UniVerse

SQL User Guide

Version 10.3
February, 2009

IBM Corporation
555 Bailey Avenue
San Jose, CA 95141

Documentation Team: Claire Gustafson, Shelley Thompson, Anne Waite

Table of Contents

Chapter 5 Modifying Data

Chapter 6 Establishing and Using Views

Preface

This manual is for application developers and system administrators who are familiar with UniVerse and want to use the additional functionality of SQL in their UniVerse applications.

This document uses a multilayered approach. It starts by discussing how to *query* an existing, up-to-date database. Then it discusses how to *modify* the database by adding, deleting, and changing rows of data. Interspersed among these topics are discussions of primary keys, data constraints, referential integrity, and transaction processing.

This manual does not cover the syntax of SQL statements, nor the rules for forming table and column names. You can find this and related information in the *UniVerse SQL Reference*.

Organization of This Manual

This manual contains the following:

Chapter 1, "Understanding SQL Concepts," introduces SQL, compares UniVerse and SQL databases, and describes the sample database used throughout this document.

Chapter 2, "Using SELECT Statements," introduces the UniVerse SQL SELECT statement and shows how to select information from a single table or file, use expressions and set functions, and format output.

Chapter 3, "Using Advanced SELECT Statements," describes more advanced forms of the SELECT statement, including GROUP BY and HAVING clauses, table joins, and subqueries.

Chapter 4, "Selecting on Multivalued Columns," describes how to use UniVerse SQL to access and manipulate data stored in UniVerse's multivalued columns and use dynamic normalization.

Chapter 5, "Modifying Data," covers how to use UniVerse SQL statements to add, update, and delete data stored in tables and files.

Chapter 6, "Establishing and Using Views," discusses the application and manipulation of table views.

Appendix A, "The Sample Database," contains the CREATE TABLE statements used to create the tables in the sample database.

The Glossary defines common UniVerse SQL terms.

Documentation Conventions

This manual uses the following conventions:

Convention	Usage
Bold	In syntax, bold indicates commands, function names, and options. In text, bold indicates keys to press, function names, menu selections, and MS-DOS commands.
UPPERCASE	In syntax, uppercase indicates UniVerse commands, keywords, and options; UniVerse BASIC statements and functions; and SQL statements and keywords. In text, uppercase also indicates UniVerse identifiers such as file names, account names, schema names, and Windows file names and paths.
Italic	In syntax, italic indicates information that you supply. In text, italic also indicates UNIX commands and options, file names, and paths.
`Courier`	Courier indicates examples of source code and system output.
`Courier Bold`	In examples, courier bold indicates characters that the user types or keys the user presses (for example, **`<Return>`**).
[ ]	Brackets enclose optional items. Do not type the brackets unless indicated.
{ }	Braces enclose nonoptional items from which you must select at least one. Do not type the braces.
itemA | itemB	A vertical bar separating items indicates that you can choose only one item. Do not type the vertical bar.
...	Three periods indicate that more of the same type of item can optionally follow.
ä	A right arrow between menu options indicates you should choose each option in sequence. For example, "Choose **File** ä **Exit**" means you should choose **File** from the menu bar, then choose **Exit** from the File pull-down menu.
`I`	Item mark. For example, the item mark (`I`) in the following string delimits elements 1 and 2, and elements 3 and 4: `1I2F3I4V5`

Documentation Conventions

Convention	Usage
F	Field mark. For example, the field mark (**F**) in the following string delimits elements FLD1 and VAL1: `FLD1`**`F`**`VAL1`**`V`**`SUBV1`**`S`**`SUBV2`
V	Value mark. For example, the value mark (**V**) in the following string delimits elements VAL1 and SUBV1: `FLD1`**`F`**`VAL1`**`V`**`SUBV1`**`S`**`SUBV2`
S	Subvalue mark. For example, the subvalue mark (**S**) in the following string delimits elements SUBV1 and SUBV2: `FLD1`**`F`**`VAL1`**`V`**`SUBV1`**`S`**`SUBV2`
T	Text mark. For example, the text mark (**T**) in the following string delimits elements 4 and 5: `1`**`F`**`2`**`S`**`3`**`V`**`4`**`T`**`5`

Documentation Conventions (Continued)

The following conventions are also used:

- Syntax definitions and examples are indented for ease in reading.
- All punctuation marks included in the syntax—for example, commas, parentheses, or quotation marks—are required unless otherwise indicated.
- Syntax lines that do not fit on one line in this manual are continued on subsequent lines. When entering syntax, type the entire syntax entry, including the continuation lines, on the same input line.

UniVerse Documentation

UniVerse documentation includes the following:

UniVerse Installation Guide: Contains instructions for installing UniVerse 10.3.

UniVerse New Features Version 10.3: Describes enhancements and changes made in the UniVerse 10.3 release for all UniVerse products.

UniVerse BASIC: Contains comprehensive information about the UniVerse BASIC language. It is for experienced programmers.

UniVerse BASIC Commands Reference: Provides syntax, descriptions, and examples of all UniVerse BASIC commands and functions.

UniVerse BASIC Extensions: Describes the following extensions to UniVerse BASIC: UniVerse BASIC Socket API, Using CallHTTP, and Using WebSphere MQ with UniVerse.

UniVerse BASIC SQL Client Interface Guide: Describes how to use the BASIC SQL Client Interface (BCI), an interface to UniVerse and non-UniVerse databases from UniVerse BASIC. The BASIC SQL Client Interface uses ODBC-like function calls to execute SQL statements on local or remote database servers such as UniVerse, DB2, SYBASE, or INFORMIX. This book is for experienced SQL programmers.

Administering UniVerse: Describes tasks performed by UniVerse administrators, such as starting up and shutting down the system, system configuration and maintenance, system security, maintaining and transferring UniVerse accounts, maintaining peripherals, backing up and restoring files, and managing file and record locks, and network services. This book includes descriptions of how to use the UniAdmin program on a Windows client and how to use shell commands on UNIX systems to administer UniVerse.

Using UniAdmin: Describes the UniAdmin tool, which enables you to configure UniVerse, configure and manage servers and databases, and monitor UniVerse performance and locks.

UniVerse Security Features: Describes security features in UniVerse, including configuring SSL through UniAdmin, using SSL with the CallHttp and Socket interfaces, using SSL with UniObjects for Java, and automatic data encryption.

UniVerse Transaction Logging and Recovery: Describes the UniVerse transaction logging subsystem, including both transaction and warmstart logging and recovery. This book is for system administrators.

UniVerse System Description: Provides detailed and advanced information about UniVerse features and capabilities for experienced users. This book describes how to use UniVerse commands, work in a UniVerse environment, create a UniVerse database, and maintain UniVerse files.

UniVerse User Reference: Contains reference pages for all UniVerse commands, keywords, and user records, allowing experienced users to refer to syntax details quickly.

Guide to RetrieVe: Describes RetrieVe, the UniVerse query language that lets users select, sort, process, and display data in UniVerse files. This book is for users who are familiar with UniVerse.

Guide to ProVerb: Describes ProVerb, a UniVerse processor used by application developers to execute prestored procedures called procs. This book describes tasks such as relational data testing, arithmetic processing, and transfers to subroutines. It also includes reference pages for all ProVerb commands.

Guide to the UniVerse Editor: Describes in detail how to use the Editor, allowing users to modify UniVerse files or programs. This book also includes reference pages for all UniVerse Editor commands.

UniVerse NLS Guide: Describes how to use and manage UniVerse's National Language Support (NLS). This book is for users, programmers, and administrators.

UniVerse SQL Administration for DBAs: Describes administrative tasks typically performed by DBAs, such as maintaining database integrity and security, and creating and modifying databases. This book is for database administrators (DBAs) who are familiar with UniVerse.

UniVerse SQL User Guide: Describes how to use SQL functionality in UniVerse applications. This book is for application developers who are familiar with UniVerse.

UniVerse SQL Reference: Contains reference pages for all SQL statements and keywords, allowing experienced SQL users to refer to syntax details quickly. It includes the complete UniVerse SQL grammar in Backus Naur Form (BNF).

Related Documentation

The following documentation is also available:

UniVerse GCI Guide: Describes how to use the General Calling Interface (GCI) to call subroutines written in C, C++, or FORTRAN from UniVerse BASIC programs. This book is for experienced programmers who are familiar with UniVerse.

UniVerse ODBC Guide: Describes how to install and configure a UniVerse ODBC server on a UniVerse host system. It also describes how to use UniVerse ODBC Config and how to install, configure, and use UniVerse ODBC drivers on client systems. This book is for experienced UniVerse developers who are familiar with SQL and ODBC.

UV/Net II Guide: Describes UV/Net II, the UniVerse transparent database networking facility that lets users access UniVerse files on remote systems. This book is for experienced UniVerse administrators.

UniVerse Guide for Pick Users: Describes UniVerse for new UniVerse users familiar with Pick-based systems.

Moving to UniVerse from PI/open: Describes how to prepare the PI/open environment before converting PI/open applications to run under UniVerse. This book includes step-by-step procedures for converting INFO/BASIC programs, accounts, and files. This book is for experienced PI/open users and does not assume detailed knowledge of UniVerse.

API Documentation

The following books document application programming interfaces (APIs) used for developing client applications that connect to UniVerse and UniData servers.

Administrative Supplement for Client APIs: Introduces IBM's seven common APIs, and provides important information that developers using any of the common APIs will need. It includes information about the UniRPC, the UCI Config Editor, the *ud_database* file, and device licensing.

UCI Developer's Guide: Describes how to use UCI (Uni Call Interface), an interface to UniVerse and UniData databases from C-based client programs. UCI uses ODBC-like function calls to execute SQL statements on local or remote UniVerse and UniData servers. This book is for experienced SQL programmers.

IBM JDBC Driver for UniData and UniVerse: Describes UniJDBC, an interface to UniData and UniVerse databases from JDBC applications. This book is for experienced programmers and application developers who are familiar with UniData and UniVerse, Java, JDBC, and who want to write JDBC applications that access these databases.

InterCall Developer's Guide: Describes how to use the InterCall API to access data on UniVerse and UniData systems from external programs. This book is for experienced programmers who are familiar with UniVerse or UniData.

UniObjects Developer's Guide: Describes UniObjects, an interface to UniVerse and UniData systems from Visual Basic. This book is for experienced programmers and application developers who are familiar with UniVerse or UniData, and with Visual Basic, and who want to write Visual Basic programs that access these databases.

UniObjects for Java Developer's Guide: Describes UniObjects for Java, an interface to UniVerse and UniData systems from Java. This book is for experienced programmers and application developers who are familiar with UniVerse or UniData, and with Java, and who want to write Java programs that access these databases.

UniObjects for .NET Developer's Guide: Describes UniObjects, an interface to UniVerse and UniData systems from .NET. This book is for experienced programmers and application developers who are familiar with UniVerse or UniData, and with .NET, and who want to write .NET programs that access these databases.

Using UniOLEDB: Describes how to use UniOLEDB, an interface to UniVerse and UniData systems for OLE DB consumers. This book is for experienced programmers and application developers who are familiar with UniVerse or UniData, and with OLE DB, and who want to write OLE DB programs that access these databases.

Chapter

1

Understanding SQL Concepts

This chapter includes an introduction to SQL, a discussion of the differences between UniVerse and SQL concepts, some important database terms, and a description of the sample database (called Circus) used in examples in this manual.

Introduction to SQL

SQL is a popular relational database language. It is not a database management system or a stand-alone product. SQL is a part of many database management systems, and over the past decade it has become the standard database language.

UniVerse SQL follows the ANSI/ISO 1989 standard with extensions to accommodate multivalued fields and other features unique to UniVerse.

