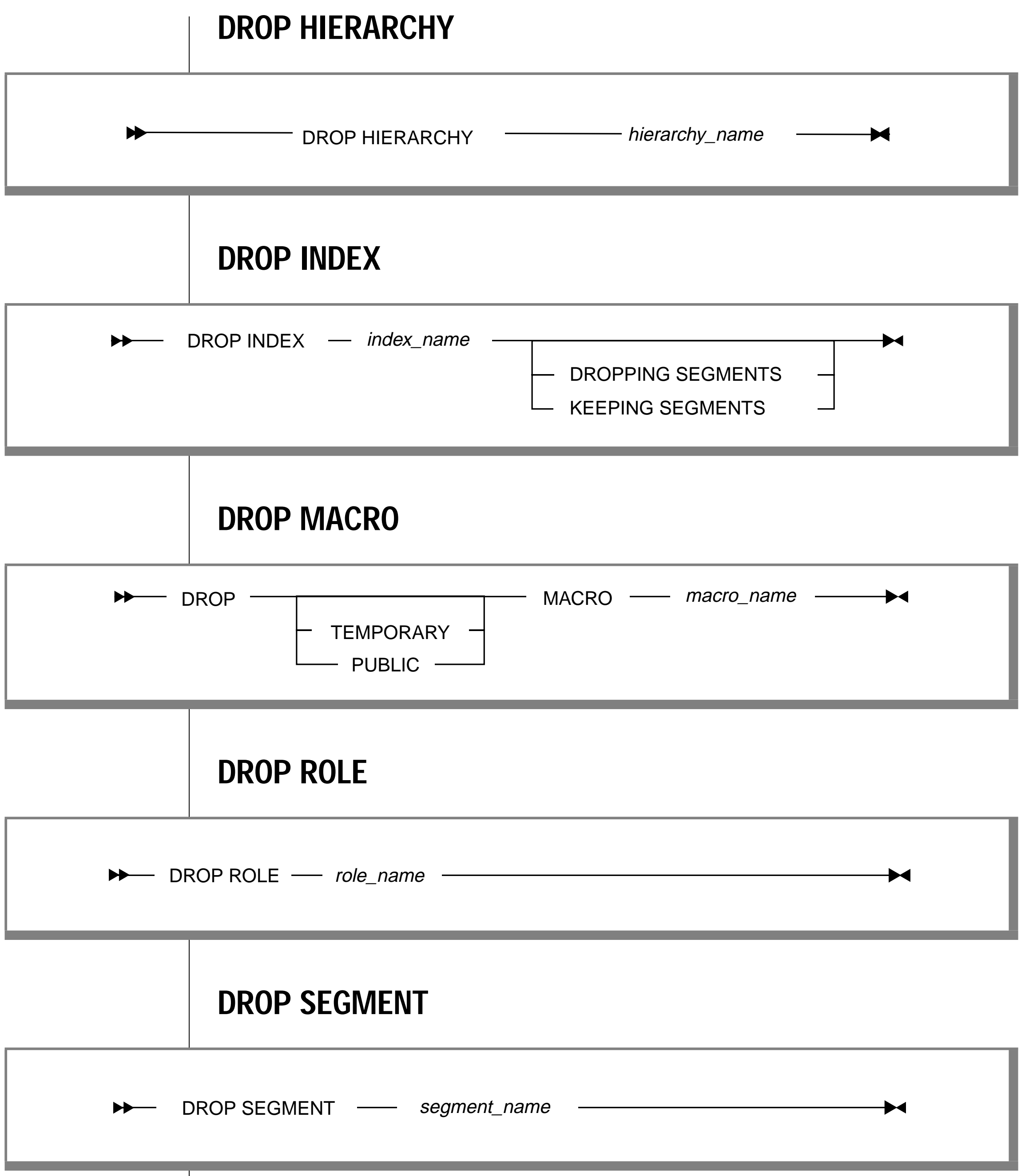


CREATE TEMPORARY TABLE



CREATE VIEW




DROP HIERARCHY
DROP HIERARCHY
hierarchy_name
DROP INDEX
DROP INDEX
index_name
DROPPING SEGMENTS
KEEPING SEGMENTS
DROP MACRO
DROP
TEMPORARY
PUBLIC
MACRO
macro_name
DROP ROLE
DROP ROLE
role_name
DROP SEGMENT
DROP SEGMENT
segment_name

DROP SYNONYM

```
►►── DROP  SYNONYM ── synonym_name ──────────────────►◄
```

DROP TABLE

```
►►── DROP  TABLE ──┬──────────┬── table_name ──┬──────────────────────┬──►◄
                   └ creator. ┘                ├─ DROPPING SEGMENTS ──┤
                                               └─ KEEPING SEGMENTS ───┘
```

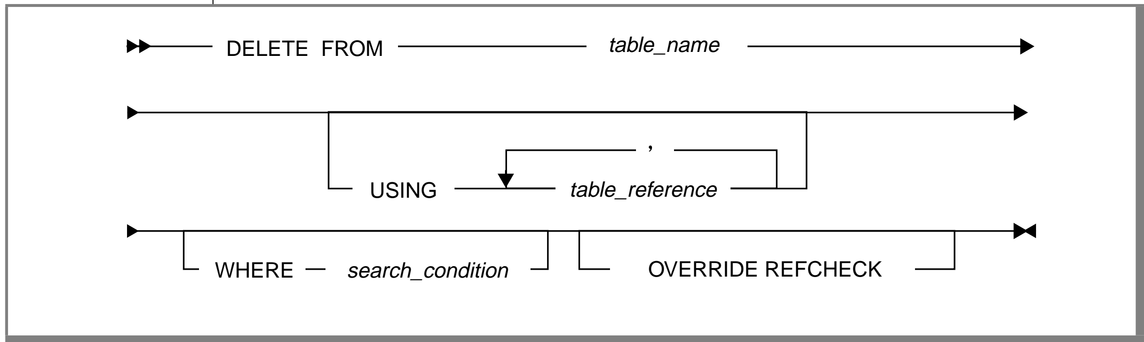
DROP VIEW

```
►►── DROP VIEW ── view_name ──────────────────►◄
```

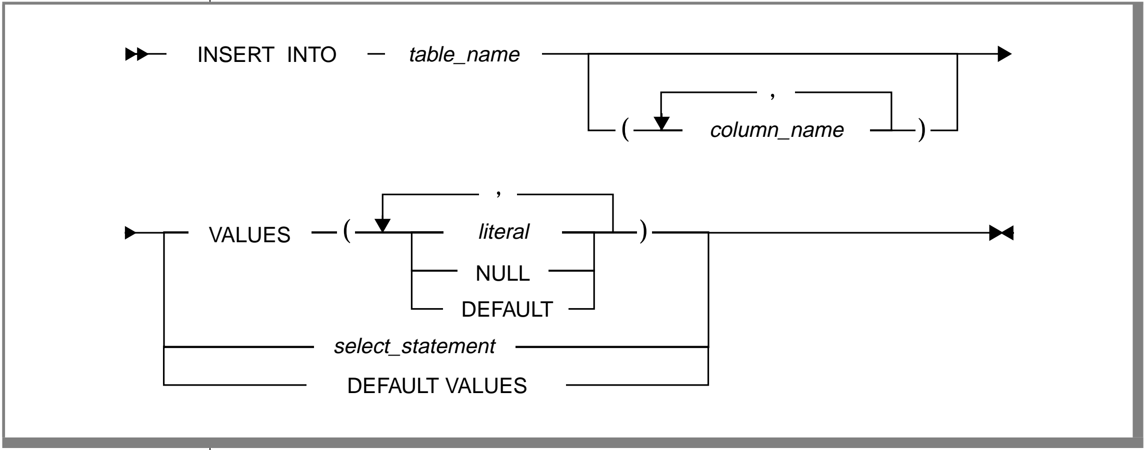
Data Manipulation Commands

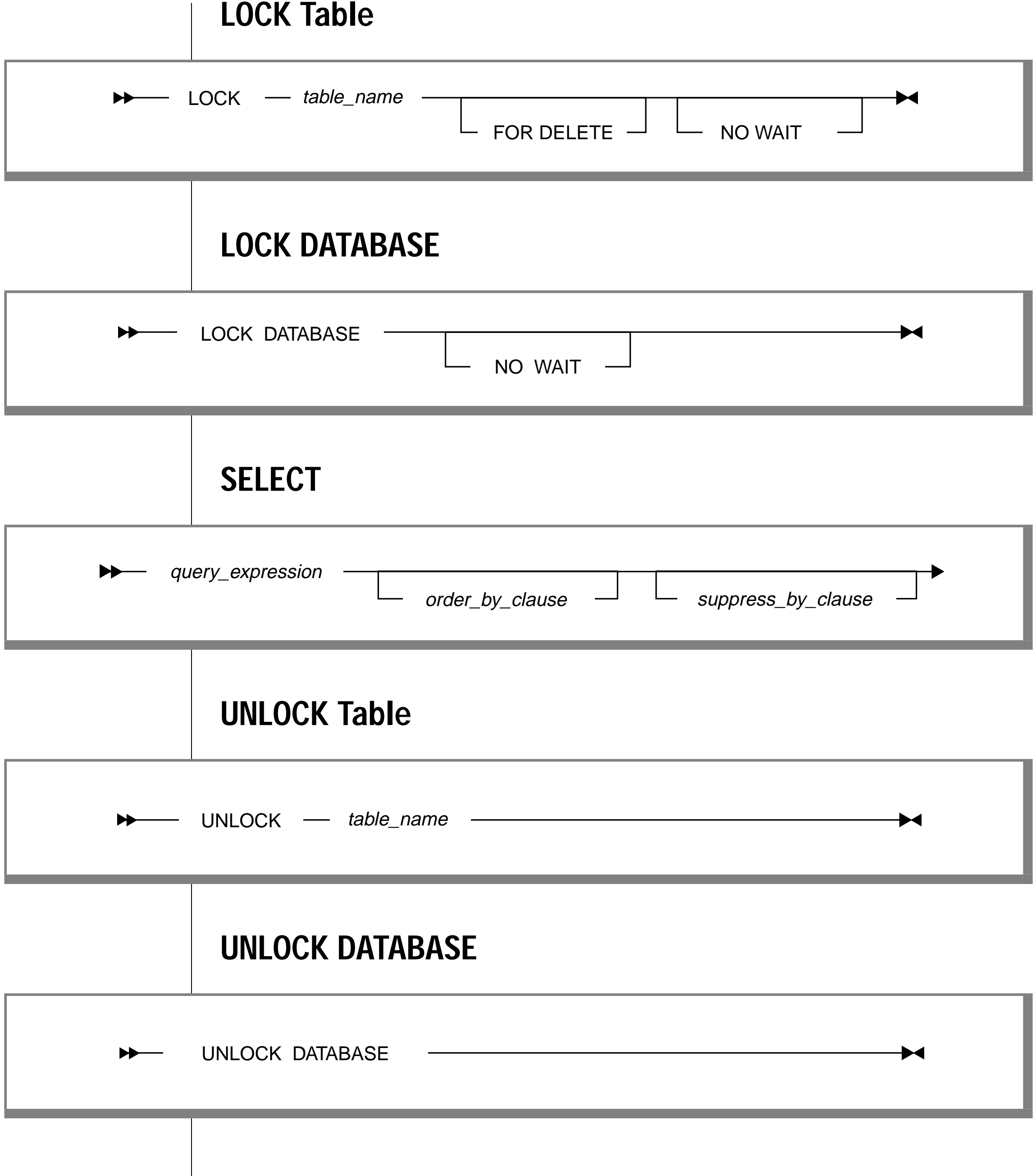
The following commands modify rows of data in database tables.

DELETE



INSERT




LOCK Table
LOCK
table_name
FOR DELETE
NO WAIT
LOCK DATABASE
LOCK DATABASE
NO WAIT
SELECT
query_expression
order_by_clause
suppress_by_clause
UNLOCK Table
UNLOCK
table_name
UNLOCK DATABASE
UNLOCK DATABASE

UPDATE

UPDATE table_name SET column_name = expression | NULL [, ...] [FROM table_reference [, ...]] [WHERE search_condition]

Miscellaneous Commands

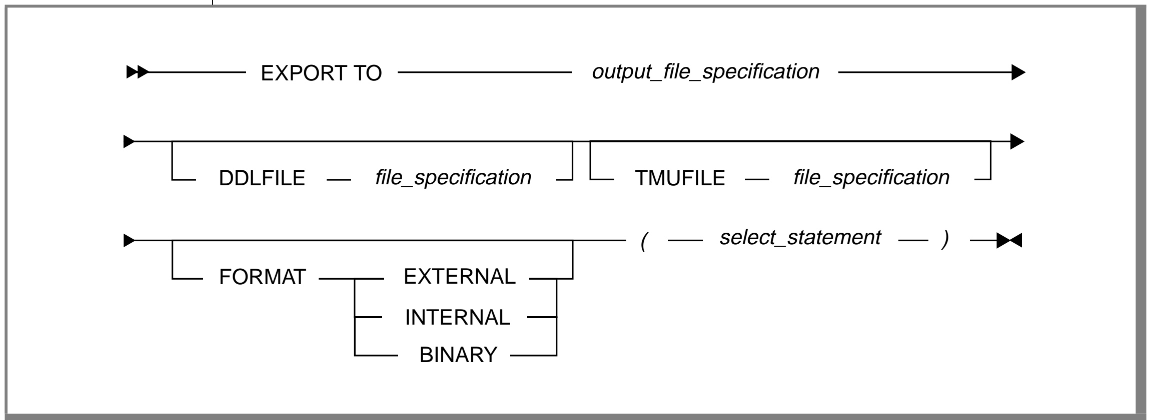
EXPAND

EXPAND [(width)] SQL_statement

EXPLAIN

EXPLAIN SQL_statement

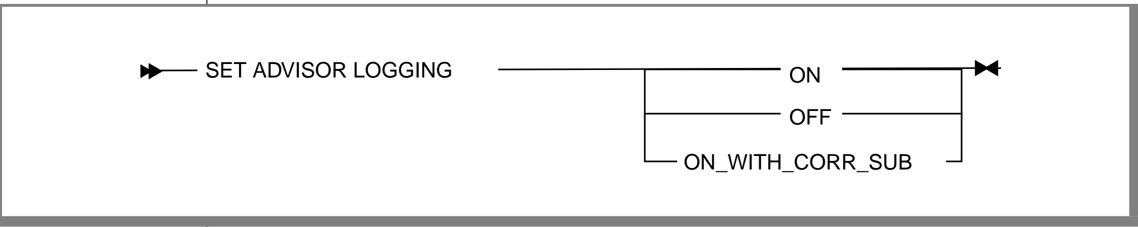
EXPORT



SET Commands

The SET commands are used to change the behavior or performance of the server. These commands can be entered anywhere you can enter an SQL command; in addition, many of the parameters can also be set in the *rbw.config* file.

SET ADVISOR LOGGING



SET ARITHIGNORE, ARITHABORT

```
>>-- SET --+-- ARITHABORT --+------------------><
           +-- ARITHIGNORE -+
```

SET AUTO INVALIDATE PRECOMPUTED VIEWS

```
>>-- SET AUTO INVALIDATE PRECOMPUTED VIEWS --+-- ON --+-->< 
                                             +-- OFF -+
```

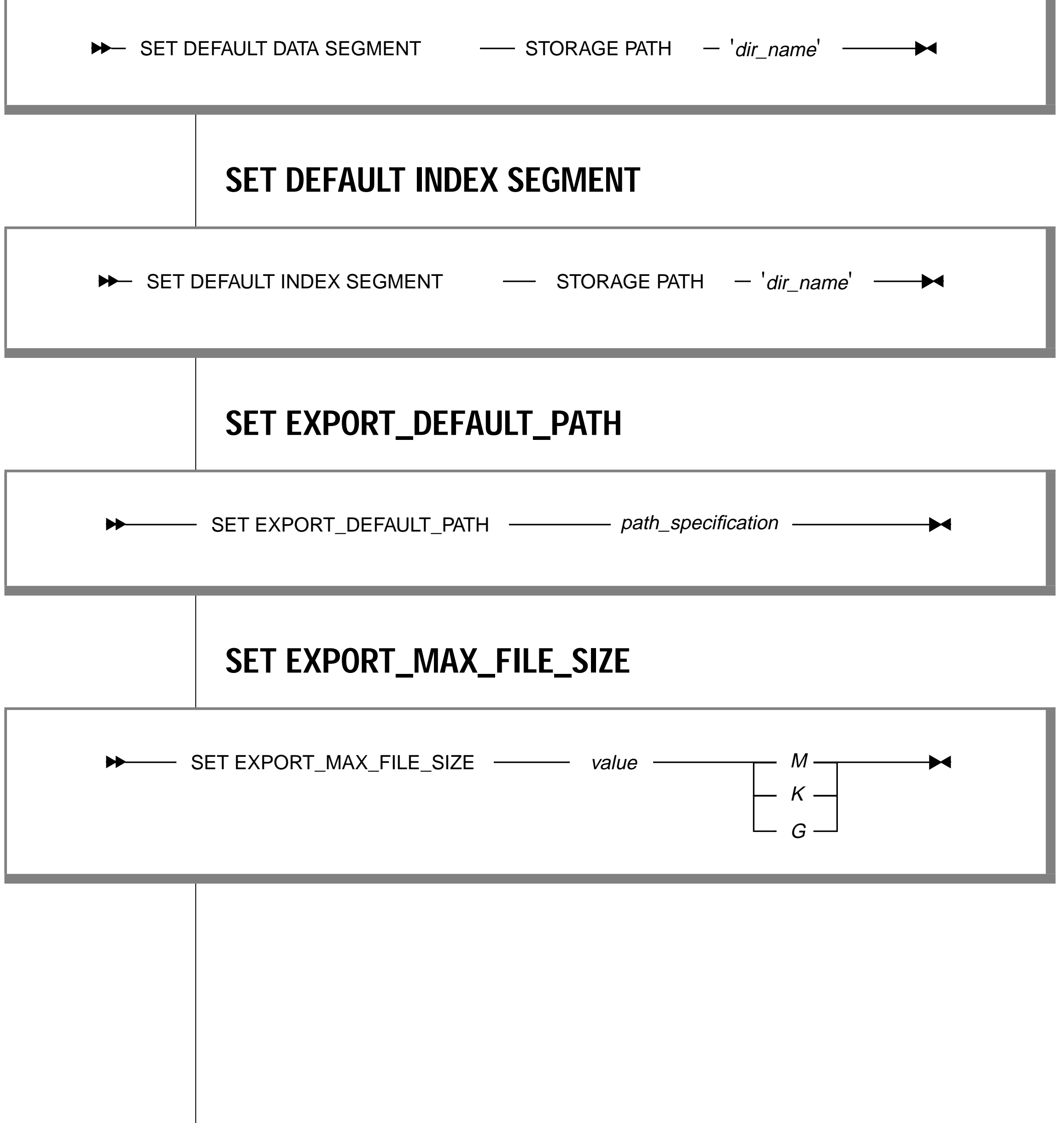
SET COUNT RESULT

```
>>-- SET COUNT RESULT --+-- INTEGER --+-->< 
                        +-- INT ------+
                        +-- DECIMAL --+
                        +-- DEC ------+
```

SET CROSS JOIN

```
>>-- SET CROSS JOIN --+-- OFF --+------------><
                      +-- ON ---+
```

SET DEFAULT DATA SEGMENT



SET DEFAULT INDEX SEGMENT

SET EXPORT_DEFAULT_PATH

SET EXPORT_MAX_FILE_SIZE

SET FIRST DAYOFWEEK

```
SET FIRST DAYOFWEEK { DEFAULT | daynum }
```

SET FORCE TASKS

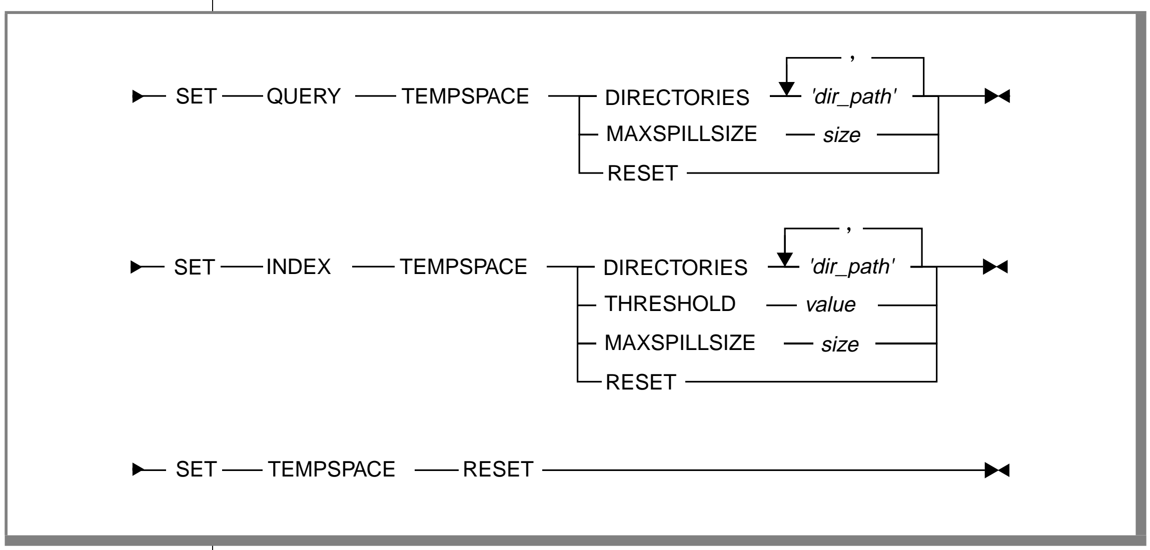
```
SET FORCE_SCAN_TASKS [ OFF | value ]
SET FORCE_FETCH_TASKS [ OFF | value ]
SET FORCE_JOIN_TASKS [ OFF | value ]
SET FORCE_HASHJOIN_TASKS { OFF | value }
```

SET IGNORE OPTICAL INDEXES

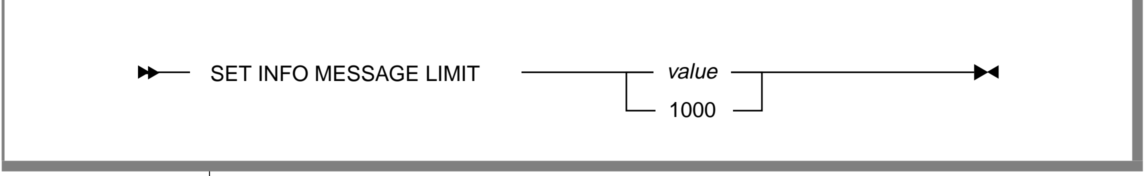
```
SET IGNORE OPTICAL INDEXES { OFF | ON }
```