The implementation of SQL in UniVerse adds a new level of capability to UniVerse's many features. SQL-related enhancements include:

- Subquerying, which allows you to nest queries
- Relational joining, which allows you to work with data from more than one file or table in a single command or statement
- Database security, for added protection of your data
- Database integrity, to prevent writing invalid data to your database

All SQL features are integrated seamlessly into the UniVerse system without losing or compromising UniVerse's inherent capabilities. Although SQL began as a user-friendly language for querying databases ad hoc, it in fact has many important uses in the UniVerse world, as summarized in the following table. SQL uses terms that differ from those used in UniVerse, both in this text and when dealing with SQL in general. See the Glossary for common SQL terms.

Function	Description
Interactive access	Use SQL to directly query and update your UniVerse files and SQL tables in an ad hoc fashion.
Database programming	Include SQL statements within application programs to access data in UniVerse files and SQL tables.
Database administration	Use SQL to define database structures and impose security and integrity constraints on the data.
Client/server	Use SQL to communicate with database servers over a local area network.

Overview of UniVerse SQL

Overview of Databases, Files, and Tables

To the UniVerse user planning to use SQL, there are more similarities between UniVerse files and SQL tables than there are differences. However, the distinctions are important.

UniVerse and SQL Databases

Comparing UniVerse with conventional SQL at the database level involves two major areas: the concept and structure of the database itself, and the data model on which it is based. These differences are summarized in the following table and then discussed in greater detail.

	Traditional UniVerse Databases	UniVerse SQL Databases
Located in:	An account	A schema
Created by:	—	CREATE SCHEMA
Described in:	VOC file	SQL catalog
Contains:	One or more UniVerse files	One or more SQL tables, UniVerse files, or both
Data model:	Nonfirst-normal form/ postrelational	First normal form/ relational and nonfirst-normal form/postrelational

Comparison of Traditional UniVerse Databases to SQL Databases

Database Concepts and Structures

In the traditional (non-SQL) UniVerse environment, a database is loosely defined as being “one or more UniVerse files.” The database evolves as those files are created; there is no single command or process for creating a UniVerse database. Generally, such files reside in a single account.

UniVerse SQL associates a database with a schema, which is created using a CREATE SCHEMA statement, and defines that database in the SQL catalog tables. In UniVerse SQL, a database comprises one or more SQL tables or UniVerse files, or both.

Data Models

UniVerse uses a three-dimensional file structure, commonly referred to as a nonfirst-normal-form (NF^2) data model to store multivalued fields. This enables a single file (table) to contain the information that would otherwise be scattered among several interrelated files (tables). Related multivalued columns can be grouped together in an association, which can be thought of as a "table within a table," or nested table.

Conventional SQL uses a two-dimensional table structure called a first normal form (1NF). Instead of using multivalued fields, it tends to use smaller tables that are related to one another by common key values. However, the UniVerse implementation of SQL has added enhancements that allow you to store and process multivalued fields.

The implications of these differences in data modeling and the relational design of SQL are discussed further under "File and Table Structures" on page 7.

UniVerse Files and SQL Tables

UniVerse files and SQL tables share much in common. In fact, SQL tables are implemented as UniVerse files and can be accessed by UniVerse commands.

- The SQL statement to create a table, CREATE TABLE, functions like the UniVerse CREATE.FILE command.
- Each UniVerse file or SQL table is actually two files: a data file and a file dictionary.
- The data structures of files and tables are comparable, although UniVerse files commonly are described as containing fields and records, and SQL tables as containing columns and rows. Under the UniVerse implementation of SQL, tables can contain multivalued columns (fields).
- Both UniVerse files and SQL tables can be accessed using either UniVerse commands and processes or SQL statements.

UniVerse files and SQL tables also differ in some respects. A comparison is summarized in the following table.

	Traditional UniVerse Files	SQL Tables
Created by:	CREATE.FILE	CREATE TABLE
Removed by:	DELETE.FILE	DROP TABLE
Components:	Data file + file dictionary.	Data table + table dictionary. A security and integrity constraints area (SICA) in the data table allows establishment and maintenance of data structure, permissions, and integrity constraints.
Structure:	Fields and records.	Columns and rows.
Accessed by:	UniVerse commands (such as RetrieVe, ReVise), UniVerse BASIC, UniVerse Editor, and other processes, and SQL statements.	UniVerse commands (such as RetrieVe, ReVise), UniVerse BASIC, UniVerse Editor, and other processes, and SQL statements.
Security:	Permissions (read/write) granted and revoked by owners/groups/others.	In addition to operating system permissions, more extensive privileges—SELECT, INSERT, UPDATE, DELETE—on tables, plus DBA Privilege and RESOURCE Privilege, may be granted or revoked.
Data integrity:	Checked during certain conversions.	Integrity constraints can be defined, which will be enforced for all attempted writes.
Primary keys:	CREATE.FILE allows for only single-column record IDs.	CREATE TABLE allows for both single- and multicolumn primary keys.
SQL Data Types	Not native to UniVerse, but certain output conversion and formatting codes can be included in a field definition.	An essential part of column definitions, and associated with precise default characteristics such as a restricted character set, alignment, etc.

Comparison of Traditional UniVerse Files to SQL Tables

File and Table Structures

UniVerse is a nonfirst-normal-form database that permits multivalued fields (a row-and-column position that can hold more than one data value). SQL works with first-normal-form databases, which store only one value for every row and column (singl-evalued fields), but in the UniVerse implementation, SQL can store and process multivalued fields also.

SQL is relationally oriented, and allows you to access multiple tables by joining them on common values (or keys), as if they were one table. For example, using SQL, a retailer can inquire about an inventory item (in an INVENTORY table) and its supplier (in a DISTRIBUTOR table), provided that the INVENTORY table has a "distributor code" column that can be used to join it to the DISTRIBUTOR table.

UniVerse without SQL is designed primarily for accessing one file at a time, although you can extract information from a second file, using the TRANS function or the T*file* correlative, to obtain a similar result. But with the SQL enhancement, you can use a SELECT statement to join multiple tables *and* UniVerse files in any combination.

Security and Authorization

In addition to UniVerse's security provisions (controlling read/write access to files), SQL allows you to grant or revoke privileges based on user, table, and operation (retrieving or selecting data, and inserting, modifying, and deleting rows).

SQL also provides three levels of user authority. From the lowest to the highest, they are as follows:

- CONNECT allows you to create your own tables and do whatever you want with them (including granting your "owner" privileges to other users).
- RESOURCE allows you to create a schema and assign ownership to it (plus do everything allowed under CONNECT).
- DBA (a sort of "superuser" level) allows you to do everything, including reading or writing to anyone else's tables.

Data Integrity

In UniVerse, data integrity is provided by certain conversion operations (such as date conversions) that flag illegal values by returning an error STATUS code. SQL has many additional data integrity constraints, including referential integrity and checks for nulls, empty columns, nonunique values, and value ranges.

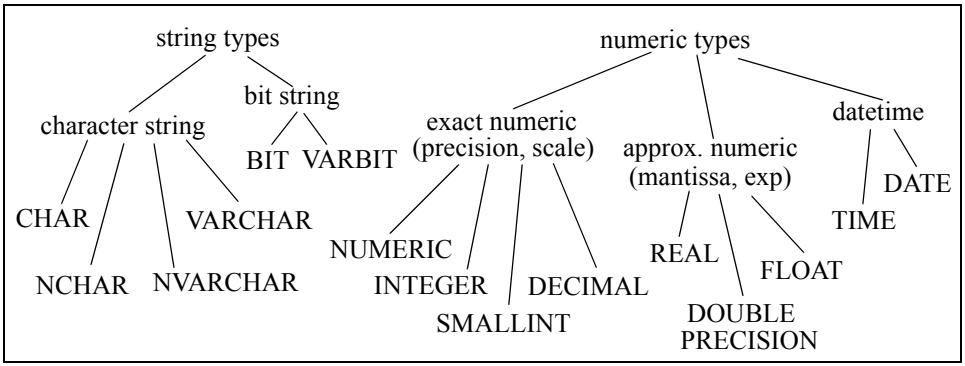
Primary Keys

The UniVerse file structure has a single-column primary key (record ID), whereas SQL allows for either single-column or multicolumn primary keys.

Data Types

Unlike a field in a UniVerse file, a column in an SQL table is defined as being of a particular data type. A data type defines a column in terms of the valid set of data characters that can be stored in the column, the alignment of the data, conversion characteristics, and so on. For more information, see the *UniVerse SQL Reference* or *UniVerse SQL Administration for DBAs*.

Data types are grouped into string types and numeric types as shown in the following illustration.



Grouped Data Types

The following table summarizes data types.

Data Type	Description
BIT	Stores bit strings.
CHAR	Stores character strings (any combination of numbers, letters, and special characters).
DATE	Stores dates as whole decimal numbers.
DECIMAL	Stores decimal fixed-scale (fixed-point) numbers (same as NUMERIC).
DOUBLE PRECISION	Stores high-precision floating-point numbers.
FLOAT	Stores floating-point numbers.
INTEGER	Stores whole decimal numbers.
NCHAR	Stores national character strings.
NVARCHAR	Stores variable-length national character strings.
NUMERIC	Same as DECIMAL.
REAL	Stores floating-point (real) numbers.
SMALLINT	Stores small whole decimal numbers.
TIME	Stores times as whole decimal numbers.
VARBIT	Stores variable-length bit strings.
VARCHAR	Stores variable-length character strings.

Data Types

The Sample Database

UniVerse provides a sample database called Circus that you can use to explore many of the features of UniVerse SQL. This database consists of 10 SQL tables and is designed to demonstrate the use of industry-standard SQL access with UniVerse, SQL extensions implemented for UniVerse's multivalued field associations and nested tables, and the benefits of programmable virtual fields (I-descriptors).

The CREATE TABLE statements that generated the SQL tables are in Appendix A, "The Sample Database."

Installing the Sample Database

You can install the Circus database as either an account of UniVerse files or as a schema of SQL tables (or both). The two versions of the database are distinguished by a suffix in the file name:

- .F identifies the UniVerse file version.
- .T identifies the SQL table version.

Thus, INVENTORY.F is the UniVerse file version of the inventory data, and INVENTORY.T is the SQL table version of the same data.

Install the version of the files that you prefer. Examples in this manual use the UniVerse SQL table version. Keep in mind that you can issue SQL statements against UniVerse files, and you can issue RetrieVe commands against SQL tables. However, the results may vary slightly, depending on which you use.

Use the following UniVerse commands to generate and remove the Circus database:

Command	Action
SETUP.DEMO.SCHEMA *username*	Registers *username* as an SQL user (if not one already) and makes the current UniVerse account into a schema called DEMO_*username*, which is owned by *username*. Only an SQL user who is a DBA (database administrator) can run this command.
MAKE.DEMO.TABLES	Creates and loads the Circus database tables into the current account, making the current user the owner of the tables. The user must be a registered SQL user, the account must be an SQL schema, and the tables must not already exist in this schema. Resultant tables have a .T suffix.
REMOVE.DEMO.TABLES	Deletes the Circus database tables from the current schema. The user must be a registered SQL user who is either the owner of the tables or a DBA.
MAKE.DEMO.FILES	Creates and loads the Circus database files into the current account. The files must not already exist in this account. The file names will have an .F suffix, and the contents of the files match those of the corresponding .T tables.
REMOVE.DEMO.FILES	Deletes the Circus database files from the current account.

To install the SQL table version of the Circus database on your system:

1. Create a directory to contain the Circus database and set it up as a UniVerse account.
2. If you are a registered SQL user with RESOURCE privilege, log on to the account and make the account into a schema by entering:

   >**CREATE SCHEMA *schemaname*;**

   You can use any unique *schemaname*. You are the owner of the new schema.

   If you are not a registered user with RESOURCE privilege, have your database administrator (DBA) log in to your account and enter:

   >**SETUP.DEMO.SCHEMA *username***

   username is your operating system user name. This command registers you as an SQL user, makes the directory into a schema called DEMO_*username*, and sets you up as the schema's owner.

3. To create the tables and load data into them, enter:

   >**MAKE.DEMO.TABLES**

   The table names all have the .T suffix. You are the owner of the tables.