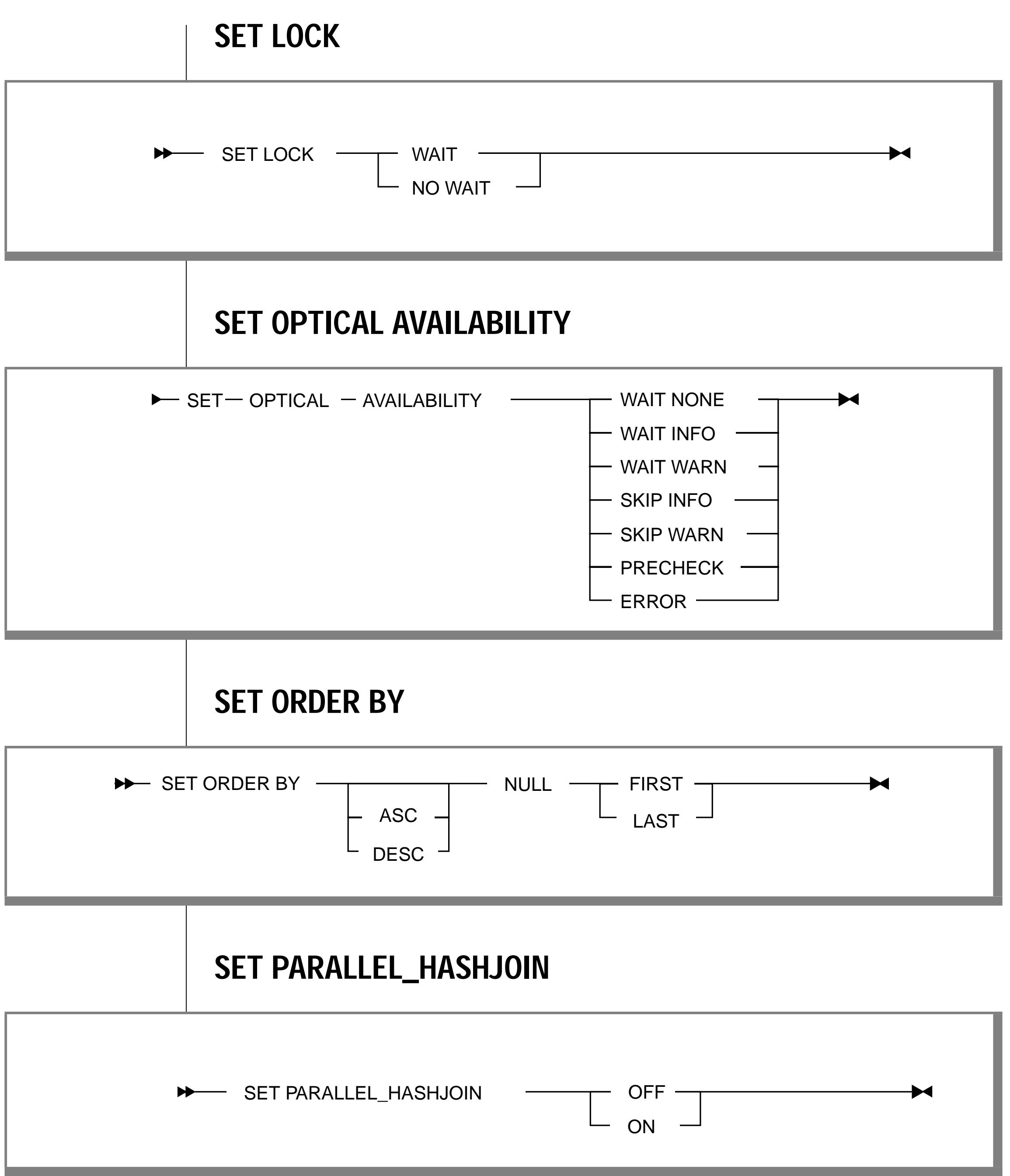
SET IGNORE PARTIAL INDEXES

SET INDEX TEMPSPACE and SET QUERY TEMPSPACE



SET INFO MESSAGE LIMIT




SET LOCK
SET LOCK
WAIT
NO WAIT
SET OPTICAL AVAILABILITY
SET
OPTICAL
AVAILABILITY
WAIT NONE
WAIT INFO
WAIT WARN
SKIP INFO
SKIP WARN
PRECHECK
ERROR
SET ORDER BY
SET ORDER BY
ASC
DESC
NULL
FIRST
LAST
SET PARALLEL_HASHJOIN
SET PARALLEL_HASHJOIN
OFF
ON

SET PARTIAL AVAILABILITY

SET PRECOMPUTED VIEW view_name

SET PRECOMPUTED VIEW QUERY REWRITE

SET PRECOMPUTED VIEWS FOR detail_table

SET QUERY MEMORY LIMIT

```
►── SET QUERY MEMORY LIMIT ──┬── value ───┬──►◄
                             └── DEFAULT ─┘
```

SET QUERYPROCS

```
►►── SET QUERYPROCS ── num_per_query ──────►◄
```

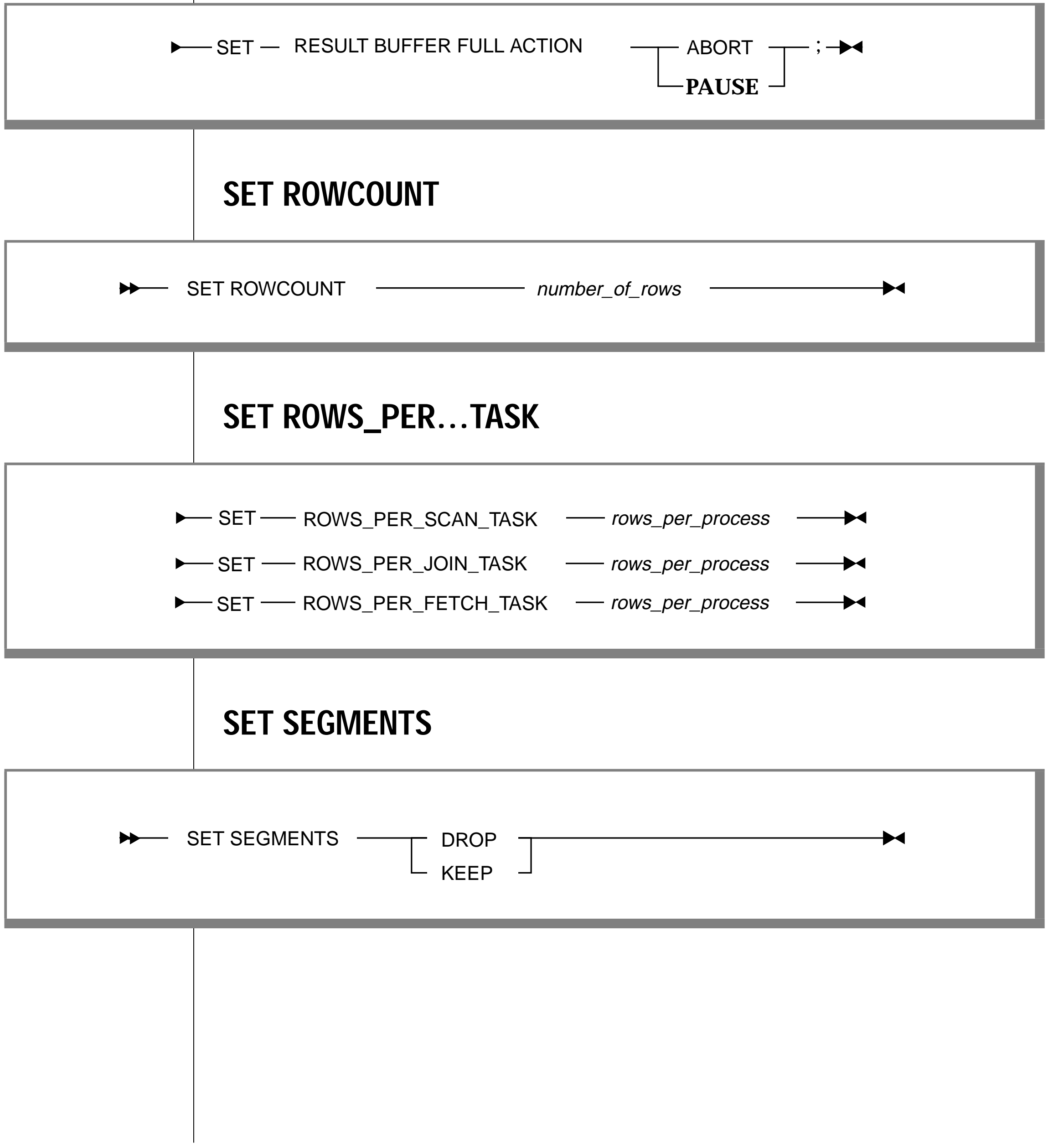
SET REPORT_INTERVAL

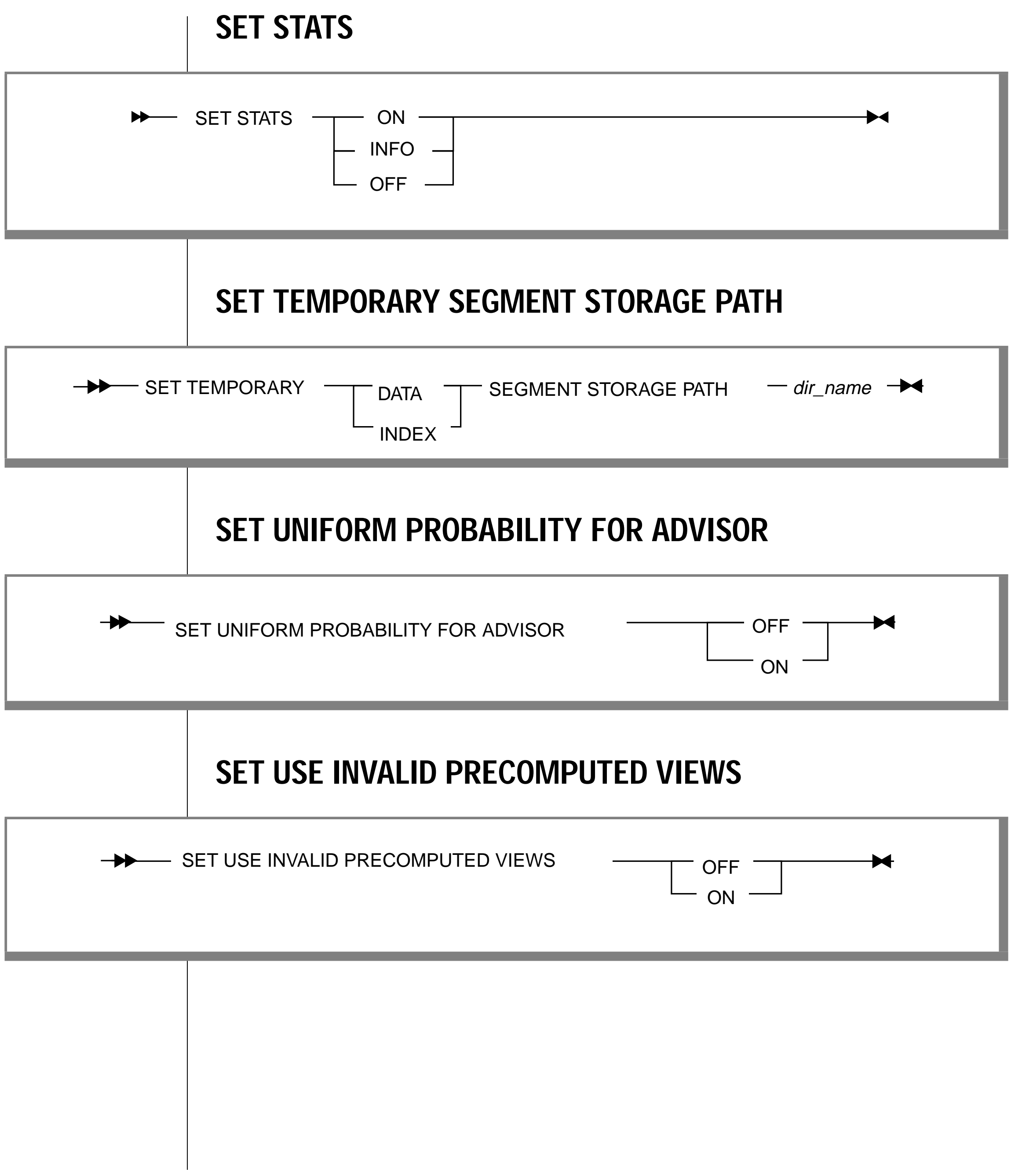
```
►►── SET REPORT_INTERVAL ── integer ──────►◄
```

SET RESULT BUFFER

```
►── SET ── RESULT BUFFER ──┬── value ─┬─K─┬──┬── ; ─►◄
                           │          ├─M─┤  │
                           │          └─G─┘  │
                           └── UNLIMITED ────┘
```

SET RESULT BUFFER FULL ACTION




SET STATS
SET STATS
ON
INFO
OFF
SET TEMPORARY SEGMENT STORAGE PATH
SET TEMPORARY
DATA
INDEX
SEGMENT STORAGE PATH
dir_name
SET UNIFORM PROBABILITY FOR ADVISOR
SET UNIFORM PROBABILITY FOR ADVISOR
OFF
ON
SET USE INVALID PRECOMPUTED VIEWS
SET USE INVALID PRECOMPUTED VIEWS
OFF
ON

Appendix

Reserved Words

B

An SQL or RISQL reserved word cannot be used as a database name or identifier. The words listed in the following table are reserved words.

Also avoid naming database objects in the following format:

```
rbw_object_name
dst_object_name
```

The *rbw_* and *dst_* prefixes are used to name the Red Brick Decision Server system tables, dynamic statistic tables (DSTs), and other objects such as indexes and columns.

ADD

ALL

ALTER

AND

ANY

AS

ASC

ATTACH

AUTHORIZATION

AVG

BACKUP

BETWEEN

BIT

BREAK

BY

CASE

CHANGE

CHANGEPATH

CHAR

CHARACTER

CHECK

CLOSE

COALESCE

COBOL

COLUMN

COMMIT

CONNECT

CONSTRAINT

CONSTRAINTS

COUNT

CREATE

CROSS

CUME

CURRENT

CURRENT_DATE

CURRENT_TIME

CURRENT_TIMESTAMP

CURRENT_USER

CURSOR

DATA

DATABASE

DBA

DEC

DECIMAL

DECLARE

DEFAULT

DEFERRED

DELETE

DESC

DESCRIBE

DISTINCT

DISTRIBUTED

DOUBLE

DROP

ELSE

END

ESCAPE

EXCEPT

EXCL

EXCLUSIVE

EXISTS

EXPAND

EXPLAIN

EXTRACT

FETCH

FILENAME

FIRST

FLOAT

FOR

FOREIGN

FORTRAN

FROM

FULL

FULLINDEXCHECK

GRANT

GROUP

HAVING

IMMEDIATE

IN

INCL

INCLUSIVE

INDEX

INDICATOR

INITIALLY

INNER

INSERT

INT

INTEGER

INTERSECT

INTO

IS

JOIN

KEY

LANGUAGE

LAST

LEFT

LIKE

LOCK

MACRO

MAPPING

MAX

MIN

MODEL

MODULE

MOVINGAVG

MOVINGSUM

NATURAL

NO

NOT

NTILE

NULL

NULLIF

NUMERIC

OF

OFF

ON

OPEN

OPTION

OR

ORDER

OUTER

OVERRIDE

PASCAL

PERMANENT

PL1

PRECISION

PREPARE

PRIMARY

PRIVILEGES

PROCEDURE

PUBLIC

QUIT

RANK

RATIOTOREPORT

REAL

REFERENCES

RESET

RESOURCE

RESUME

REVOKE

RIGHT

ROLE

ROLLBACK

SCHEMA

SEGMENT

SELECT

SET

SMALLINT

SOME

SQLCODE

STAR

SUM

SUMMING

SUPPRESS

SYNONYM

TABLE

TEMPORARY

TERTILE

TEXTSIZE

THEN

TINYINT

TO

UNION

UNIQUE

UNLOCK

UPDATE

USER

USING

VALUES

VARCHAR

VARYING

VIEW

WAIT

WHEN

WHERE

WITH

WORK

Alternative Datetime Formats

Red Brick Decision Server supports the following datetime formats for tools that generate datetime formats other than those defined by ANSI SQL-92:

- Alternative date formats, which allow both numeric months and month names; numeric months require the use of the DATEFORMAT variable.
- Alternative time formats
- Alternative timestamp formats, which consist of the alternative date and time formats.
- DATEPART scalar function, which is similar to the ANSI SQL-92 EXTRACT function.

This support is provided as a transition from SQL Server date formats to ANSI SQL-92; new database development and tool support should use the ANSI SQL-92 definitions.

Use of an alternative datetime format is implied by the absence of the DATE, TIME, or TIMESTAMP keyword in a datetime literal.

This chapter is divided into four main sections:

- Alternative Date Literals
- Alternative Time Literals
- Alternative Timestamp Literals
- Dateparts and the DATEPART Function

Alternative Date Literals

Alternative date formats allow the month to be expressed either by number or by name, and years can be expressed with either 2 or 4 digits. Because date formats with numeric months permit ambiguity (for example, does 7/4/99 mean July 4, 1999, or April 7, 1999?) you must set the DATEFORMAT variable to define the order of the date components.

If the server locale does not specify the language as English and the territory as UnitedStates, the formatting rules associated with the specified language and location are followed and the SET DATEFORMAT command is ignored. Literals that are not consistent with the ANSI SQL-92 standard are not supported in non-U.S.-English locales.

Setting DATEFORMAT

The following syntax diagram shows how to set the DATEFORMAT variable:

Usage Notes

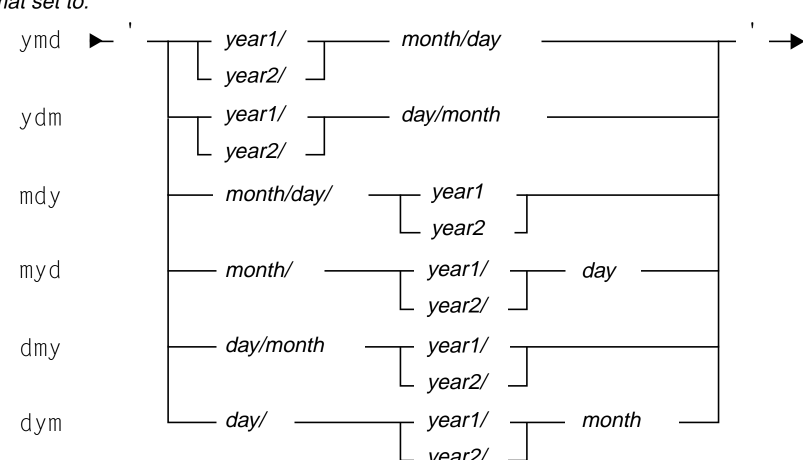
- Set the DATEFORMAT variable just as you enter SQL or RISQL statements.
- *y* is year, *m* is month, and *d* is day.
  The default value is *mdy*, or month day year.
- The DATEFORMAT value is used only when months are represented with integers and the DATE keyword is not present.

The SET DATEFORMAT command cannot be used to change the display of date values; its purpose is to inform the server what format is used by dates that do not comply with ANSI SQL-92.

Using Month Numbers

The following syntax diagram shows how to construct alternative date literals with numeric months:

Date format set to:



Usage Notes

- The date format is determined by the DATEFORMAT variable.
- The separators between the date elements can be a slash (/) as shown, a hyphen (-) , or a period (.). Spaces between date elements and separators are not allowed.
- *year1*, *year2*, *month*, and *day* are unsigned integers, with the following ranges:

year1	1 to 9999 inclusive
year2	1 to 99 inclusive; 00–49 inclusive implies 2000 to 2049 50–99 inclusive implies 1950 to 1999
month	1 to 12
day	1 to 31

Examples

The following table contains examples of valid date literals that use numbers to designate months. In each case, the separator can be a slash (/), as shown, a hyphen (-), or a period (.).

DATEFORMAT value	4-digit years:	2-digit years:
ymd	'1999/4/15'	'99/4/15'
ydm	'1999/15/4'	'99/15/4'
mdy	'4/15/1999'	'4/15/99'
myd	'4/1999/15'	'4/99/15'
dmy	'15/4/1999'	'15/4/99'
dym	'15/1999/4'	'15/99/4'

The following query, which uses a date literal to constrain a DATE column, will fail unless DATEFORMAT is set to *ymd*:

```
select * from period where date_col = '1999-12-25'
```

If DATEFORMAT is set to *dmy*, the following query is valid:

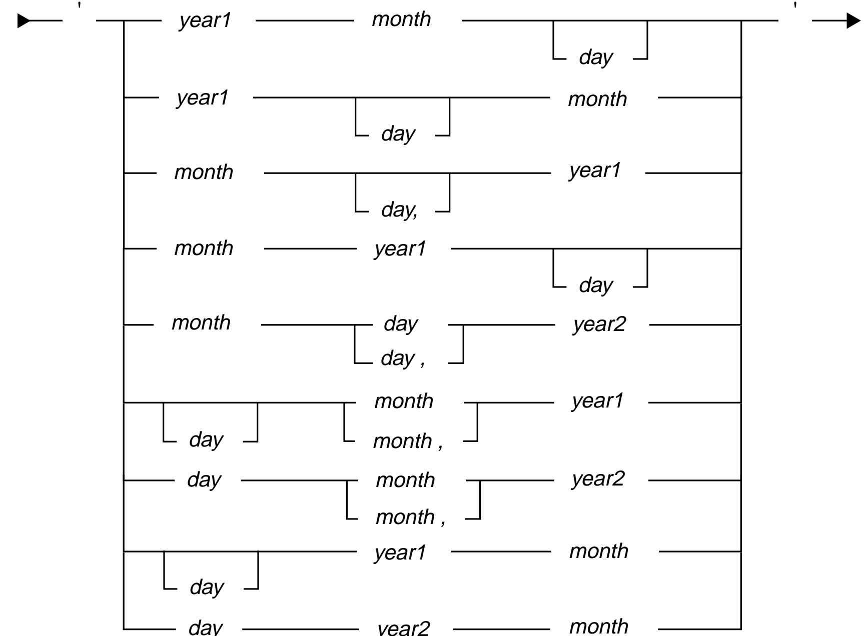
```
select * from period where date_col = '25-12-1999'
```

Because the default value for DATEFORMAT is *mdy*, the following query will be accepted even if DATEFORMAT is not set explicitly:

```
select * from period where date_col = '12-25-1999'
```

Using Month Names

This syntax diagram shows how to construct dates using month names:



Usage Notes

- Each component (year1, year2, month, day) is separated from the following component by one or more spaces.
- If a comma is used, no spaces are allowed between a component and the comma that follows it. One or more spaces must follow a comma.
- *year1*, *year2*, and *day* are unsigned integers, with ranges:

year1	1 to 9999 inclusive
year2	0 to 99 inclusive; 00–49 inclusive implies 2000 to 2049 50–99 inclusive implies 1950 to 1999
day	1 to 31 (The default value is 1)

- *month* must be either the full month name or the first three letters of the month name. For example, *January* or *Jan*. Month names are not case-sensitive.

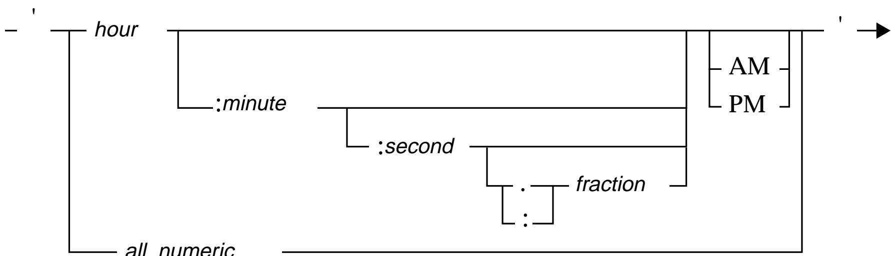
Examples

The following examples are valid date values, with months represented by name. In each example, *April* can be replaced by *Apr*.

Day value and 4-digit years	Day value and 2-digit years	No day value and 4-digit years
'1988 April 15'	'April 15 88'	'1988 April'
'1988 15 April'	'April 15, 88'	'April, 1988'
'April 15 1988'	'15 April 88'	'April 1988'
'April 15, 1988'	'15 April, 88'	
'April 1988 15'	'15 88 April'	
'15 April 1988'		
'15 April, 1988		

Alternative Time Literals

The following syntax diagram shows how to construct alternative time values:



Usage Notes

- No spaces are allowed between time components except before the AM/PM designators.
- *hour*, *minute*, *second*, and *fraction* must be unsigned integers with the following ranges:

hour	0 to 23; if AM/PM designator is used, then 0 to 12
minute	0 to 59
second	0 to 59
fraction	0 to 999999

- The AM/PM designators are not case-sensitive.
- If a fraction of a second is preceded by a decimal (.), it is interpreted as a decimal fraction of a second. If it is preceded by a colon (:), it is interpreted as milliseconds.