Deinstalling the Sample Database

To deinstall the database, use either REMOVE.DEMO.TABLES (if the database is the SQL table version) or REMOVE.DEMO.FILES (if the database is the UniVerse file version). For example, to drop the SQL tables for the Circus database, enter:

```
>REMOVE.DEMO.TABLES
Dropping table constraint UVCON_2
Dropping table constraint UVCON_3
Dropping table constraint UVCON_2
.
.
.
Dropping Table LIVESTOCK.T
Dropping Table VENDORS.T

All demo tables removed.
```

***Note**: To restore the Circus database to its original state, first delete the tables or files with REMOVE.DEMO.TABLES or REMOVE.DEMO.FILES and then repeat MAKE.DEMO.TABLES or MAKE.DEMO.FILES.*

Chapter

2

Using SELECT Statements

This chapter covers the simplest forms of the SELECT statement and explains how to query a single table in various ways, including arranging columns, selecting rows, using virtual columns to hold derived results, and using qualifiers to process and format your output. More advanced discussions of the SELECT statement follow in Chapter 3, "Using Advanced SELECT Statements," and Chapter 4, "Selecting on Multivalued Columns."

The SQL Language

The SQL language, as defined in ANSI/ISO standards, is made up of many distinct statements, each communicating a specific request to the database “engine,” or core code.

Each SQL statement starts with a verb, followed by one or more clauses. Each clause starts with a keyword. UniVerse implements 17 of these verbs. The following table lists the DML (data manipulation language) statements. The next table lists the DDL (data definition language) statements.

Verb	Description
SELECT	Retrieves data from tables and UniVerse files.
INSERT	Inserts new rows into a table or UniVerse file.
UPDATE	Modifies data in a table or UniVerse file.
DELETE	Removes rows from a table or UniVerse file.

UniVerse DML Verbs

Verb	Description
ALTER TABLE	Modifies the definition of an existing base table.
CREATE INDEX	Creates a new index on a table.
CREATE SCHEMA	Creates a new schema.
CREATE TABLE	Creates a new table in a schema.
CREATE TRIGGER	Creates a trigger for a table.
CREATE VIEW	Creates a view of a table.
DROP INDEX	Delete an index from a table.
DROP SCHEMA	Deletes a schema.
DROP TABLE	Deletes a table.

UniVerse DDL Verbs

Verb	Description
DROP TRIGGER	Deletes a trigger.
DROP VIEW	Deletes a view.
GRANT	Assigns privileges on tables and views to a user.
REVOKE	Revokes previously granted privileges from a user.

UniVerse DDL Verbs (Continued)

Introduction to UniVerse SQL SELECT

SQL primarily is a database query language, and many installations use SQL almost exclusively as a database query tool. SELECT is the primary statement for querying both SQL tables and UniVerse files.

Using the Command Processor

Every statement entered at the system prompt—as well as commands entered from a stored command sequence, a proc, or a BASIC program—is examined and parsed by a system program called the command processor.

The command processor maintains a list of the most recent command lines entered at the system prompt. This list is called the *sentence stack*, and you can use it to recall, delete, change, or reexecute a previous statement, or to save a sentence or paragraph in your UniVerse VOC file. By default, the sentence stack preserves up to 99 sentences from your current session. Each sentence is numbered from 01 through 99, with 01 being the most recent.

The UniVerse command processor has a few conventions that should be familiar to you.

- Enter a statement using the processor's natural wordwrap and do not press **Enter**. However, to control how lines are broken, press **Enter** to start a new line. You get a system prompt and then can continue entering your statement. For example, you could enter the following statement in either of the ways shown:

```
>SELECT ENGAGEMENTS.T.LOCATION_CODE, "DATE", TIME, DESCRIPTION,
NAME FROM ENGAGEMENTS.T, LOCATIONS.T WHERE
ENGAGEMENTS.T.LOCATION_CODE = LOCATIONS.T.LOCATION_CODEORDER BY
ENGAGEMENTS.T.LOCATION_CODE, "DATE";<Return>
>SELECT ENGAGEMENTS.T.LOCATION_CODE, "DATE", TIME,<Return>
SQL+DESCRIPTION, NAME FROM ENGAGEMENTS.T, LOCATIONS.T<Return>
SQL+WHERE ENGAGEMENTS.T.LOCATION_CODE =<Return>
SQL+LOCATIONS.T.LOCATION_CODE<Return>
SQL+ORDER BY ENGAGEMENTS.T.LOCATION_CODE, "DATE";<Return>
```

  You can end a line with an underscore before pressing **Enter**, but it is not necessary.

- To terminate and execute a statement, do one of the following:
  - Type a **;** (semicolon) and press **Enter**:

```
>SELECT LOCATION_CODE, "DATE" FROM ENGAGEMENTS.T;<Return>
```

  - Press **Return**, then at the continuation prompt, press **Enter** again:

```
>SELECT LOCATION_CODE, "DATE" FROM ENGAGEMENTS.T<Return>
SQL+<Return>
```

- To terminate and save a statement without executing it (for example, when you notice a typo you want to correct), type a **?** (question mark) and press **Enter**.
- You can choose commands to edit, recall, insert, reexecute, or delete an SQL statement in the sentence stack:
  - .A to add text to the end of a sentence
  - .C to modify a sentence
  - .D to delete a sentence
  - .I to insert a new sentence
  - .L to list the contents of the sentence stack
  - .R to recall a sentence
  - .S to save a sentence
  - .U to convert a sentence to uppercase
  - .X to execute a sentence
  - .? to obtain help about sentence stack commands

This sequence is an example of using command processor commands. **Bold** indicates user input and *italic* denotes explanatory comments.

```
>SELECT LOCATION_CODE,<Return>
SQL+"DATEE", "TIME" ?<Return>      Notices typo; enters ?
>.C/DATEE/DATE<Return>             Corrects typo
01 SELECT LOCATION_CODE, "DATE", "TIME"
>.A FROM ENGAGEMENTS.T;<Return>    Continues statement
01 SELECT LOCATION_CODE, "DATE", "TIME" FROM ENGAGEMENTS.T;
>.X<Return>                        Executes completed statement
01 SELECT LOCATION_CODE, "DATE", "TIME" FROM ENGAGEMENTS.T;
Statement is parsed and executed
>.L<Return>                        Lists sentence stack
03 SELECT GATE_NUMBER, AVG(GATE_TICKETS)...
02 SELECT LOCATION_CODE, "DATE" FROM ENGAGEMENTS.T;
01 SELECT LOCATION_CODE, "DATE", "TIME" FROM ENGAGEMENTS.T;
>.R3<Return>                       Recalls sentence 3
03 SELECT GATE_NUMBER, AVG(GATE_TICKETS)...
>.X<Return>                        Executes recalled statement
```

```
01 SELECT GATE_NUMBER, AVG(GATE_TICKETS)...
Statement is parsed and executed
>.L<Return>                          Lists sentence stack again
04 SELECT GATE_NUMBER, AVG(GATE_TICKETS)...
03 SELECT LOCATION_CODE, "DATE" FROM ENGAGEMENTS.T;
02 SELECT LOCATION_CODE, "DATE", "TIME" FROM ENGAGEMENTS.T;
01 SELECT GATE_NUMBER, AVG(GATE_TICKETS)...
>.D3<Return>                         Deletes sentence 3 from stack
History #3 DELETEd.
>.X2<Return>                         Executes sentence 2 in stack
02 SELECT LOCATION_CODE, "DATE", "TIME" FROM ENGAGEMENTS.T;
Statement is parsed and executed
>
```

SELECT Statement Elements

The full form of the SELECT statement consists of the following elements:

Element	Description
SELECT clause	Describes columns to be retrieved, either columns from the table or file or calculated (derived) columns (mandatory).
FROM clause	Describes tables or files containing the data (mandatory).
WHERE clause	Describes the rows to be retrieved.
WHEN clause	Limits output from multivalued columns.
GROUP BY clause	Describes how the data is to be grouped or summarized.
HAVING clause	Describes which groups are to be retrieved.
ORDER BY clause	Describes how the results are to be ordered or sorted.

SELECT Statement Elements

Comparing UniVerse SQL SELECT to RetrieVe

If you are familiar with RetrieVe LIST and SORT commands, you will recognize the similarities. Either command can be used with UniVerse files and SQL tables. The following table compares the UniVerse SQL SELECT statement to the RetrieVe syntax.

Feature	SQL SELECT	RetrieVe
Source	FROM *tablelist*	file name
Columns/fields to be retrieved	List of columns	List of output fields
Virtual columns	expressions and EVAL and I-descriptors	EVAL *i.type.expr* and I-descriptors
Record specification	List of primary key values	List of records (record IDs)
Record/row selection criteria	WHERE clause	WITH clause
Multivalued column output filter	WHEN clause	WHEN clause
Sorting	ORDER BY	*field* (BY, BY.DSND, BY.EXP, or BY.EXP.DSND)
Control breaks	GROUP BY	BREAK.ON or DET.SUP
Aggregate functions	Set functions	Not supported

Comparison of UniVerse SQL SELECT to RetrieVe

Results as Tables

One unique feature of relational databases is that the results of a query are also in the form of a table that can be treated as though it were a physical table in the database. For example, if you were to select ITEM_TYPE, DESCRIPTION, and QOH from the INVENTORY.T table in the sample database, and there were 44 rows in that table, the result would be a *table* of 3 columns and 44 rows.

Thus you can query the results themselves as though they were just another database table. The usefulness of this feature is apparent when you use subqueries, which are discussed in Chapter 3, "Using Advanced SELECT Statements."

Retrieving Data from a Single Table

This section explains the various ways to retrieve data from a single table or file. Starting with a simple SELECT statement to retrieve all rows and columns of a table, you then ask for the following:

- Specific columns
- Virtual columns (derived data)
- Data sorted by rows
- Specific rows
- Summary of a table's content using set functions

Retrieving an Entire Table

The simplest form of the SELECT statement is:

SELECT *selectlist* **FROM** *tablename*

In most instances, *selectlist* is a list of the specific columns you want to see. You may want to look at every column in a table, particularly when dealing with a new database and want an idea of what it contains. Use an asterisk (*) to indicate all columns. For example, you never saw the sample database and want to see what is in the LIVESTOCK.T table. To do this, enter:

```
>SELECT * FROM LIVESTOCK.T;
ANIMAL_ID...    80
NAME........ Kungu
DESCRIPTION. Puma
USE......... Z
DOB......... 02/13/84
ORIGIN...... Chile
COST........      3940.00
EST_LIFE....  19
VAC_TYPE VAC_DATE.. VAC_NEXT.. VAC_CERT
R        03/31/93   03/30/96   892361
P        01/09/92   01/08/95   147953
L        01/05/92   01/04/95   432996

ANIMAL_ID...    24
NAME........ Warri
DESCRIPTION. Civet
USE......... Z
```

```
DOB......... 07/28/81
ORIGIN...... Pakistan
COST........     10198.00
EST_LIFE....  18
VAC_TYPE VAC_DATE.. VAC_NEXT.. VAC_CERT
Press any key to continue...
```

The four vaccination information columns are multivalued with each value listed separately.

Using the syntax as follows would achieve the same result:

>SELECT LIVESTOCK.T.* FROM LIVESTOCK.T;

***Note**: The * form of selectlist retrieves columns as specified in the table's @SELECT phrase. If there is no @SELECT phrase, columns are retrieved in the order in which they were defined in the CREATE TABLE statement. If you are selecting from a UniVerse file, "all columns" refers to the columns listed in the @SELECT phrase for the file. If there is no @SELECT phrase, "all columns" refers to the columns listed in the @ phrase, plus the primary key (unless the @ phrase contains the keyword ID.SUP). If there is no @ phrase, you get just the primary key column. If the table has no primary key, you get the @ID column.*

UniVerse SQL assumes that you are using the primary file dictionary associated with the table or file. To use an alternate file dictionary, include the USING DICT *filename* clause (as in a RetrieVe command):

>SELECT * FROM LIVESTOCK.T USING DICT LV2;

In this case, the column attributes (such as output formatting) defined in the LV2 file dictionary are applied to the data in the LIVESTOCK.T data file.