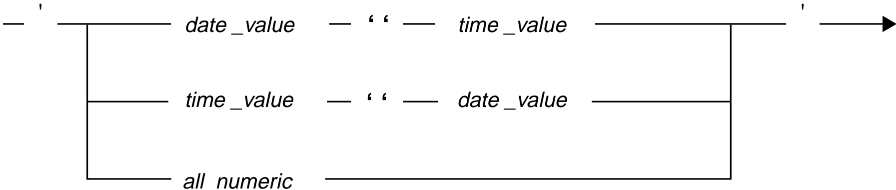
Examples

The following examples are valid time values:

```
'13:45:10:33'                (33 milliseconds)
'13:45:10.33'                (33/100 seconds)
'3AM'
'3 PM'
'03pm'
```

Alternative Timestamp Literals

The following syntax diagram shows how to construct alternative timestamp values:



Usage Notes

- *date_value* and *time_value* can be any of the strings defined for the alternative date and alternative time formats.
- Exactly one space is required between the *date_value* and *time_value*.
- A date value can be specified in an all-numeric format of 4, 6, or 8 digits but a timestamp component is not supported.

  4 digits: January 1, where 4 digits are interpreted as year
  6 digits: 19YYMMDD, where 6 digits are interpreted as YYMMDD
  8 digits: YYYYMMDD.
- If not specified, the default is used. The default for *date_value* is January 1, 1900. The default for *time_value* is midnight.

Examples

The following examples are valid timestamp examples:

```
'1988 April 15 3:45am'
'4/15/88 13:45:10:33'
'13:45:10:33 4-15-88'
'13:45:10 April 1988'
'April 15, 1988 3AM'
```

The following examples are valid all-numeric timestamp examples:

`'1234'` (Jan 1, 1234)
`'780123'` (Jan 23, 1978)
`'12340506'` (May 6, 1234)

Dateparts and the DATEPART Function

Some scalar functions operate on the individual elements that comprise the datetime datatypes, or dateparts; these elements are defined under "Datetime Scalar Functions" on page 5-54.

The DATEPART function, similar to the ANSI SQL-92 EXTRACT function, extracts the specified datepart component from a datetime value. The following syntax diagram shows how to construct a datetime expression with the DATEPART function:

—DATEPART – (— *datepart,* — *datetime_expression* —) —▶

datepart The *datepart* argument specifies the datepart from which to extract the date component.

datetime_expression The expression must be a DATE, TIME, or TIMESTAMP datatype.

Result

If the datetime expression does not contain the specified datepart, a default time of midnight is used for missing time parts and a default date of "1900 Jan 1" is used for missing date parts. These default values are also returned as character strings.

If *datepart* is week, these functions take into account the day of the week on which January 1 fell in the specified year. For example, if January 1 is a Saturday, January 2 is in week 2.

Example

```
select datepart (year, date_col) from table_1
```

The datepart function can be used to change the format of the displayed date from the default YYYY/MM/DD format:

Example

```
select date,
concat (substr (string (datepart(month,date_col)), 10, 2),
'/',
substr (string (datepart (day,date_col)), 10, 2),
'/',
substr (string (datepart (year,date_col)), 10, 2)) as
new_date
from period;
DATE                    NEW_DATE
1998-01-01              1/1/98
1998-01-02              1/2/98
1998-01-03              1/3/98
```

The "/" can be omitted or changed to"-" as needed.

Index

A

ABS function 5-15
ADD COLUMN clause, ALTER TABLE command 8-68
ADD CONSTRAINT specification, ALTER TABLE command 8-75
ADD STORAGE clause, ALTER SEGMENT command 8-45
aggregate tables 8-162, 8-165, 8-184
aggregation columns 8-161
aggregation functions, *See* set functions
aliases
  column 7-19
  defined 2-7
ALL comparison predicate 3-11
ALL keyword
  AVG function 4-4
  in select list 7-18
ALL PRIVILEGES, granting 8-204
ALTER COLUMN specification, ALTER TABLE command 8-72
ALTER CONSTRAINT specification, ALTER TABLE command 8-80
ALTER DATABASE command
  description 8-6
  syntax description 8-7
  syntax summary A-2
ALTER DATABASE CREATE BACKUP DATA command 8-8
ALTER DATABASE DROP BACKUP DATA command 8-8
ALTER INDEX command
  examples 8-19
  syntax description 8-15
  syntax summary A-8
  TARGET indexes 8-18
  usage 8-18
ALTER MACRO command
  syntax description 8-19
  syntax summary A-8
ALTER ROLE command
  syntax description 8-21
  syntax summary A-9
ALTER SEGMENT command
  ATTACH clause 8-23 to 8-29
  DETACH clause 8-32
  examples 8-48 to 8-49
  MIGRATE TO clause 8-44
  modify clauses 8-30 to 8-47
  overriding full index checks 8-33, 8-42
  overriding referential integrity checks 8-39, 8-42
  SEGMENT BY segmenting column 8-37
  syntax description 8-24 to 8-47
  syntax summary A-9
ALTER SYNONYM command
  syntax description 8-50
  syntax summary A-11
ALTER SYSTEM command
  alter accounting specification 8-55 to 8-61
  alter logging specification 8-55 to 8-60
  alter user activity specification 8-54, 8-55 to 8-57

alter user priority
specification 8-58 to 8-59
description 8-51
syntax description 8-52 to 8-61
syntax summary A-2
ALTER TABLE command 8-62, 8-137
ADD COLUMN
specification 8-67 to 8-69
ADD CONSTRAINT
specification 8-75
ALTER COLUMN
specification 8-72 to 8-74
ALTER CONSTRAINT
specification 8-80
deferring referential integrity
checks 8-77
DROP COLUMN
specification 8-70 to 8-71
DROP CONSTRAINT
specification 8-79
MAXROWS PER SEGMENT
value 8-66
MAXSEGMENTS value 8-66
syntax description 8-63 to 8-74
syntax summary A-12
ALTER USER command
syntax description 8-83
syntax summary A-5
usage notes 8-84
ALTER VIEW command
syntax description 8-84
syntax summary A-15
alternative table names 7-14
AND connective 3-19
ANY comparison predicate 3-11
arithmetic operators, precedence of 3-7
Aroma database 1-14
AS keyword, for column aliases 2-7, 7-19
ASC 7-48
ascending order 7-48
assignment
character 2-27
datetime 2-28
numeric 2-29
asterisk, in select list 7-18
authorizations and privileges
required for SQL execution 1-9
averages, moving 6-9
AVG function 4-4, 8-161

B

backup segment
altering 8-30
creating 8-6
dropping 8-178
dropping data 8-6
BETWEEN predicate 3-13
BREAK BY subclause
disallowed in INSERT
statement 7-47, 7-56
of ORDER BY clause 7-55
B-TREE indexes, *See* indexes, B-TREE

C

calculations
RISQL display functions 6-3
CANCEL USER COMMAND clause, ALTER SYSTEM command 8-56
Cartesian products
computed by cross joins 7-13
preventing computation of 9-10
CASCADE keyword
ALTER TABLE command 8-73, 8-77, 8-81
CREATE TABLE command 8-143
CASE expressions
syntax description 5-4
with NTILE function 6-17
case sensitivity
character literals 2-11
names and identifiers 2-6
cases, tracked by technical support 12
CATEGORY clause, CREATE MACRO command 8-121
cautions
defining hierarchies 8-92
CEIL function 5-17
CHANGE
ACCOUNTING LEVEL, ALTER SYSTEM command 8-61
DOMAIN clause, ALTER INDEX command 8-17
EXTENDSIZE clause, ALTER SEGMENT command 8-44
FILLFACTOR clause, ALTER INDEX command 8-17
LOGGING LEVEL clause, ALTER SYSTEM command 8-60
MAXROWS clause, ALTER TABLE command 8-66
MAXSIZE clause, ALTER SEGMENT command 8-43
PATH clause, ALTER SEGMENT command 8-44
USER clause, ALTER SYSTEM command 8-58
character
datatypes 2-27
datetime conversion 5-61
empty character string 2-11
ESCAPE 3-17
wildcard 3-17
character data
length of 2-19, 8-137
single- and multibyte 2-3
character data literals
case sensitivity 2-11
length 2-11
single quote 2-11
circular schema references, disallowed 8-78, 8-142
CLEAR keyword, ALTER SEGMENT command 8-42
CLOSE USER SESSION clause, ALTER SYSTEM command 8-56
COALESCE function 5-8
collation sequence 7-49
column functions
columns
adding 8-64, 8-67
aliases 2-7, 7-19
altering existing 8-72
changing refcheck mode 8-73, 8-77, 8-81
comments 8-74

definitions 8-134
dropping 8-64, 8-70
names 8-134
qualified names 2-8, 7-23
renaming 8-73
selecting from tables 7-17
SET values 8-224
commands, *See* SQL commands
COMMENT clause 8-66
columns 8-74
indexes 8-17
macros 8-20, 8-123
roles 8-21
segments 8-45
synonyms 8-51
users 8-83
views 8-85
comment icons 10
comparisons
character 2-29
datetime 2-29
numeric 2-31
predicates 3-10, 3-11
compound expressions 3-5
CONCAT function 5-36, 5-37, 5-39
conditional scalar functions
CASE 5-4
COALESCE 5-8
DECODE 5-9
IFNULL 5-11
NULLIF 5-13
conditions
correlation 2-7
definition 3-10
HAVING clause 7-32
in correlated subqueries 7-64
search 3-19
three-valued logic 2-26
WHEN clause 7-34
configuration file, *See* SET commands, SQL
CONNECT system role
dropping users 8-219
granting 8-199
connectives, logical 3-19
constants
character 2-11
decimal 2-15
floating-point numbers 2-16
integer 2-14
numeric 2-10
constraint names
adding to tables 8-75
altering foreign key constraints 8-81
dropping from tables 8-79
for foreign key references 8-103
foreign key 8-142
primary key 8-140
constraints
altering table 8-80
contact information 17
conventions
syntax diagrams 7
syntax notation 6
correlated subqueries 7-64
correlation conditions 2-7
correlation names 7-65
self-joins 7-15
table 2-7
COUNT function 4-6, 8-161
datatypes of results 9-9
COUNT(*) function 4-6
CREATE HIERARCHY command
described 8-92
syntax description 8-93
syntax summary A-16
CREATE INDEX command
B-TREE indexes 8-99
described 8-98
domain sizes 8-117
examples 8-108
examples of segment specifications 8-114
index specifier 8-102
See also indexes
segment range specification 8-109, 8-113
segment specification 8-106
STAR indexes 8-97
syntax description 8-97
syntax summary A-16
TARGET indexes 8-98
CREATE MACRO command
examples 8-124
syntax description 8-119 to 8-123
syntax summary A-19
CREATE ROLE command
examples 8-126
syntax description 8-125
syntax summary A-19
usage 8-126
CREATE SEGMENT command
examples 8-130
storage specification 8-127
syntax description 8-127 to 8-129
syntax summary A-20
CREATE SYNONYM command
examples 8-132
syntax description 8-131
syntax summary A-20
CREATE TABLE command
described 8-137
examples 8-148, 8-153, 8-153 to 8-154
foreign key references clause 8-141 to 8-143
MAXROWS PER SEGMENT value 8-134
MAXSEGMENTS value 8-134
primary key reference clause 8-140
primary key references 8-139
SEGMENT LIKE DATA clause 8-149
segment specification 8-145 to 8-148
syntax description 8-133, 8-148
syntax summary A-21
CREATE TEMPORARY TABLE command
described 8-154
syntax description 8-156
syntax summary A-23
CREATE VIEW command
examples 8-164
query expressions 8-159
syntax description 8-158
syntax summary A-23
cross joins
SET CROSS JOIN command 9-10
syntax 7-13
usage notes 7-23
cross-references, as correlation conditions 7-64
CUME function 6-5

CURRENT_DATE function 5-56
CURRENT_TIME function 5-56
CURRENT_TIMESTAMP
function 5-56
CURRENT_USER function 5-67
cyclical schema references,
disallowed 8-78

D

damaged segments 8-182
data cube, *See* simple star schema
DATABASE keyword, ALTER
SYSTEM command 8-53
databases
adding users 8-198
Aroma 1-14
changing passwords 8-198
dropping users 8-218
object names 2-3
datatypes
compatibility of 2-27
CREATE TABLE command 8-136
datetime 2-21
decimal 2-24
floating-point 2-25
numeric 2-24
summary 2-18
DATE datatype 2-21
DATE function 5-57
date literals
ANSI SQL-92 standard 2-12
non-standard C-2
DATEADD function 5-58
DATEDIFF function 5-59
DATEFORMAT variable C-2
DATENAME function 5-61
dateparts
DATEPART function C-9
defined 5-54
with EXTRACT 5-63
datetime datatypes 2-21
datetime literals
ANSI SQL-92 standard 2-12
nonstandard C-1
datetime scalar functions
CURRENT_DATE 5-56
CURRENT_TIME 5-56
CURRENT_TIMESTAMP 5-56
DATE 5-57
DATEADD 5-58
DATEDIFF 5-59
DATENAME 5-61
EXTRACT 5-63
TIME 5-65
TIMESTAMP 5-66
DBA system role
granting 8-193
list of task authorizations 8-194
revoking 8-216
DEC function 5-19, 5-20
decimal constant 2-15
DECODE function 5-9
DEC, DECIMAL datatype 2-24
DEFAULT clause
ALTER TABLE command 8-69
CREATE TABLE command 8-138
DEFAULT parameters, *See* SET
commands, SQL
DEFAULT VALUES subclause,
INSERT command 8-208
DELETE command
examples 8-169 to 8-171
syntax description 8-166 to 8-167
syntax summary A-24, A-25,
A-27, A-28
usage 8-168
DELETE privilege 8-204
delete trigger action 8-73, 8-77,
8-81, 8-143
delimited identifiers 2-5
in macro names 8-121
demonstration database
script to install 4
dependencies, software 4
derived dimensions, precomputed
view for 8-165
derived tables 7-15
examples 7-25
syntax 7-22
DESC 7-48
descending order 7-48
DETACH clause, ALTER
SEGMENT command 8-32
dimension tables, *See* referenced
tables
display functions, *See* RISQL
display functions
DISTINCT function 8-162
DISTINCT keyword
AVG function 4-4
COUNT function 4-7
in select list 7-18
division by zero 9-7
documentation
list of Red Brick Decision
Server 13
documentation, types of
on-line manuals 16
printed manuals 16
domain sizes
altering for TARGET
indexes 8-18
defining, for TARGET
indexes 8-117
domain, defined 8-98
DROP COLUMN specification,
ALTER TABLE command 8-71
DROP CONSTRAINT specification,
ALTER TABLE command 8-79
DROP DEFAULT clause, ALTER
TABLE command 8-74
DROP HIERARCHY command
described 8-171
syntax description 8-172
syntax summary A-24
DROP INDEX command
examples 8-173
restrictions on inserts 8-174
syntax description 8-173
syntax summary A-24
DROP MACRO command
syntax description 8-176
syntax summary A-24
DROP ROLE command
example 8-177
syntax description 8-177
syntax summary A-24
DROP SEGMENT command
examples 8-179
syntax description 8-178
syntax summary A-24
DROP SYNONYM command
syntax description 8-180
syntax summary A-25

DROP TABLE command
example 8-183
syntax description 8-181 to 8-182
syntax summary A-25
DROP VIEW command
syntax description 8-183
syntax summary A-25
DROPPING SEGMENTS keyword
DROP INDEX command 8-173
DROP TABLE command 8-182
DSTs (dynamic statistic tables) 9-44
duplicate rows 7-18
dynamic statistic tables (DSTs) 9-44

E

embedded macros 8-123
entity integrity 1-9
equijoins 7-7
errors, division by zero 9-7
escape character
LIKE predicate 3-17
macro name 8-123
EXCEPT operator 7-36
EXISTS predicate 3-14
EXPAND command
syntax description 8-184
syntax summary A-28
expected domain size 8-98
EXPLAIN command
syntax description 8-185
syntax summary A-28
explicit tables 7-18
EXPORT command
syntax description 8-187
syntax summary A-29
expressions
arithmetic 3-7
as search condition for join 7-11
compound 3-5
evaluation 3-7
nested 3-6
query 7-3 to 7-68, ?? to 7-68
RISQL display functions as 6-4
simple 3-4
EXTENDSIZE clause, ALTER SEGMENT command 8-47, 8-129
EXTRACT
dateparts 5-54
function 5-63
EXTRACT function
DATENAME function 9-17

F

fact tables, *See* referencing tables
feature icons 10
features of this product, new 5
filenames, restricted to single-byte characters 2-4
fill factors
CREATE INDEX command 8-116
FLOAT datatype 2-25
FLOAT function 5-22
floating-point constant 2-16
FLOOR function 5-23
FOR DELETE option, LOCK command 8-213
FORCE INTACT clause, ALTER SEGMENT command 8-36
FOREIGN KEY clause, CREATE TABLE command 8-142
foreign keys 1-7
constraint names 8-103, 8-142
definition 8-142
restrictions on updates 8-226
FROM clause 7-21
cross joins 7-23
inner and outer joins in 7-22
join specifications 7-23
NATURAL JOIN syntax 7-23
ON subclause 7-23
outer joins 7-22
performance of subqueries in 7-63
query expressions in 7-15
subqueries in 7-15
table references 7-22
USING subclause 7-23
full index checks, overriding with the ALTER SEGMENT command 8-33, 8-42
full outer joins
syntax 7-10
FULLINDEXCHECK keyword, ALTER SEGMENT command 8-33, 8-42
functional dependencies, *See* hierarchies
functions, SQL
RISQL display 6-26
scalar 5-3 to 5-68
set 4-3 to 4-11

G

GRANT authorization and role command
examples 8-197
syntax description 8-193 to 8-198
syntax summary A-5
usage 8-196
GRANT CONNECT command
examples 8-202
syntax description 8-199 to 8-201
syntax summary A-5
GRANT privilege command
examples 8-205
syntax description 8-203 to 8-204
syntax summary A-6
GROUP BY clause 7-29
group functions

H

hash joins, parallel 9-37
HAVING clause 7-32
hierarchies
creating 8-92
dropping 8-171
hybrid hash joins, parallel 9-37
hybrid target indexes, *See* TARGET indexes, mixed-domain

I

icons
  feature 10
  important 10
  platform 10
  tip 10
  warning 10
identifiers
  delimited 2-5, 8-121
  standard 2-4
IFNULL function 2-27, 5-11
IGNORE_OPTICAL_INDEXES parameter, syntax 9-24
important paragraphs, icon for 10
IN predicate 3-15
indexes
  altering 8-14
  B-TREE 8-99
    dropping system-generated 8-174
  comments 8-17
  creating 8-97
  creating in named segments 8-106 to 8-115
  domain sizes 8-117
  dropping 8-172
  fill factor 8-116
  fill factors, altering 8-14
  names 8-16, 8-103
  naming 2-3
  order of key columns in 8-175
  partially available 9-26
  selection with optical segments 9-24
  STAR 8-174
  system-created names 8-16
  TARGET
    creating 8-98
    mixed-domain 8-117
  types 1-7
  user-created 8-103
informational messages, limiting display 9-31
Informix Customer Support 11
INFO_MESSAGE_LIMIT parameter, rbw.config file 9-31
INITIALLY DEFERRED option, ALTER TABLE ADD CONSTRAINT command 8-77
INITSIZE clause, ALTER SEGMENT command 8-47, 8-129
inner joins 7-7
INSERT command
  assignment of values 2-27
  examples 8-210 to 8-211
  restrictions on inserts 8-174
  syntax description 8-206 to 8-208
  syntax summary A-26
  usage 8-209
INSERT privilege 8-204
INT datatype 2-23
INT function 5-25, 5-27, 5-31
integers
  constant 2-14
  INT 2-23
  SMALLINT 2-23
  TINYINT 2-23
INTERSECT operator 7-36
interval, datetime 2-21
inverse ranking 6-19
IN_PLACE keyword, ALTER TABLE command 8-64
IS NOT NULL predicate 2-26
IS NULL predicate 2-26, 3-16

J

join query expressions 7-4
joined tables 7-6, 7-22
  as table references 7-22
  Cartesian products 9-10
  cross joins 7-13
  defined 7-22
  FROM clause specifications 7-23
  inner joins 7-7
  join specifications 7-7
  outer joins 7-8
  self-joins 7-15
  syntax 7-23
  system tables 7-26
  WHERE clause specifications 7-27

K

KEEPING SEGMENTS keyword
  DROP INDEX command 8-173
  DROP TABLE command 8-182
keys
  foreign 1-7
  foreign key references 8-141
  primary 1-7, 8-140
keywords
  in syntax diagrams 9
  reserved B-1

L

left outer joins
  syntax 7-10
length, character literal 2-11
LIKE predicate 3-17
literals
  character 2-10
  datetime 2-10, 2-12
  decimal 2-15
  integer 2-14
  length of character 2-11
localized output for scalar functions 5-54
LOCK DATABASE command
  syntax description 8-215
  syntax summary A-27
  usage 8-215
LOCK table command
  example 8-214
  syntax description 8-212
  syntax summary A-27
logical connectives 3-19
logic, three-valued 2-26
LOWER function 5-40
lowercase and uppercase
  conversion 2-6
  literals 2-11
LTRIM function 5-41

M

macros
  categories 8-121
  comments 8-19, 8-123
  creating 8-118
  defined 1-12
  dropping 8-120, 8-175
  embedded 8-123
  expansion 8-184
  naming 2-3
  parameters 8-121
  private 8-120
  PUBLIC 8-120
major tables, *See* referencing tables
MAX function 4-8, 8-161
maximum number of segments, for tables 8-134
MAXROWS PER SEGMENT value
  ALTER TABLE command 8-66
  CREATE TABLE command 8-134
MAXSEGMENTS value
  ALTER TABLE command 8-66
  CREATE TABLE command 8-134
MAXSIZE parameter
  ALTER SEGMENT command 8-46
  CREATE SEGMENT command 8-128
MAXSPILLSIZE value, SET INDEX/QUERY TEMPSPACE command 9-30
MAX, MIN keywords, in ranges 8-110, 8-152
memory-tuning parameters 9-41
MIGRATE TO clause, ALTER SEGMENT command 8-44
MIN function 4-9, 8-161
minor tables, *See* referenced tables
MIN, MAX keywords, in ranges 8-110, 8-152
missing values 3-9
  character string 2-11
  evaluation 2-26
  NULL 2-26
mixed-domain TARGET indexes 8-117
MOVINGAVG function 6-9
MOVINGSUM function 6-12

N

named-columns joins 7-12, 7-23
naming database objects 2-3
natural joins 7-10, 7-23
nested aggregates 4-3
nested expressions 3-6
nested scalar functions 5-3
new features of this product 5
NO ACTION keyword
  ALTER TABLE command 8-73, 8-77, 8-81
  CREATE TABLE command 8-143
NO WAIT keywords
  LOCK command 8-212
  LOCK DATABASE command 8-214
  SET LOCK command 9-32
non-equijoins, when executed as cross joins 9-10
non-join query expressions 7-4
NOT NULL keywords
  ALTER TABLE command 8-68
  CREATE TABLE command 8-137
notation conventions 5
NTILE function 6-15
NULL
  arithmetic operators 3-9
  character string 2-11
  missing values 2-27
  placement 9-36
  removing values with SUPPRESS BY clause 7-58
  search condition evaluation 3-19
  SET ORDER BY command 9-36
  values displayed by outer joins 7-8
NULLIF function 5-13
numeric constants 2-10
numeric scalar functions
  ABS 5-15
  CEIL 5-17
  DEC 5-19, 5-20
  FLOAT 5-22
  FLOOR 5-23
  INT 5-25, 5-27, 5-31
  REAL 5-28
  SIGN 5-29