If you use just *tablename* in the FROM clause, it refers to the data file for that table. To refer to the file dictionary instead, include the DICT keyword (similar to preceding *filename* with DICT in a RetrieVe command):

```
>SELECT * FROM DICT LIVESTOCK.T;
               Type &
Field......... Field. Field..... Conversion Column...... Output
Depth &
Name.......... Number Definition Code...... Heading..... Format
Assoc..

ANIMAL_ID      D    0             MD0                     5R     S
@ID            D    0                         LIVESTOCK.T  5R     S
@KEY           PH     ANIMAL_ID

USE            D    3                                     1L     S
NAME           D    1                                     10T    S
```

```
DESCRIPTION     D     2                                    10T     S
DOB             D     4              D2/                   10L     S
ORIGIN          D     5                                    12T     S
EST_LIFE        D     7              MD0                   3R      S
COST            D     6              MD22                  12R     S
VAC_CERT        D    11                                    6L      M
VAC_ASSOC
VAC_DATE        D     9              D2/                   10L     M
VAC_ASSOC
VAC_TYPE        D     8                                    1L      M
VAC_ASSOC
VAC_NEXT        D    10            D2/                    10L     M
VAC_ASSOC
@REVISE         PH      NAME
                        DESCRIPTION
                        USE DOB
                        ORIGIN COST
                        EST_LIFE
                        VAC_TYPE
                        VAC_DATE
                        VAC_NEXT
                        VAC_CERT
@               PH      ID.SUP
```

Selecting Specific Columns

You may want to see only certain columns of a table or file. UniVerse SQL allows you to specify those column names in your SELECT statement. To see only the name and description for each animal, enter:

```
>SELECT NAME, DESCRIPTION FROM LIVESTOCK.T;
NAME.....    DESCRIPTION

Kungu        Puma
Warri        Civet
Morie        Kinkajou
Marone       Ocelot
.
.
.
Wukari       Kodkod
Press any key to continue...
```

Note that the two *columnnames* are separated by a comma in the command line. Also note that the listing is unsorted. To sort the output in a particular order, specify that order. Refer to "Sorting Output" on page 41.

To obtain the various seating capacities of the sites the circus visits, enter:

```
>SELECT SEATS FROM LOCATIONS.T;
SEATS

 3000
 1000
 6000
 6000
 6000
 .
 .
 .
 5000
Press any key to continue...
```

You get a long list of seatings with many duplicates. To see just the *different* seating capacities, use the keyword DISTINCT to eliminate duplicates:

```
>SELECT DISTINCT SEATS FROM LOCATIONS.T;
SEATS

 1000
 3000
 6000
 4000
 2000
 7000
10000
 5000
 8000

9 records listed.
```

Now, you can see clearly that you booked the show into nine different sizes of stadium, ranging from 1,000 to 10,000 seats.

You can use the CAST function to force the data type of a SELECT statement to be different than defined in the table. This can be very useful if you want to perform operations not normally allowed on a data type. If you have numerical data stored in a character column, you can perform numerical operations on the column by using the CAST function to define the column as INT for the operation.

To find the date representation of an integer, enter:

```
>SELECT CAST('11689' AS DATE) FROM TABLE;
CAST ( "11689" AS DATE )

             01 JAN 2000
             01 JAN 2000
             01 JAN 2000
```

```
              01 JAN 2000
              01 JAN 2000
              01 JAN 2000
              01 JAN 2000

7 records listed.
```

Obtaining Derived Data

A column can be an expression. This often is referred to as a calculated column, or virtual column, which is a column that does not exist physically in the database but instead is calculated from data stored in the columns of the table or file. In such cases, specify an expression using column names, arithmetic operators, and constants. Group expressions with parentheses to indicate order of precedence. Use calculated columns in the same way as physical columns.

For example, to examine the effects of an across-the-board cost increase of 10% for supplies, enter:

```
>SELECT DESCRIPTION, COST, (COST * 1.10) FROM INVENTORY.T;
DESCRIPTION.............    COST........    ( COST * 1.10 )

Jerky                                48.90             53.79
Cookies                              98.32            108.152
Mustard                              91.52            100.672
Handbills                            42.78             47.058
French Fries, Frozen                 34.95             38.445
Horse Feed                           28.37             31.207
Lemonade                             14.57             16.027
.
.
.
Elephant Chow                        11.00               12.1
Beer                                 76.92             84.612
Press any key to continue...
```

In the previous example, COST * 1.10 is a calculated, or virtual, column created by multiplying COST by 1.10.

To calculate the markup on the inventory items, enter:

```
>SELECT DESCRIPTION, COST, PRICE, (COST / PRICE)
SQL+CONV 'MD2' COL.HDG 'Markup' FROM INVENTORY.T;
DESCRIPTION.............    COST........    PRICE.......
Markup

Jerky                                48.90           64.55    0.76
Cookies                              98.32          143.55    0.68
Mustard                              91.52          135.45    0.68
```

```
Handbills                          42.78          57.33    0.75
French Fries, Frozen               34.95          45.78    0.76
.
.
.
Elephant Chow                      11.00          16.61    0.66
Beer                               76.92         116.92    0.66
Press any key to continue...
```

Usually, the column heading is the expression itself (COST/PRICE), but here a COL.HDG field qualifier changes it to "Markup." You probably would not want the markup calculated out to nine decimal places, and there are ways to truncate these values, which are covered later. CONV 'MD2' is a conversion code that simply rounds off the results to two decimal places.

You also can use the EVAL expression to obtain derived data. EVAL Expressions specify an I-descriptor and can be thought of as an enhancement to SQL's calculated column feature. In addition to the column names, constants, and arithmetic operators allowed in simple column expressions, EVAL expressions can contain UniVerse BASIC language elements such as conditional statements and even UniVerse BASIC subroutines.

Selecting Rows

Now that you know how to retrieve data from certain *columns*, you can limit retrieval to certain selected *rows* (records):

- Primary key selection
- Sampling
- Selection criteria
- Negation
- Compound search criteria
- Select lists
- INQUIRING and in-line prompting

Selecting Rows by Primary Key

A primary key, whether made up of a single column or multiple columns, uniquely identifies each row in a table or file. If a table has no primary key, the values in the @ID column uniquely identify each row. Therefore, one of the simplest ways to select rows is to use primary keys (or values in the @ID column) to specify the rows you want to examine. Always enclose the primary key value in single quotation marks. For example, to look at all the data for animal 48 in the LIVESTOCK.T table, enter:

```
>SELECT * FROM LIVESTOCK.T '48';
ANIMAL_ID...    48
NAME........ Marone
DESCRIPTION. Ocelot
USE......... Z
DOB......... 11/01/91
ORIGIN...... Texas
COST........      8838.00
EST_LIFE....  19
VAC_TYPE VAC_DATE.. VAC_NEXT.. VAC_CERT
R        01/23/92   01/22/95   812616
P        05/08/92   05/08/95   659260
L        08/31/92   08/31/95   643116

1 records listed.
```

To see the data for more than one primary key value, list them (although as described in "Set Membership" on page 24, it is advisable to use a set membership test). Do *not* use commas to separate the items in a series of primary key values.

```
>SELECT * FROM LIVESTOCK.T '63' '29' '55';
ANIMAL_ID...    63
NAME........ Foula
DESCRIPTION. Shetland
USE......... P
DOB......... 10/05/79
ORIGIN...... England
COST........      6608.00
EST_LIFE....  16
VAC_TYPE VAC_DATE.. VAC_NEXT.. VAC_CERT
R        01/04/92   01/03/95   121250
P        01/12/93   01/12/96   332255
L        04/04/93   04/03/96   1647

ANIMAL_ID...    29
NAME........ Okene
DESCRIPTION. Lion
USE......... P
```

```
DOB......... 11/12/90
ORIGIN...... Kenya
COST........      8574.00
EST_LIFE....  14
VAC_TYPE VAC_DATE.. VAC_NEXT.. VAC_CERT
Press any key to continue...
```

Selecting Rows by Sampling (SAMPLE and SAMPLED)

Another simple, but less common way of selecting rows is by sampling, which limits the number of rows selected for output. Sampling is often used to test a complex query against a large table without consuming the system resources that would be required to run the query against the entire table. Sampling is not a standard SQL feature, but is one of the many SQL extensions in UniVerse SQL that are related to UniVerse features.

Two processing qualifiers control sampling: SAMPLE *n* selects the first *n* rows, and SAMPLED *n* selects every *n*th row. Even though the SAMPLE or SAMPLED clause is at the end of the statement, sampling is done first, before any sorting or other function is performed.

To examine a small sampling of the vendors with which you do business, and ask for the *first* 10 vendors in the table, enter:

```
>SELECT COMPANY FROM VENDORS.T SAMPLE 10;
COMPANY..................

Pure Academy
Central Automation
Illinois Operations
Utopia Professionals
Continental Mart
Red Controls
Republic Manufacturers
Northern Outlets
Hollywood Retail
Ohio Treating

Sample of 10 records listed.
```

However, you may want to browse the entire table for your samples, looking at every *n*th row. Because the VENDORS.T table contains 232 rows, selecting every 25th row produces 9 records (232 rows divided by 25). To ask for a sorted listing of every 25th row, enter:

```
>SELECT COMPANY FROM VENDORS.T ORDER BY COMPANY SAMPLED 25;
COMPANY..................

Affordable Merchandise
Bayou Manufacturers
Country Traders
Eve Mart
Immediate Enterprises
Lucky Environmental
Main Street Traders
New York Advisers
True Manor

Sample of 9 records listed.
```

Selecting Rows Based on Selection Criteria (WHERE)

The third way to select rows for retrieval is by using the WHERE Clause to specify selection criteria. Whenever you use a WHERE clause, SQL evaluates each row of the table, testing it against the criteria you have specified. If a row passes the test, it is included in the results. If not, it is excluded from the results.

FMT ‘30L’ is a format option that leaves enough space for DESCRIPTION and NAME so that they will not wordwrap onto a second line.

To see a listing of only those animals suitable for the petting zoo area, enter:

```
>SELECT DESCRIPTION FROM LIVESTOCK.T WHERE USE = 'Z';
DESCRIPTION

Puma
Civet
Kinkajou
Ocelot
Kodkod
Sable
Jaguar
.
.
.
Linsang
Press any key to continue...
```

To see only those engagements scheduled for the fourth quarter of 1995, enter:

```
>SELECT LOCATION_CODE, "DATE" FROM ENGAGEMENTS.T
SQL+WHERE "DATE" BETWEEN '10/01/95' AND '12/31/95';
LOCATION_CODE    DATE......

CIAH001          10/03/95
CIAH001          10/04/95
WSEA001          12/07/95
WSEA001          12/08/95
CCLE001          12/15/95
CCLE001          12/16/95
CDFW001          10/15/95
CDFW001          10/16/95
ENYC001          11/23/95
EHAR001          10/23/95
ENYC001          11/24/95
WLAX001          12/26/95
EHAR001          10/24/95
WLAX001          12/27/95

14 records listed.
```

To see only those locations that have 50, 100, or 125 acres, enter:

```
>SELECT DESCRIPTION FMT '30L', NAME FMT '30L', ACRES FROM
LOCATIONS.T
SQL+WHERE ACRES IN (50, 100, 125);
DESCRIPTION.................      NAME.....................
ACRES

Houston State Fair Ground        Houston Properties, Inc.
50
Minneapolis State Fair Ground    Minneapolis Properties, Inc.
125
Washington State Fair Ground     Washington Properties, Inc.
100
Springfield State Fair Ground    Springfield Properties, Inc.
125
Los Angeles State Fair Ground    Los Angeles Properties, Inc.
50
Boston State Fair Ground         Boston Properties, Inc.
125
Philadelphia State Fair          Philadelphia Properties, Inc.
50
Seattle State Fair Ground        Seattle Properties, Inc.
125
Jacksonville State Fair Ground   Jacksonville Properties, Inc.
100
Indianapolis State Fair Ground   Indianapolis Properties, Inc.
125

10 records listed.
```

To list only those staff members whose last name is pronounced similarly to *Kowslowsky*, enter:

```
>SELECT NAME FROM PERSONNEL.T
SQL+WHERE NAME SAID 'KOWSLOWSKY';
NAME.....................

Kozlowski, Nicholas
Kozlowski, Bill
Kozlowski, Joe

3 records listed.
```

To list only those rides whose description begins with *Carousel*, enter:

```
>SELECT DESCRIPTION FROM RIDES.T
SQL+WHERE DESCRIPTION LIKE 'Carousel%';
DESCRIPTION.........

Carousel - Horses
Carousel - Jet
Planes
Carousel - Rockets

3 records listed.
```

To see only those engagements for which the advance payment is null, enter:

```
>SELECT LOCATION_CODE, "DATE", ADVANCE FROM ENGAGEMENTS.T
SQL+WHERE ADVANCE IS NULL;
LOCATION_CODE    DATE......    ADVANCE.....

WREN001          01/10/94
CMSP001          08/17/92
WDEN001          04/30/93
WREN001          01/11/94
CMSP001          08/18/92
.
.
.
134 records listed.
```

Null in SQL refers to an *unknown* value, not a 0 or blank, or an empty string. Null values are covered in greater detail in "Testing for Null Values" on page 26.

As the previous examples illustrate, selection can be based on:

- Comparisons
- Ranges
- Set membership

- Phonetic matching
- Pattern matching
- Null values

In addition, you can add the keyword NOT to negate a search condition, and create compound search conditions by using the logical operators AND and OR. For more information, see "Negation" on page 27.

Comparisons

Just as with RetrieVe operations, you can select rows by comparing the contents of a column to a value. Comparisons can be simple, testing only one column, or complex, and employing comparison operators (=, <>, #, <, <=, >, >=) and logical operators (AND, OR, NOT). The simplest form is a column and a constant, as shown by the following query, which retrieves all the data concerning employee 93:

```
>SELECT * FROM PERSONNEL.T
SQL+WHERE BADGE_NO = 93;

BADGE_NO.    93
DOB..... 09/11/65
BENEFITS. O,G,C
NAME..... Lewis, Wayne
ADR1..... 6030 Argonne Street
ADR2..... Security  CO 80911
ADR3.....
PHONE.... 719/984-2824
DEP_NAME.. DEP_DOB... DEP_RELATION

EQUIP_CODE EQUIP_PAY.
        17      11.84
        60      14.44
ACT_NO ACT_PAY...
     2      13.53
     1      11.99
     4      15.85
RIDE_ID RIDE_PAY..
      7      12.37
      1       8.60

Press any key to continue...
```

This type of query is used commonly in forms-based data retrieval, in which the user types a customer number into a screen form and that number is used to build and execute a query.

As an example of a more complex comparison, you could ask for all engagements with an advance over $10,000 that are scheduled before the end of 1995:

```
>SELECT ADVANCE, LOCATION_CODE, "DATE" FROM ENGAGEMENTS.T
SQL+WHERE ADVANCE > 10000 AND "DATE" <= '12/31/95';
ADVANCE.....    LOCATION_CODE    DATE......

    10572.00    WSEA001          12/07/95
    11935.00    WPHX001          08/09/95
    10863.00    CCLE001          12/15/95
    11971.00    CDFW001          10/15/95
    .
    .
    .
    10160.00    EPHI001          03/17/95
    10280.00    EJAC001          03/18/95

16 records listed.
```

Remember that in SQL, null represents an unknown value, not a 0 (zero) or an empty value. Consequently, a row with a null value can seem to "disappear," as in a case where you ask for a list of equipment `WHERE COST > $10000` and then you ask for an equipment list `WHERE COST <= $10000`. One might assume that the combined output would equal the number of rows in the table, but if COST is null for one or more rows, those rows do not appear in either output.

Ranges

Another type of row selection is the range test, which you use to select rows in which the contents of a column lies between two values, inclusive. Range tests use the keyword BETWEEN, which provides a shorthand for *column* >= *value* AND *column* <= *value*.

A range test is handy for selecting rows belonging to a certain calendar period, items whose dollar amounts fall within a certain monetary range, and so on. The following two examples demonstrate this.

To find equipment that costs between $50,000 and $75,000 and get a listing of that equipment in *descending* order by cost, enter:

```
>SELECT COST, DESCRIPTION FROM EQUIPMENT.T
SQL+WHERE COST BETWEEN 50000 AND 75000
SQL+ORDER BY COST DESC;
COST........    DESCRIPTION..............

    75000.00    Calliope
    70591.65    Cooling System
    70081.99    Electrical Generator
```

```
    69990.43    Mail Machine
    68278.35    Truck 665 B C C
    67521.49    Coffee/cookies Stand
    67448.24    Desk Credenza Sets
    66700.54    Truck 897 M X X
    61558.17    Truck 102 T I U
    58555.15    Hot Dog Stand
    57581.61    Subsidiary Tent Frame
    57355.77    Harness Equipment
    55594.85    V C R
    52005.88    Soft Drinks Stand
    51004.87    Truck 243 Y G N
    50370.08    Copier

16 records listed.
```

To list the products that have a markup of between 60% and 80%, enter:

```
>SELECT DESCRIPTION, COST, PRICE, (COST / PRICE) CONV 'MD2'
SQL+FROM INVENTORY.T
SQL+WHERE (COST / PRICE) BETWEEN 0.6 AND 0.8;
DESCRIPTION.............    COST........    PRICE.......  ( COST
/ PRICE )

Mustard                           91.52         135.45
0.68
French Fries, Frozen               34.95          45.78
0.76
Jerky                             48.90          64.55
0.76
Cookies                           98.32         143.55
0.68
Handbills                         42.78          57.33
0.75
Horse Feed                        28.37          38.58
0.74
Lemonade                          14.57          20.25
0.72
.
.
.
Nachos                            28.61          42.06
0.68
Imported Ale                      13.51          20.13
0.67
Press any key to continue...
```

Set Membership

When comparing a column to more than one value, you will type less by using the IN Keyword rather than writing out the comparison as a series of *column = value* clauses. In effect, the target values against which you are testing the column constitute a mathematical set, and is sometimes called a set membership test. Here are two examples.

To list acts 1, 3, and 5 and their duration, enter:

```
>SELECT DESCRIPTION, DURATION FROM ACTS.T
SQL+WHERE ACT_NO IN (1, 3, 5);

DESCRIPTION    DURATION

Salute to            12
the Circus
Animals on            6
Parade
Rock Around           5
the Big Top

3 records listed.
```

To list all engagements where rides 5, 9, or 11 have been booked, enter:

```
>SELECT LOCATION_CODE, "DATE"
SQL+FROM ENGAGEMENTS.T
SQL+WHERE RIDE_ID IN (5, 9, 11);
LOCATION_CODE    DATE......

CKAN001          06/05/96
ENYC001          06/05/96
CKAN001          06/06/96
CDET001          12/16/96
WREN001          01/10/94
.
.
.
CIAH001          10/03/95
Press any key to continue...
```

Phonetic Matching

Phonetic matching uses a phonetic, or sounds like, algorithm (invoked by the relational operator Phonetic Matching: SAID) to match a text string to a sound. To list animals with names that sound like “lyon,” “fauks,” or “tyger,” enter:

```
>SELECT DISTINCT DESCRIPTION FROM LIVESTOCK.T
SQL+WHERE DESCRIPTION SAID 'LYON'
SQL+OR DESCRIPTION SAID 'FAUKS'
SQL+OR DESCRIPTION SAID 'TYGER';
DESCRIPTION

Lion
Tiger
Fox

3 records listed.
```

Pattern Matching

As with RetrieVe, you can use pattern matching to select rows of data. Using Pattern Matching: LIKE, select rows whose columns match a certain pattern.

The percent sign (%) is a wildcard that matches zero or more characters. An underscore (_) is a wildcard that matches exactly one character.

Placing the % wildcard character before *and* after the text string effectively says “search for this text string no matter where it appears in the value.” To select vendors whose company names contain the word *Manufacturers*, enter:

```
>SELECT DISTINCT COMPANY FROM VENDORS.T
SQL+WHERE COMPANY LIKE '%Manufacturers%';
COMPANY..................

Republic Manufacturers
Southern Manufacturers
City Manufacturers
New Orleans Manufacturers
Bayou Manufacturers

5 records listed.
```

To retrieve all vendor companies that begin with *San* and have *D* as the fifth letter, enter:

```
>SELECT COMPANY FROM VENDORS.T
SQL+WHERE COMPANY LIKE 'San_D%';
COMPANY.................

San Diego Promotions

1 records listed.
```

To use either wildcard character (% or _) as a pattern match character, remove its wildcard status with an escape character. This is a two-step process:

1. In the pattern to be matched, precede the wildcard-character-turned-search-character with an escape character.
2. Define the escape character using the ESCAPE clause, with the escape character enclosed in single quotation marks. The rarely used backslash (\) is a recommended escape character.

For example, assume that the INVENTORY.T table contains a MARKUP column, and that it stores values with an actual "%". To search for markups between 20% and 29%, enter:

```
>SELECT INVENTORY.T
SQL+WHERE MARKUP LIKE '2_\%' ESCAPE '\';
```

Here, the % is preceded by a \, which identifies the % as an actual character rather than a wildcard. Then \ then is defined as the escape character.

Testing for Null Values

In SQL, the null value represents data whose value is unknown. Null is not an empty string (a character string of 0 length known to have no value), nor is it a string of zeros or blanks.

For any given row, the result of a search can be TRUE, FALSE, or (if one of the columns contains a null value), UNKNOWN. It is a good idea to check explicitly for null values before proceeding to apply other search conditions.

Use Testing for the Null Value: IS NULL (but *not* = NULL) to select rows based on the presence of a null value in a column. To list those engagements in the last quarter of 1994 that have an ADVANCE of NULL, enter:

```
>SELECT LOCATION_CODE, "DATE" FROM ENGAGEMENTS.T
SQL+WHERE "DATE" BETWEEN '10/01/94' AND '12/31/94'
SQL+AND ADVANCE IS NULL;
LOCATION_CODE    DATE......

CIND001          10/04/94
CIND001          10/05/94
EJAC001          12/08/94
EJAC001          12/09/94
EPHI001          11/13/94
EPHI001          11/14/94

6 records listed.
```

Using CAST with WHERE

You can use the CAST function to search for patterns in numerical data. For example, to find all employees whose badge number ends with 44, enter:

```
>SELECT BADGE_NO, NAME FROM PERSONNEL.T
SQL+WHERE CAST(BADGE_NO AS VARCHAR) LIKE '%44';
BADGE_NO    NAME...................

     144    Hanson, Daniel
      44     Vaughan, Mary

2 records listed.
```

Negation

To negate a search criterion, in most cases all you have to do is precede it with the keyword NOT, which effectively reverses the original meaning. Thus, taking the same examples that introduced this section on selecting rows, you could change the effect of each query by preceding each search criterion with the keyword NOT.