O

object privileges
  granting 8-202
  revoking 8-219
objects, database 2-3
OFFLINE mode, ALTER SEGMENT command 8-39
ON DELETE clause
  ALTER TABLE command 8-73, 8-77, 8-81
  CREATE TABLE command 8-143
ON ERROR ABORT clause, CREATE INDEX command 8-99
ON ERROR CONTINUE clause, CREATE INDEX command 8-99
ON subclause, in FROM clause 7-23
on-line manuals 16
ONLINE mode, ALTER SEGMENT command 8-41
operators
  arithmetic 3-7
  null value 3-9
optical storage
  access to 9-33
  index selection 9-24
  moving segments to or from 8-44
OPTICAL_AVAILABILITY parameter, syntax 9-33
OR connective 3-19
Oracle-style syntax, for outer joins 7-28
ORDER BY clause 7-47
  BREAK BY subclause 7-55
  RESET By subclause 7-52
  restrictions for UNION, EXCEPT, and INTERSECT expressions 7-49
order of precedence 3-7
  arithmetic operators 3-7
  logical connectives 3-20
ordering rows in a table 7-47
outer joins
  defined 7-8
  FROM clause 7-10
  WHERE clause 7-28

OVERRIDE REFCHECK clause
  ALTER SEGMENT
    command 8-39, 8-42
  CREATE TABLE command 8-143
  DELETE command 8-167

P

parallel hash joins 9-37
parallel queries, allocating tasks
    for 9-19 to 9-22
parameters, macro 8-121
parentheses, order of
    precedence 3-7
passwords
  adding new users 8-200
  changing 8-200
  restricted to single-byte
    characters 2-4
percent wildcard character 3-17
physical storage units
  extend size 8-44, 8-129
  initial size 8-47, 8-129
  maximum size 8-43, 8-128
  names 8-128
  pathnames 8-44
  sequence ID 8-43
platform icons 10
precedence, order of 3-7
precision
  decimal constant 2-15
  decimal datatype 2-24
  double precision 2-25
  integer constant 2-14
  numeric datatypes 2-10
precomputed query
    expression 8-159
precomputed views 8-165, 9-40
  creating 8-157
  described 8-158
  SET commands 9-8, 9-39, 9-41,
    9-57
predicates
  ALL 3-11
  BETWEEN 3-13
  comparison 3-11
  constructing 3-10
  EXISTS 3-14
  IN 3-15
  IS NULL 3-16
  LIKE 3-17
  quantified 3-11
  SOME or ANY 3-11
primary keys 1-7, 8-139
  constraint names 8-140
  naming columns 8-140
  reference clause, CREATE TABLE
    command 8-140
  restrictions on updates 8-226
printed manuals 16
private macros 8-120
privileges, *See* object privileges
PROCESS clause, ALTER SYSTEM
    command 8-56, 8-59
processes
  allocation for parallel
    queries 9-19 to 9-22
  number per query 9-43
processing, SELECT
    statements 7-59
PSU sequence ID 8-43
PSUs, *See* physical storage units
PUBLIC
  access 8-203
  object privileges 8-204

Q

qualified column names 2-8, 7-23
  in ORDER BY clause 7-49
qualified joins 7-23
  description 7-7
  usage notes 7-23
quantified predicates 3-10
queries
  parallel processes for 9-19 to 9-22
  stopping execution with SET
    ROWCOUNT 9-46
  writing 7-3 to 7-68
query expressions 7-3 to 7-68
  correlated subqueries 7-64
  examples 7-25
  explicit tables 7-6
  in views 8-159
  joined tables 7-5, 7-6
  overview 7-3
  precomputed 8-159
  query specifications 7-5, 7-16
  simple tables 7-6
  subqueries 7-60
  syntax 7-5
  UNION, INTERSECT, and
    EXCEPT expressions 7-6, 7-36
query processes, number per
    query 9-43
query processing, order of
    operations 7-59
query specifications 7-16
QUIESCE clause, ALTER SYSTEM
    command 8-53

R

range values
  CREATE TABLE command 8-151
  in ALTER SEGMENT 8-26, 8-38
  in CREATE INDEX 8-110, 8-112
  segmenting column 8-38
RANK function 6-19
  compared with NTILE 6-15
  top ten 6-19
RATIOTOREPORT function 6-22
rbwadmd daemon 9-44
rbw.config file, *See* SET commands,
    SQL
REAL function 5-28
referenced tables
  segmenting indexes like 8-108
references
  column aliases 7-19
  table 7-21
references, foreign key 8-142
referential integrity 8-143, 8-167
  deferred checking 8-77
  defined 1-9
  overriding checks with ALTER
    SEGMENT 8-39, 8-42
referential integrity cycle
  and ALTER TABLE ADD
    CONSTRAINT
    command 8-78
  not allowed 8-142
relation scans 9-20
relational database tables 1-4

RENAME AS clause, ALTER
TABLE command 8-73
RENAME clause, ALTER
SEGMENT command 8-43
replacement value, DECODE
function 5-9
reserved words B-1
RESET BY subclause
and WHEN clause 7-54
syntax description 7-52
RESET clause, ALTER TABLE
command 8-65
RESET STATISTICS clause, ALTER
SYSTEM command 8-53
resetting TEMPSPACE
parameters 9-30
resolution of column
references 7-66
RESOURCE system role
granting 8-193
list of task authorizations 8-194
revoking 8-216
RESTRICT keyword, ALTER
TABLE command 8-71
RESUME clause
ALTER SYSTEM command 8-53
ALTER TABLE command 8-65
reverse ranking 6-19
REVOKE authorization and role
command
examples 8-217
syntax description 8-216
syntax summary A-6
usage 8-217
REVOKE CONNECT command
syntax description 8-219
syntax summary A-7
REVOKE privilege command
examples 8-221
syntax description 8-220
syntax summary A-7
usage 8-221
right outer joins, syntax 7-10
RISQL display functions
as expressions 3-5, 6-4
CUME 6-5
MOVINGAVG 6-9
MOVINGSUM 6-12
nested inside scalar
functions 5-26
nesting 3-6
NTILE 6-15
ordered result sets 6-7
RANK 6-19
RATIOTOREPORT 6-22
TERTILE 6-15, 6-24
WHEN clause 7-54
RISQL extensions
overview 1-11
role-based security 1-9
roles
assigning comments 8-21
creating 8-125
dropping 8-177
granting 8-192, 8-199
naming 2-3
revoking 8-216
row IDs
altering segments 8-27
row subqueries 7-61
ROWCOUNT parameter 9-47
rows
limiting display with SET
ROWCOUNT 9-46
selecting from tables 7-26
ROWS_PER_FETCH_TASK
parameter
syntax and usage 9-49
ROWS_PER_JOIN_TASK
parameter
syntax and usage 9-49
ROWS_PER_SCAN_TASK
parameter
syntax and usage 9-48
ROWS_PER_TASK
parameters 9-18, 9-48
RTRIM function 5-42
running totals, calculating 6-5

S

sample database 1-14
scalar functions 5-3
ABS 5-15
CASE 5-4
CEIL 5-17
COALESCE 5-8
CONCAT 5-36, 5-37, 5-39
CURRENT_DATE 5-56
CURRENT_TIME 5-56
CURRENT_TIMESTAMP 5-56
CURRENT_USER 5-67
DATE 5-57
DATEADD 5-58
DATEDIFF 5-59
DATENAME 5-61
DEC 5-19, 5-20
DECODE 5-9
EXTRACT 5-63
FLOAT 5-22
FLOOR 5-23
IFNULL 5-11
INT 5-25, 5-27, 5-31
LOWER 5-40
LTRIM 5-41
nested 3-6, 5-3
nested inside RISQL display
functions 5-26
NULLIF 5-13
REAL 5-28
results affected by territory 5-55
RTRIM 5-42
SIGN 5-29
STRING 5-44
SUBSTR 5-47, 5-49, 5-50
TIME 5-65
TIMESTAMP 5-66
TRIM 5-51
UPPER 5-52
scalar subqueries 7-61
scale
decimal and numeric
datatype 2-24
decimal constant 2-15
double precision 2-25
integer constant 2-14
numeric datatypes 2-10
schemas
flexibility of design 1-11
star 1-11
scoping, *See* resolution of column
references
search conditions
definition 3-19
FROM clause 7-11

HAVING clause 7-32
WHEN clause 7-34
WHERE clause 7-26
security 1-9
SEGMENT BY HASH clause, CREATE TABLE command 8-148
SEGMENT BY VALUES OF clause
CREATE INDEX command 8-110
SEGMENT BY VALUES OF clause, CREATE TABLE command 8-150
segment IDs
altering segments 8-27
SEGMENT LIKE DATA clause
CREATE INDEX command 8-107
SEGMENT LIKE DATA clause, CREATE TABLE command 8-107, 8-149
SEGMENT LIKE REFERENCED TABLE clause 8-108
segment range specification
BTREE and TARGET indexes
STAR indexes 8-111
segment ranges
altering for STAR indexes 8-27
segment specification
CREATE INDEX command 8-106
segmenting columns
ALTER SEGMENT command 8-37
segments
altering 8-22
backup 8-6, 8-30, 8-178
changing maximum number 8-66
clearing 8-42
comments 8-45
creating 8-126
damaged 8-182
detaching 8-32
dropping 8-178, 9-50
hashing 8-148
keeping 9-50
maximum number per table 8-134
modifying 8-30 to 8-47
moving entire 8-44
naming 8-107, 8-127
OFFLINE mode 8-39
ONLINE mode 8-41
partially available indexes 9-26
partially available tables 9-38
physical storage units 8-43, 8-45, 8-127
ranges 8-26, 8-38, 8-110, 8-112
renaming 8-43
segment range specification 8-109, 8-149
segmenting column 8-150
SET SEGMENTS command 9-50
setting default directories 9-11, 9-13
setting intact 8-36
specifying ranges 8-151
verifying 8-36
segments, naming 2-3
SELECT privilege 8-204
SELECT statement
BREAK BY subclause 7-55
EXCEPT operator 7-36
FROM clause 7-21
GROUP BY clause 7-29
HAVING clause 7-32
INTERSECT operator 7-36
joined tables 7-22
ORDER BY clause 7-47
processing sequence 7-59
RESET BY subclause 7-52
select list 7-17
subqueries 7-60
SUPPRESS BY clause 7-58
syntax description 7-46, 8-222
syntax summary A-27
UNION operator 7-36 to 7-45
WHEN clause 7-34
WHERE clause 7-26
self-joins
correlation names in 7-15
described 7-23
SET ADVISOR LOGGING command
syntax description 9-6
syntax summary A-29
SET ARITHIGNORE, ARITHABORT command
syntax description 9-7
syntax summary A-30
SET AUTO INVALIDATE PRECOMPUTED VIEWS command
syntax summary 9-8, A-30
SET clause, UPDATE command 8-224
SET commands
ADVISOR LOGGING 9-6
ARITHIGNORE, ARITHABORT 9-7
AUTO INVALIDATE PRECOMPUTED VIEWS 9-8
COUNT RESULT 9-9
CROSS JOIN 9-10
DEFAULT DATA SEGMENT 9-11
DEFAULT INDEX SEGMENT 9-13
FIRST DAYOFWEEK 9-17
FORCE_AGGREGATION_TASKS 9-24
FORCE_FETCH_TASKS 9-19, 9-21
FORCE_HASHJOIN_TASKS 9-19, 9-23
FORCE_JOIN_TASKS 9-19, 9-21
FORCE_SCAN_TASKS 9-19, 9-20
IGNORE OPTICAL INDEXES 9-25
IGNORE PARTIAL INDEXES 9-26
INDEX TEMPSPACE 9-27
INFO MESSAGE LIMIT 9-31
LOCK 9-32
OPTICAL AVAILABILITY 9-34
ORDER BY 9-36
PARALLEL_HASHJOIN 9-37
PARTIAL AVAILABILITY 9-38
PRECOMPUTED VIEW QUERY REWRITE 9-40
PRECOMPUTED VIEW view_name 9-39
PRECOMPUTED VIEWS FOR detail_table 9-41
QUERY MEMORY LIMIT 9-41
QUERY TEMPSPACE 9-27
QUERYPROCS 9-43
REPORT_INTERVAL 9-44
RESULT BUFFER 9-45

RESULT BUFFER FULL ACTION 9-45
ROWCOUNT 9-46
ROWS_PER_FETCH_TASK 9-49
ROWS_PER_JOIN_TASK 9-49
ROWS_PER_SCAN_TASK 9-48
SEGMENTS 9-50
STATS 9-52
STATS INFO 8-186
TEMPORARY SEGMENT STORAGE PATH 9-54
UNIFORM PROBABILITY FOR ADVISOR 9-56
USE INVALID PRECOMPUTED VIEWS 9-57
VERSIONING 9-59