To see those engagements that do *not* fall within the fourth quarter of 1995, enter:

```
>SELECT LOCATION_CODE, "DATE" FROM ENGAGEMENTS.T
SQL+WHERE "DATE" NOT BETWEEN '10/01/95' AND '12/31/95';
LOCATION_CODE    DATE......

CKAN001          06/05/96
ENYC001          06/05/96
CDET001          12/15/96
```

```
CKAN001          06/06/96
.
.
.
WSDO001          04/08/95
Press any key to continue...
```

To see those locations whose land area is *not* 50, 100, or 125 acres, enter:

```
>SELECT DESCRIPTION FMT '30L', NAME FMT '30L', ACRES FROM
LOCATIONS.T
SQL+WHERE ACRES NOT IN (50, 100, 125);
DESCRIPTION.................       NAME...................
ACRES

Milwaukee State Fair Ground      Milwaukee Properties Inc.
25
Detroit State Fair Ground         Detroit Properties, Inc.
150
Dallas State Fair Ground          Dallas Properties, Inc.
200
.
.
.
Hartford State Fair              Hartford Properties, Inc.
25
Press any key to continue...
```

To see all personnel whose last names are *not* pronounced similarly to *Kowslowsky*, enter:

```
>SELECT NAME FROM PERSONNEL.T
SQL+WHERE NAME NOT SAID 'KOWSLOWSKY';
NAME......................

Torres, Stephen
Hanson, Daniel
Niederberger, Brian
.
.
.
Young, Carol
Press any key to continue...
```

To list the rides whose names do *not* begin with *Carousel*, enter:

```
>SELECT DESCRIPTION FROM RIDES.T
SQL+WHERE DESCRIPTION NOT LIKE 'Carousel%';
DESCRIPTION.........

Bumper Cars
Moonwalk
Mechanical Bull
```

```
.
.
.
Tilt

12 records listed.
```

To see the fourth quarter 1994 engagements where the advance payment is *not* null, enter:

```
>SELECT LOCATION_CODE, "DATE", ADVANCE FROM ENGAGEMENTS.T
SQL+WHERE "DATE" BETWEEN '10/01/94' AND '12/31/94'
SQL+AND ADVANCE IS NOT NULL;
LOCATION_CODE    DATE......    ADVANCE.....

CIAH001          12/28/94           7392.00
CIAH001          12/29/94           8286.00
WREN001          12/31/94           8757.00

3 records listed.
```

For comparisons, use the appropriate comparison operator rather than NOT to form the negation:

The negation of...	Is...
=	<> or # (inequality)
<> or #	= (equality)
<	>=
>	<=
>=	<
<=	>

Comparison Operators

Consequently, the negation of the first example

```
>SELECT DESCRIPTION, USE FROM LIVESTOCK.T
SQL+WHERE USE = 'Z';
```

is as follows:

```
>SELECT DESCRIPTION, USE FROM LIVESTOCK.T
SQL+WHERE USE <> 'Z';
DESCRIPTION    USE

Shetland       R
Lion           P
Dog            P
.
.
.
Shetland       P
Press any key to continue...
```

Compound Search Criteria

You can combine simple search criteria to create more complex search conditions using the logical operators AND, OR, and NOT.

Thinking of *each* search criterion as returning a value of TRUE, FALSE, or UNKNOWN, you can visualize how compound search criteria will operate. Looking at the "truth tables" for AND, OR, and NOT may be helpful.

AND Truth Table

Using the AND operator requires that both or all parts of a statement are true for the entire statement to be true. If just one part of a statement joined using AND is false, the entire statement is considered false.

	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

AND Truth Table

OR Truth Table

Using the OR operator requires that only one portion of a statement joined using the operator OR is true for the entire statement to be considered true. If one portion of a statement is true, and one portion is false, the entire statement is considered true.

	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

OR Truth Table

NOT Truth Table

Using the operator NOT negates all portions of a statement.

	TRUE	FALSE	UNKNOWN
	FALSE	TRUE	UNKNOWN

NOT Truth Table

The preceding tables show that if criterion_1 is true and criterion_2 is false, then criterion_1 AND criterion_2 is false, but criterion_1 OR criterion_2 is true.

When one of the criteria is unknown, the logic behind the result is not as obvious. For example, in the OR table, criterion_1 being unknown and criterion_2 being false produces a result of unknown. This is because the unknown quality could be true or false, making the possible result true or false or, in other words, unknown. This is known as three-valued logic.

Use AND to connect two search conditions when you want to select a row only if *both* conditions are true.

To list all locations that offer at least 150 or more acres *and* 6,000 or more seats:

```
>SELECT DESCRIPTION, ACRES, SEATS FROM LOCATIONS.T
SQL+WHERE ACRES >= 150 AND SEATS >= 6000;
DESCRIPTION.............    ACRES    SEATS

Topsfield Fair                500    10000
Golden Gate Exposition        175     8000
```

```
Center
Las Vegas Convention           175     8000
Center
Chicago State Fair Ground      150     6000

4 records listed.
```

Use OR to connect two search conditions when you want to select a row if *either* search condition is (or both are) true. To list all locations that offer either 150 or more acres *or* 6,000 or more seats, enter:

```
>SELECT DESCRIPTION FMT '30L', ACRES, SEATS FROM LOCATIONS.T
SQL+WHERE ACRES >= 150 OR SEATS >= 6000;
DESCRIPTION...................    ACRES    SEATS

Detroit State Fair Ground            150     1000
Milwaukee State Fair Ground           25     6000
Springfield State Fair Ground        125     6000
.
.
.
Portland State Fair Ground           175     5000
Reno State Fair Ground               175     4000

21 records listed.
```

You can string together more than two search conditions, and the search conditions connected by the ANDs and ORs can themselves be compound search conditions. To list the Shetlands vaccinated after January 1, 1993, that are suitable for rides, enter:

```
>SELECT ANIMAL_ID, NAME FROM LIVESTOCK.T
SQL+WHERE DESCRIPTION LIKE 'Shetland%'
SQL+AND USE = 'R'
SQL+AND VAC_DATE > '1/1/93';
ANIMAL_ID    NAME......

       33    Eggau
       46    Mora
       58    Tumu
       43    Gombi
       31    Akure
       51    Banyo
       71    Bousso
       15    Kontagora
        5    Sokoto

9 records listed.
```

Use parentheses to clarify the order in which the search conditions should be evaluated. Conditions enclosed in parentheses are evaluated together and produce a single result (true, false, or unknown). To list the employees living in Pennsylvania, Massachusetts, or New York *and* born before 1950 or after 1969, enter:

```
>SELECT NAME, ADR2, DOB FROM PERSONNEL.T
SQL+WHERE (ADR2 LIKE '%PA%'
SQL+OR ADR2 LIKE '%MA%'
SQL+OR ADR2 LIKE '%NY%')
SQL+AND (DOB < '1/1/50' OR DOB > '12/31/69');
NAME.....................  ADR2.....................  DOB.......

Carter, Joseph             Boston MA 02116
Galloway, Jane             Summer Isle NY 10322       06/09/47
Kozlowski, Bill            Sterling MA 01564          09/06/74
Bacon, Roger               Leicester MA 01524         03/12/39
Carsley, Rusty             Harrisburg, PA 10964       04/10/70

5 records listed.
```

In the previous example, placing the state and date-of-birth conditions within parentheses causes them to be evaluated separately, and each set produces one result. Then, because these two results are connected by AND, both results must be true to select the row.

Omitting parentheses causes conditions to be evaluated in the following order:

- NOTs
- ANDs
- ORs

If you omit the parentheses, the order of precedence results in the selection of everyone from Pennsylvania or Massachusetts, anyone from New York who was born before 1950, and anyone from any state who was born after 1969:

```
>SELECT NAME, ADR2, DOB FROM PERSONNEL.T
SQL+WHERE ADR2 LIKE '%PA%'
SQL+OR ADR2 LIKE '%MA%'
SQL+OR ADR2 LIKE '%NY%'
SQL+AND DOB < '1/1/50' OR DOB > '12/31/69';
NAME.....................    ADR2.....................
DOB.......

Nelson, Lisa                 Energy  WY 82422             02/07/71
Niederberger, Brian          Equity  OH 43749             05/27/72
Torres, Stephen              Cash  VA 22942               12/04/74
Carter, Joseph               Boston MA 02116
Clark, Lisa                  What Cheer  IA 50268         01/03/71
Grant, Nancy                 Beautiful  PA 15009          07/30/59
```

```
.
.
.
Martinez, Suzanne            Merit  MS 38759              02/03/71
Schultz, Mary Lou            Happy  TX 79042              12/15/73

27 records listed.
```

Selecting Rows Through Select Lists

The SELECT and SSELECT commands let you select a subset of rows from a table, and put their record IDs into a select list. You then can process the select list with another RetrieVe command.

A UniVerse SQL SELECT statement also can be used to process an active select list by including the SLIST keyword in the statement.

The following example uses a SELECT sentence to create a select list containing the record IDs of all INVENTORY.T rows where QOH is greater than 100. A UniVerse SQL SELECT statement with the SLIST keyword further selects only those items with a value of over $60:

```
>SELECT INVENTORY.T WITH QOH > 100
32 record(s) selected to SELECT list #0.
>>SELECT QOH, COST FROM INVENTORY.T SLIST 0 WHERE COST > 60;
QOH..    COST........

  174           98.32
  125           91.52
  181           90.48
  135           87.22
  143           79.78
  127           76.92
  131           96.36
  185          102.83
  .
  .
  .
  193           69.23

15 records listed.
```

Selecting Rows Through INQUIRING and Inline Prompting

To instruct the system to interactively prompt for the primary key values (or @ID values if the table has no primary key) of the rows you want to see, use the INQUIRING Keyword or an inline (<<…>>) prompt.

Using the INQUIRING Keyword

As with RetrieVe commands, use the INQUIRING keyword to display a prompt to specify interactively what rows are to be selected from a table. When you specify INQUIRING in your SELECT statement, the system asks you for the record ID of the row you want to see and displays the requested columns of that row.

For example, to create a SELECT statement that you can use to do an ad hoc check on inventory levels, enter:

```
>SELECT DESCRIPTION, QOH FROM INVENTORY.T INQUIRING;
DESCRIPTION.............    QOH..

Primary key for table INVENTORY.T =  10
DESCRIPTION.............    QOH..

Franks                        151

Primary key for table INVENTORY.T =
```

The system prompts for a record ID (in this case, an inventory item number). After you have typed the item number, the system displays the item's description and quantity on hand, and then prompts for another record ID. To terminate the inquiry cycle, respond to `Primary key for table tablename =` by pressing **Enter**.

Note that INQUIRING is specified immediately after the table name, just as you specify SLIST or explicit record IDs in RetrieVe. It is not placed at the end of the statement and is not a report qualifier.

Also note that if you enter something like the following, the output is different depending on whether or not you enter the record ID of an item that satisfies the search criteria (QOH > 150):

```
>SELECT DESCRIPTION, QOH FROM INVENTORY.T INQUIRING
SQL+WHERE QOH > 150;
DESCRIPTION.............    QOH..

Primary key for table INVENTORY.T =
```

If you enter the record ID of an item that fits the selection criteria, you get this output:

```
Primary key for table INVENTORY.T =  28

DESCRIPTION.............    QOH..

Cookies                       174

Primary key for table INVENTORY.T =
```

However, if you ask for an item that does *not* fulfill the selection criteria, you get no output except column headers:

```
Primary key for table =  27

DESCRIPTION.............    QOH..

Primary key for table INVENTORY.T =
```

Using an Inline Prompt

Alternatively, use an inline prompt to prompt the user for values required to complete the SQL statement. For example:

```
>SELECT DESCRIPTION, QOH FROM INVENTORY.T
SQL+WHERE QOH > <<Enter Quantity On Hand>>;
Enter Quantity On Hand = 150
DESCRIPTION.............    QOH..

Cookies                      174
Handbills                    154
Horse Feed                   155
Lemonade                     153
Fried Clams                  174
.
.
.
Cheese Slices                169
Ice Bags                     193

19 records listed.
```

Summarizing Table Contents (Set Functions)

Rather than retrieve a table as individual rows, you may want one value that summarizes the contents of an entire column. Like RetrieVe and UniVerse BASIC, SQL provides set functions that produce sums, averages, minimums and maximums, and counts:

Set Function	Purpose
AVG	Averages the values in a column or an expression.
COUNT(*)	Counts the number of selected rows.
COUNT	Counts the number of values in a column.

Set Functions

Set Function	Purpose
MAX	Finds the largest value in a column or an expression.
MIN	Finds the smallest value in a column or an expression.
SUM	Adds the values in a column or an expression.

Set Functions (Continued)

Averaging and Summing

To find the average cost of the equipment, use the AVG function:

```
>SELECT AVG(COST) CONV 'MD2$' FROM EQUIPMENT.T;
AVG ( COST )

   $49104.94

1 records listed.
```

To compute the total value of the inventory, for each row multiply the price of each by the quantity on hand. Then use the SUM function to add the results for the entire table:

```
>SELECT SUM(QOH * PRICE) CONV 'MD2$' FROM INVENTORY.T;
SUM ( QOH * PRICE )

          398309.34

1 records listed.
```

CONV ‘MD2$’ displays the result of the computation in dollars and cents. CONV is a field qualifier that enables you to convert a column’s value.

Finding the Lowest and Highest Values in a Column

To find the lowest and highest values of a column, use the MIN and MAX functions:

```
>SELECT MIN(COST), MAX(COST) FROM INVENTORY.T;
MIN ( COST )    MAX ( COST )

       10.76          103.80

1 records listed.
```

Counting

The most common form of COUNT is COUNT(*), sometimes called the "row count" function, which counts the number of selected rows. For example, to count all the engagements, use COUNT(*):

```
>SELECT COUNT(*) FROM ENGAGEMENTS.T;
COUNT ( * )

        248

1 records listed.
```

COUNT counts the number of values in a column. Because COUNT ignores the actual values in the column, in most cases it does not matter which singlevalued column you use, because it will return the same answer as COUNT(*).

To do a *row*-count on LIVESTOCK.T for all petting zoo animals, enter:

```
>SELECT COUNT(*) FROM LIVESTOCK.T
SQL+WHERE USE = 'Z';
COUNT ( * )

         50

1 records listed.
```

To do a *value*-count on the DESCRIPTION column in LIVESTOCK.T for all petting zoo animals, enter:

```
>SELECT COUNT(DESCRIPTION) FROM LIVESTOCK.T
SQL+WHERE USE = 'Z';
COUNT ( DESCRIPTION )

                  50

1 records listed.
```

When COUNT is counting all the selected rows of a table or counting the number of values in the singlevalued DESCRIPTION column, the results are the same. There are two exceptions.

First, if you specify a multivalued column as the argument for COUNT, it counts the total number of values, which will be different from doing a COUNT(*) or counting a singlevalued column on the same table. The following query uses VAC_TYPE, which is a multivalued column. You can see that the count differs from the counts obtained previously.

```
>SELECT COUNT(VAC_TYPE) FROM LIVESTOCK.T
SQL+WHERE USE = 'Z';
COUNT ( VAC_TYPE )

               150

1 records listed.
```

Second, you can add the DISTINCT keyword to the COUNT argument, and use it to count the number of different values in a column. To count the different kinds of animal suitable for zoo duty and use COUNT(DISTINCT DESCRIPTION), you will again get a different result:

```
>SELECT COUNT(DISTINCT DESCRIPTION) FROM LIVESTOCK.T
SQL+WHERE USE = 'Z';
COUNT ( DISTINCT DESCRIPTION )

                            33

1 records listed.
```

Manipulating the Output

Manipulate the output of a UniVerse SQL query by doing any of the following:

- Sorting the output rows according to the content of one or more columns
- Formatting the individual columns in the report
- Formatting the overall report itself

Sorting Output

Rows in SQL tables (and UniVerse data files) are not stored in any fixed order, and the physical position of a row has no significance. Consequently, when you retrieve data from a table or file, there is no guarantee that it will be presented in any particular sequence. For example, asking for a list of engagement dates produces an unsorted list, in which even instances of the same date might be scattered throughout:

```
>SELECT "DATE" FROM ENGAGEMENTS.T;
DATE......

06/05/96
06/05/96
12/15/96
06/06/96
.
.
.
04/08/95
Press any key to continue...
```

The ORDER BY Clause sorts the output rows meaningfully and is similar to RetrieVe's BY keywords. Rows are sorted in ascending order by default, but you can add the ASC (ascending) keyword. Specify DESC (descending) to sort rows in descending order. The following example specifies neither sequence and therefore presents the dates in ascending order:

```
>SELECT "DATE" FROM ENGAGEMENTS.T ORDER BY "DATE";
DATE......

05/31/92
06/01/92
06/01/92
06/02/92
```

```
08/17/92
.
.
.
12/03/92
Press any key to continue...
```

You can use expressions and specify more than one sort in a query (and even ask for each sorted column to be in a different order). Specifying more than one sort is sometimes called a *nested sort* or a *sort within a sort*.

An example of all three options is to list the inventory in descending order of markup and ascending order of item code:

```
>SELECT (PRICE / COST * 100) COL.HDG 'Markup'
SQL+FMT 'R2', ITEM_CODE, DESCRIPTION FROM INVENTORY.T
SQL+ORDER BY 1 DESC, 2;
Markup    ITEM_CODE    DESCRIPTION..............

                37    Dog Chow
                42    Cheese Slices
159.00          31    Programs
157.01          29    Paper Plates
156.01          27    Ice Tea
154.00          22    Egg Rolls
153.00          18    Salsa
152.00           1    Beer
151.00          33    Elephant Chow
150.01          35    Domestic Cat Chow
.
.
.
110.00          21    Sea Snails
110.00          44    Onion Rings
  0.00          17    Nachos

45 records listed.
```

Column numbers rather than column names are used in the ORDER BY clause. A column number represents the position of the column specification in the SELECT clause, and is a shorthand way to refer to columns already named in the query. (However, if you used SELECT *, a column number would represent the position of the column specification as it appeared in the CREATE TABLE statement that created the table, but this is rarely done.)

The second sort (and any subsequent sorts) is performed only within equal values of the previous sort field.

Column numbers are useful particularly when one or more of the columns is an expression, as is the case here. If you were not able to use column numbers (or an AS field qualifier to create a column alias, as explained later), you would have had to repeat the entire (PRICE / COST * 100) expression.

The field qualifier FMT 'R2' reduced the results of the markup computation to two decimal places, and the field qualifier COL.HDG changed its column heading to read 'Markup' rather than (PRICE / COST * 100).

Formatting Columns

The way a column appears in the output of a query depends, by default, on the way the column is described in the dictionary of the table or file. SQL provides you with many ways to modify this output, including:

- Field modifiers
- Text
- Field qualifiers

Most of these should be familiar to users of RetrieVe.

Using Field Modifiers

Field Modifiers act on the selected contents of a column, EVAL expression, or temporary name (alias) and include the following:

Field Modifier	Synonym
AVERAGE	AVG
BREAK ON "*text* '*options*'…"	BREAK.ON "*text* '*options*'…"
BREAK SUPPRESS "*text* '*options*'…"	BREAK.SUP "*text* '*options*'…"
CALCULATE	CALC
PERCENT *n*	PERC *n,* %*n,* PERCENTAGE *n*
TOTAL	

Field Modifiers

The AVERAGE (or AVG) and TOTAL field modifiers are not the same as the AVG and SUM set functions. AVG and SUM produce a *singlerow* table containing the sum or average of the requested column:

```
>SELECT AVG(COST) FROM LIVESTOCK.T
SQL+WHERE COST > 9000;
AVG ( COST )

       10046

1 records listed.
```

However, the AVERAGE and TOTAL field modifiers produce a *multirow* (detailed table), followed by the requested average or total on a separate line at the end. COST is not enclosed in parentheses after AVERAGE, since AVERAGE is not an SQL set function:

```
>SELECT AVERAGE COST FROM LIVESTOCK.T
SQL+WHERE COST > 9000;
COST........

    10198.00
     9924.00
    10063.00
    10576.00
     9235.00
    10229.00
     9362.00
    10697.00
    10626.00
    10078.00
     9518.00
============
    10046.00

11 records listed.
```

Using Text

To add text to output results so they are more readable, specify the text (enclosed in single quotation marks) where you want it to appear in the output. For example, to show each employee's name and date of birth in the form "*name* was born on *date*", enter:

```
>SELECT NAME, 'was born on',
SQL+P_DOB FROM PERSONNEL.T SUPPRESS COLUMN HEADER;
Torres, Stephen              was born on    12/04/74
Hanson, Daniel               was born on    12/02/55
Niederberger, Brian          was born on    05/27/72
.
.
.
Sullivan, William            was born on    07/24/63
Press any key to continue...
```

The SUPPRESS COLUMN HEADER report qualifier suppresses output of the column headers, because the use of text in the previous example makes such headers redundant.

Delimited Identifiers

Text surrounded by double quotation marks is called *delimited identifiers* or *quoted identifiers*. Thus you can use reserved SQL words and identifiers (schema name, table name, view name, column name, association name, constraint name, index name, table alias, column alias or user name) as quoted identifiers. For more information about quoted identifiers, see *UniVerse SQL Administration for DBAs*.

Using the Current Date and Time

The CURRENT_DATE and CURRENT_TIME keywords make it easy to maintain a date last modified or time stamp column in a table, among other uses. CURRENT_DATE and CURRENT_TIME literally mean "today's date" and "current time," respectively. They refer to the local date and time as maintained by the local operating system.

CURRENT_DATE and CURRENT_TIME are constant during execution of a single SQL DML statement and can be used in an SQL statement anywhere a date literal or time literal can be used. For example, to find all engagements from more than 90 days ago, enter:

```
>SELECT * FROM ENGAGEMENTS.T WHERE "DATE" < CURRENT_DATE-90;
```

Using Field Qualifiers

Field Qualifiers specify an alternative format or conversion for a column. Field qualifiers override column definitions in the table's dictionary and are in effect only for the duration of the current SELECT statement. They are summarized as follows:

Field Qualifier	Synonym
AS *alias*	
ASSOCIATION "*association*"	ASSOC *"association"*
ASSOCIATED *column*	ASSOC.WITH *column*
CONVERSION *code*	CONV *"code"*
DISPLAYLIKE *column*	
DISPLAYNAME "text"	DISPLAY.NAME "*text"*, COL.HDG *"text"*
FORMAT *"format"*	FMT *"format"*
MULTIVALUED	MULTI.VALUE
SINGLEVALUED	SINGLE.VALUE

Field Qualifiers

Four of the most commonly used field qualifiers are AS, DISPLAYNAME (or COL.HDG), FORMAT (or FMT), and CONVERSION (or CONV).

Assigning a Column Alias

The AS field qualifier, although optional, specifies an alias for a column. Refer to the column later in the query by using its alias rather than its actual name. However, an alias cannot duplicate any entry in the file's dictionary.

A column alias is stored temporarily in the file's dictionary, which therefore must be writable by the user. Consequently, you cannot define a column alias when using SELECT to select from DICT *tablename*, as that requires writing to DICT.DICT, which is read-only.

One use of an alias is to abbreviate a long column name. For example:

```
>SELECT DESCRIPTION AS M1, VENDOR_CODE AS M2, COST
SQL+FROM EQUIPMENT.T
SQL+ORDER BY M2, M1;
M1................................. M2 COST........

Feeding Buckets                     1        15704.11
Truck 212 A Q S                     3        75334.22
Hamburger Stand                     4        44809.61
Harness Equipment                  10        57355.77
Taffy Stand                        15        86842.75
.
.
.
Panels                             74        48120.87
Press any key to continue...
```

If a column alias is specified, the alias is used as the column heading unless it is overridden by a COL.HDG or DISPLAY.LIKE field qualifier. You also can define a column alias within a subquery.

Here an alias is used to assign a meaningful name to an EVAL expression:

```
>SELECT DESCRIPTION, EVAL 'COST * QOH' AS VALUE FROM
SQL+INVENTORY.T WHERE VALUE > 10000;
DESCRIPTION.............    VALUE....

Cookies                          17107.68
Mustard                          11440.00
Sawdust                          16376.88
Pretzels                         11774.70
Programs                         11408.54
Dog Chow                         12623.16
Cola                             19023.55
Fried Clams                      11537.94
Sea Snails                       14040.18
Crow                             14907.49
Franks                           15087.92
Ice Cream, Various               12440.12
Ice Bags                         13361.39

13 records listed.
```

The same result is obtained using an SQL expression:

```
>SELECT DESCRIPTION, COST * QOH AS VALUE FROM INVENTORY.T WHERE
VALUE > 10000;
```

Use of the keyword AS to define an alias is optional, although it helps to distinguish aliases clearly and distinctly from other operations. If the AS keyword is omitted, the AS field qualifier must be the first field qualifier in the field qualifier list.