SET commands, SQL
TRANSACTION ISOLATION LEVEL 9-55

SET COUNT RESULT command
syntax description 9-9
syntax summary A-30

SET CROSS JOIN command
syntax description 9-10
syntax summary A-30

SET DATEFORMAT command C-2

SET DEFAULT clause, ALTER TABLE command 8-74

SET DEFAULT DATA SEGMENT command
syntax description 9-11
syntax summary A-31
usage 9-12

SET DEFAULT INDEX SEGMENT command
syntax description 9-13
syntax summary A-30, A-31
usage 9-14

SET EXPORT_DEFAULT_PATH
syntax summary A-31

SET EXPORT_DEFAULT_PATH command
syntax description 9-15

SET EXPORT_MAX_FILE_SIZE command
syntax description 9-16
syntax summary A-31

SET FIRST DAYOFWEEK command
syntax description 9-17
syntax summary A-32

SET FORCE_AGGREGATION_TASKS command
description 9-24
syntax description 9-18

SET FORCE_FETCH_TASKS command
description 9-21
syntax description 9-18
syntax summary A-32

SET FORCE_HASHJOIN_TASKS command
description 9-23
syntax description 9-18
syntax summary A-32

SET FORCE_JOIN_TASKS command
description 9-21
syntax description 9-18
syntax summary A-32

SET FORCE_SCAN_TASKS command
description 9-20
syntax description 9-18
syntax summary A-32

set functions 8-161
ALL keyword 4-4
AVG 4-4
COUNT 4-6
COUNT(*) 4-6
described 4-3
DISTINCT keyword 4-4
in a select list 7-20
MAX 4-8
MIN 4-9
nesting 3-6, 4-3
SUM 4-10

SET IGNORE OPTICAL INDEXES command
syntax description 9-25
syntax summary A-32

SET IGNORE PARTIAL INDEXES command
syntax description 9-26
syntax summary A-33
usage 9-26

SET INDEX TEMPSPACE command
syntax description 9-28
syntax summary A-33

SET INFO MESSAGE LIMIT command
syntax description 9-31
syntax summary A-33

SET LOCK command
syntax description 9-32
syntax summary A-34

SET OPTICAL AVAILABILITY command
syntax description 9-34
syntax summary A-34

SET ORDER BY command
NULL evaluation 7-49
syntax description 9-36
syntax summary A-34
usage 9-36

SET PARALLEL_HASHJOIN command
syntax description 9-37
syntax summary A-34
usage 9-37

SET PARTIAL AVAILABILITY command
syntax description 9-38
syntax summary A-35
usage 9-39

SET PRECOMPUTED VIEW QUERY REWRITE command
syntax description 9-40
syntax summary A-35

SET PRECOMPUTED VIEW view_name
syntax description 9-40
syntax summary A-35

SET PRECOMPUTED VIEWS FOR detail_table command
syntax description 9-41
syntax summary A-35

SET PRIORITY clause, ALTER SYSTEM command 8-58
SET QUERY MEMORY LIMIT command
  syntax description 9-42
  syntax summary A-36
SET QUERY TEMPSPACE command
  syntax description 9-28
  syntax summary A-33
SET QUERYPROCS command
  syntax description 9-43
  syntax summary A-36
  usage 9-43
SET REPORT_INTERVAL command
  syntax description 9-44
  syntax summary A-36
  usage 9-44
SET RESULT BUFFER command
  syntax description 9-45
  syntax summary A-36
SET RESULT BUFFER FULL ACTION command
  syntax description 9-45
  syntax summary A-37
SET ROWCOUNT command
  syntax description 9-46
  syntax summary A-37
SET ROWS_PER_FETCH_TASK command
  syntax description 9-50
  syntax summary A-37
SET ROWS_PER_JOIN_TASK command
  syntax description 9-50
  syntax summary A-37
SET ROWS_PER_SCAN_TASK command
  syntax description 9-49
  syntax summary A-37
SET SEGMENTS command
  syntax description 9-51
  syntax summary A-37
  usage 9-51
SET STATS command
  example 9-53
  syntax description 9-52
  syntax summary A-38
SET TEMPORARY SEGMENT STORAGE PATH command
  syntax description 9-54, 9-55
  syntax summary A-38
SET UNIFORM PROBABILITY FOR ADVISOR command
  syntax description 9-57
  syntax summary A-38
SET USE INVALID PRECOMPUTED VIEWS command
  syntax description 9-55, 9-57
  syntax summary A-38
SET VERSIONING command
  syntax description 9-59
SIGN function 5-29
simple expressions 3-4
simple star schema 8-105
simple tables 7-18
single quote, character literal 2-11
SMALLINT datatype 2-23
software dependencies 4
SOME quantifier 3-11
sorting rows in a table 7-47
spill files 9-27
SQL commands
  ALTER DATABASE 8-6
  ALTER INDEX 8-14
  ALTER MACRO 8-19
  ALTER ROLE 8-21
  ALTER SEGMENT 8-22
  ALTER SYNONYM 8-50
  ALTER SYSTEM 8-51
  ALTER TABLE 8-62, 8-137
  ALTER USER 8-83
  ALTER VIEW 8-84
  CREATE HIERARCHY 8-92
  CREATE INDEX 8-96
  CREATE MACRO 8-118
  CREATE ROLE 8-125
  CREATE SEGMENT 8-126
  CREATE SYNONYM 8-131
  CREATE TABLE 8-132, 8-137
  CREATE TEMPORARY TABLE 8-154
  CREATE VIEW 8-157
  DELETE 8-166
  DROP HIERARCHY 8-171
  DROP INDEX 8-172
  DROP MACRO 8-175
  DROP ROLE 8-177
  DROP SEGMENT 8-178
  DROP SYNONYM 8-180
  DROP TABLE 8-181
  DROP VIEW 8-183
  EXPAND 8-184
  EXPLAIN 8-185
  GRANT authorization and role 8-192
  GRANT CONNECT 8-198
  GRANT privilege 8-202
  INSERT 8-205
  LOCK DATABASE 8-214
  LOCK table 8-212
  REVOKE authorization and role 8-216
  REVOKE CONNECT 8-218
  REVOKE privilege 8-219
  SELECT 7-46, 8-222
  UNLOCK DATABASE 8-223
  UNLOCK table 8-222
  UPDATE 8-224
SQL Server date formats C-1
STAR indexes
  altering segments 8-27
  examples of creation 8-105
  segment range specification 8-111
  segmenting 8-109
star schemas 1-11
START ACCOUNTING clause, ALTER SYSTEM command 8-61
START LOGGING clause, ALTER SYSTEM command 8-60
START, STOP ADVISOR LOGGING clause, ALTER SYSTEM command 8-54
STOP ACCOUNTING clause, ALTER SYSTEM command 8-61
STOP LOGGING clause, ALTER SYSTEM command 8-60
storage specification of segments 8-45, 8-127
STRING function 5-44

string scalar functions
  CONCAT 5-36, 5-37, 5-39
  LOWER 5-40
  LTRIM 5-41
  RTRIM 5-42
  STRING 5-44
  SUBSTR 5-47, 5-49, 5-50
  TRIM 5-51
  UPPER 5-52
subqueries
  as derived tables 7-22
  column name resolution 7-66
  correlated and GROUP BY 7-67
  examples 7-25
  FROM clause 7-15, 7-63
  partial SELECT statement 7-60
  scalar and table 7-61
  select-list 7-19
SUBSTR function 5-47, 5-49, 5-50
subtotals 7-55
SUM function 4-10, 8-161
SUMMING clause, BREAK BY 7-55
support
  technical 11
SUPPRESS BY clause 7-58
SWITCH ACCOUNTING FILE clause, ALTER SYSTEM command 8-61
SWITCH ADVISOR LOG FILE clause, ALTER SYSTEM command 8-54
SWITCH LOGGING clause, ALTER SYSTEM command 8-60
Sybase-style syntax, for outer joins 7-28
synonyms
  altering references to 8-80
  assigning comments 8-50
  creating 8-131
  dropping 8-180
  foreign key references to 8-80
  naming 2-3, 8-131
syntax diagrams
  conventions for 7
  keywords in 9
  variables in 9
syntax notation 5
system catalog 1-10
system requirements
  database 4
  software 4
system roles
  granting 8-192, 8-199
  list of task authorizations 8-194
  revoking 8-216
system tables, joining 7-22, 7-26

T

table expressions, *See* query expressions
table references
  derived tables 7-22
  FROM clause 7-22
  syntax 7-13, 7-21
table subqueries
  defined 7-61
tables
  aggregate 8-162, 8-165, 8-184
  altering 8-62 to 8-74
  alternative table names 7-14
  comments 8-66
  creating 8-132
  creating in named segments 8-145 to 8-148
  derived 7-22
  dropping 8-181
  joining 7-6, 7-22
  naming 2-3
  referenced
  referencing 1-5
  relational database 1-4
  system 1-10
  temporary 8-154, 9-54
tablespaces, *See* segments
TARGET indexes
  altering 8-18
  CREATE INDEX command 8-98
  creating 8-98
  domain sizes 8-117
  domains, defined 8-98
  mixed-domain 8-117
  segment range specification
  unload operations 8-98
target value, DECODE function 5-9
task authorizations
  definitions 8-194
  granting 8-192
  required for SQL execution 1-9
  revoking 8-216
tasks for parallel queries, allocating 9-19 to 9-22
technical support 11
temporary tables
  creating 8-154
TERMINATE LOGGING DAEMON clause, ALTER SYSTEM command 8-60
territory
  effect on results of scalar functions 5-55
TERTILE function 6-15, 6-24
three-valued logic 2-26
THRESHOLD value
  syntax 9-29
TIME datatype 2-21
TIME function 5-65
time literals
  ANSI SQL-92 standard 2-12
  non-standard C-7
TIMESTAMP datatype 2-21
TIMESTAMP function 5-66
timestamp literals
  ANSI SQL-92 standard 2-12
  non-standard C-8
TINYINT datatype 2-23
tip icons 10
top-ten ranks 6-19
TOTALQUERYPROCS and QUERYPROCS parameters 9-20 to 9-23
TRIM function 5-51
troubleshooting
  general problems 12

U

unary operators 3-7
underscore wildcard 3-17
UNION, INTERSECT, and EXCEPT expressions 7-36 to 7-45, 7-49

UNIQUE keyword
ALTER TABLE command 8-69
CREATE TABLE command 8-137
unknown values, evaluation 2-26
unload operations, and TARGET indexes 8-98
UNLOCK DATABASE command
syntax description 8-223
syntax summary A-27
UNLOCK table command
syntax description 8-222
syntax summary A-27
UPDATE command 8-224
assignment of values 2-27
example 8-227
syntax description 8-224
syntax summary A-28
usage 8-226
UPDATE privilege 8-204
UPPER function 5-52
uppercase and lowercase 2-6, 2-11
USER function 5-67
user-created roles
creating 8-125
dropping 8-177
granting 8-192
revoking 8-216
usernames, naming 2-3
users
adding to a database 8-198
assigning comments 8-83
assigning session priority 8-83
changing passwords 8-198
dropping from a database 8-218
users, types of 3
USING subclause, in FROM clause 7-23

V

VALUES clause, INSERT command 8-207
variables
in syntax diagrams 9
VERIFY clause, ALTER SEGMENT command 8-36
views 1-6
assigning comments 8-84
creating 8-157
dropping 8-183
naming 2-3
precomputed 8-92, 8-158, 8-163, 8-165, 9-8, 9-41, 9-57, A-30, A-35, A-38

W

WAIT keyword, SET LOCK command 9-32
warning icons 10
WHEN clause
description 7-34
RISQL display functions 7-54
WHERE clause 7-26
conditions in 3-10
inner joins in 7-27
wildcard characters 3-17
WITH FILLFACTOR clause, CREATE INDEX command 8-116
working memory limit 9-41
writing queries 7-3 to 7-68

Symbols

%, SQL wildcard character 3-17
*, abbreviation in select list 7-18
_, SQL wildcard character 3-17