As of Release 9 of UniVerse, you can assign an alias to a select expression and set functions. A column alias may be defined for any of the following select expressions:

- Simple column name
- I-descriptor column name
- Literal
- USER
- NULL
- SQL expression
- Set function
- EVAL clause

If a column alias is defined for a simple column name or an I-descriptor column name, both the original column name and the alias can be referenced later in the statement. However, an alias cannot be referenced within an UNNEST clause or a joined table.

An alias can be referenced later in the same SELECT statement:

- Within an SQL expression or set function
- As the argument of a DISPLAY.LIKE field qualifier
- In WHERE, WHEN, GROUP BY, HAVING, and ORDER BY clauses

If more than one column alias is specified for the same select expression, the statement will be rejected.

Creating Column Headings (DISPLAYNAME)

By default, the column heading for a column or expression is the *columnname* or *expression* itself (or, optionally, a default DISPLAYNAME specified for the column in the file dictionary). To customize the heading, use DISPLAYNAME or COL.HDG:

```
>SELECT DESCRIPTION DISPLAYNAME 'Inventory Item',
SQL+(QOH * COST) DISPLAYNAME 'Value'
SQL+FROM INVENTORY.T;
Inventory Item...........    Value

Mustard                      11440
French Fries, Frozen         1782.
                             45
Crabcakes                    2482.
```

```
                                11
Jerky                           5085.
                                6
Cookies                         17107
                                .68
Handbills                       6588.
                                12
Horse Feed                      4397.
                                35
Lemonade                        2229.
                                21
.
.
.
Press any key to continue...
```

The output does not look quite right because:

- The 25-character inventory description results in too much space between the description and the value.
- The value (being allocated only five positions, the length of the new column header) wraps to the next line.

Use FORMAT to adjust the output.

Formatting Values (FORMAT)

Use FORMAT to change the width and justification of both items in the previous example. To reduce the inventory description to only 20 left-justified characters and extend the number of positions allocated to Value and right-justify it, enter:

```
>SELECT DESCRIPTION FORMAT '20L' DISPLAYNAME 'Inventory Item',
SQL+(QOH * COST) FORMAT '11R' DISPLAYNAME 'Value'
SQL+FROM INVENTORY.T;
Inventory Item......    Value......

Mustard                       11440
French Fries, Frozen        1782.45
Crabcakes                   2482.11
Jerky                        5085.6
Cookies                    17107.68
Handbills                   6588.12
Horse Feed                  4397.35
Lemonade                    2229.21
.
.
.
Sawdust                    16376.88
Press any key to continue...
```

Formats for the UniVerse SQL FORMAT match those in RetrieVe and UniVerse BASIC. Complete syntax of FMT formats is in *UniVerse BASIC*.

Converting Values (CONVERSION)

In the previous example, the value is not in the dollars-and-cents format you wanted. You add a conversion field qualifier (CONVERSION or CONV) to the SELECT statement to convert the result of the expression. Use a CONVERSION code such as 'MD2$,' to specify that the column is to be displayed with two decimal places, a dollar sign, and a comma every third position:

```
>SELECT DESCRIPTION FORMAT '20L' COL.HDG 'Inventory Item',
SQL+(QOH * COST) FORMAT '11R' COL.HDG 'Value'
SQL+CONVERSION 'MD2$,' FROM INVENTORY.T;
Inventory Item......    Value......

Mustard                  $11,440.00
French Fries, Frozen      $1,782.45
Crabcakes                 $2,482.11
Jerky                     $5,085.60
Cookies                  $17,107.68
Handbills                 $6,588.12
Horse Feed                $4,397.35
Lemonade                  $2,229.21
.
.
.
Sawdust                  $16,376.88
Press any key to continue...
```

As seen from this last example, you can append a DISPLAYNAME, a FORMAT, and a CONVERSION qualifier to a single column or expression to get the desired results. The UniVerse SQL conversion codes are the same as those used in RetrieVe and BASIC. A complete list of conversion codes is in *UniVerse BASIC*.

Other Field Qualifiers

The remaining field qualifiers are summarized as follows:

Field Qualifier	Description
ASSOCIATED	Same as RetrieVe's ASSOC.WITH qualifier. It associates a column with another column that is multivalued.
ASSOCIATION	Same as RetrieVe's ASSOC qualifier. It temporarily associates the column with an existing association of multivalued columns.
DISPLAYLIKE	Same as RetrieVe's DISPLAY.LIKE qualifier. It sets a column's display characteristics to the same as those of another column.
MULTIVALUED, SINGLEVALUED	Same as RetrieVe's MULTI.VALUE and SINGLE.VALUE qualifiers. Specify that the column or expression is to be treated as multivalued or singlevalued, respectively, overriding any existing definition in the file dictionary.

Field Qualifier

Formatting Reports with Report Qualifiers

Report qualifiers affect the report output as a whole, rather than individual row or column outputs. For example, use report qualifiers to control the report layout in terms of spacing between columns, spacing between rows, starting on a new page or screen, and the use of report headers/footers and column headers.

SUPPRESS COLUMN HEADER is described in "Using Text" on page 45. The report qualifiers and their synonyms are as follows:

Report Qualifier	Synonym
AUX.PORT	
COLUMN SPACES	COL.SPACES, COL.SPCS
COUNT.SUP	
DOUBLE SPACE	DBL.SPC
FOOTER *'text'*	FOOTING *'text'*

Report Qualifier Synonyms

Report Qualifier	Synonym
GRAND TOTAL	GRAND.TOTAL
HEADER *'text'*	HEADING *'text'*
LPTR [*n*]	
MARGIN *n*	
NO.INDEX	
NOPAGE	NO.PAGE
SUPPRESS COLUMN HEADING	SUPPRESS COLUMN HEADER, COL.SUP
SUPPRESS DETAIL	DET.SUP
VERTICALLY	VERT

Report Qualifier Synonyms (Continued)

Report Headings and Footings

A report heading appears at the top of every screen or page of the report; a report footing appears at the bottom. If you do not supply a heading, your output report will have no header and will start on the next line of the screen. If you *do* supply one, the report will start on a new page or at the top of the screen, and each page will have the specified heading at the top. If you supply a footing, your report will have a footer at the bottom of each screen or page.

In the following example, the SELECT statement specifies a header showing `MONTHLY LIVESTOCK REPORT` and a footer of `Press CTL-C to exit or` (which in the actual output precedes the standard message of `Press any key to continue. . .`):

```
>SELECT NAME FROM LIVESTOCK.T
SQL+HEADER 'MONTHLY LIVESTOCK REPORT'
SQL+FOOTER 'Press CTL-C to exit or';
MONTHLY LIVESTOCK REPORT
NAME......

Bussa
Warri
Ekiti
Gboko
Marone
```

```
Bassar
.
.
.
Baro
Press CTL-C to exit or
Press any key to continue...
```

Specifying HEADER DEFAULT instead produces the standard UniVerse header (query statement, time, date, and page number) at the top of each page. Besides the report heading and footing, you have a choice of column headers, such as the column name, a default COL.HDG from the file dictionary, or a COL.HDG that you supply in the query. Using SUPPRESS COLUMN HEADER eliminates the column headers entirely. Suppress the line `nn records listed.`, which appears at the end of every query output, by using COUNT.SUP.

Adjusting Spacing, Margins, Pagination, and Orientation

To refine your report layout, use DOUBLE SPACE, COLUMN SPACES, and MARGIN to adjust the spacing between columns and rows, and the left margin. Double spacing especially is important when displaying multivalues within rows. Adjusting column spacing allows you to either tighten up or spread out a report. Here, COLUMN SPACES 2 tightens up the column spacing:

```
>SELECT LOCATION_CODE, "DATE", ADVANCE FMT '8R'
SQL+FROM ENGAGEMENTS.T
SQL+ORDER BY LOCATION_CODE COLUMN SPACES 2;
LOCATION_CODE  DATE......  ADVANCE.

CCLE001        12/16/95     6988.00
CCLE001        08/19/94
CCLE001        08/20/94
.
.
.
CIAH001        12/29/94     8286.00
Press any key to continue...
```

Setting the left margin may be of interest if you are printing the results to be bound in a notebook or manual. You also can use it to shift the image to the right on the screen. Here, MARGIN 10 produces a left margin of 10 characters:

```
>SELECT LOCATION_CODE, "DATE", ADVANCE FMT '8R'
SQL+FROM ENGAGEMENTS.T
SQL+ORDER BY LOCATION_CODE COLUMN SPACES 2 MARGIN 10;
          LOCATION_CODE  DATE......  ADVANCE.

          CCLE001        12/16/95     6988.00
```

```
            CCLE001         08/19/94
            CCLE001         08/20/94
            .
            .
            .
            CIAH001         12/29/94     8286.00
Press any key to continue...
```

NO.PAGE (or NOPAGE) suppresses automatic pagination and causes the report to scroll continuously on the screen or to print without formatted page breaks on the printer.

Use VERTICALLY (or VERT) to force output to be listed in a vertical format, that is, listing each row on a separate line:

```
>SELECT NAME, USE, COST FROM LIVESTOCK.T VERTICALLY;
NAME. Bussa
USE.. P
COST.      2694.00

NAME. Warri
USE.. Z
COST.     10198.00
.
.
.

NAME. Bongor
USE.. Z
COST.      4572.00
Press any key to continue...
```

Outputting to the System Printer

Using LPTR [*n*] directs the query output to your system printer. *n* can be from 0 through 255, indicating a logical print channel number. You can omit *n*, in which case print channel 0 is assumed. To output the results of a query to print channel number 32, enter:

```
>SELECT NAME, USE, COST FROM LIVESTOCK.T LPTR 32;
>
```

Chapter

3

Using Advanced SELECT Statements

This chapter continues with queries of singlevalued rows but adds six powerful SQL features to the command repertoire:

- Grouping queries summarizes rows into groups and then selects or rejects those groups, using the GROUP BY clause.
- Selecting groups selects or rejects groups of rows, based on selection criteria, using the HAVING clause.
- Using processing qualifiers affects or reports on the processing of SQL queries.
- Joining tables allows querying multiple tables and selects data from more than one table or file.
- Subquerying uses the results of one query as input to another query.

Grouping Rows (GROUP BY)

The set functions discussed under "Summarizing Table Contents (Set Functions)" on page 37 condensed all of the detailed data selected from a table into a single, summary row of data, much like a grand total at the bottom of a report. A single total representing the cost of all of the equipment that was purchased resulted if you asked:

```
>SELECT SUM(COST) FROM EQUIPMENT.T;
SUM ( COST )

  2995401.36

1 records listed.
```

To see the total purchases by vendor, use the UniVerse SQL GROUP BY clause:

```
>SELECT VENDOR_CODE, SUM(COST) FROM EQUIPMENT.T
SQL+GROUP BY VENDOR_CODE;
VENDOR_CODE    SUM ( COST )

          1        15704.11
          3        75334.22
          4        44809.61
         10        57355.77
          .
          .
          .
        135        94255.70
Press any key to continue...
```

This second query produces multiple summary rows, one for each vendor, and summarizes the total cost of equipment purchased from each. In this instance, the system:

1. Divides the equipment rows into groups of vendors, using the values in the grouping column, VENDOR_CODE
2. For each group, totals the values in the COST column for all of the rows
3. Generates a single summary row for each group, showing the value of VENDOR_CODE and the total cost

The GROUP BY clause divides a table into groups of similar rows, producing a single result row for each group of rows that have the same values for each column in the GROUP BY clause. Frequently, GROUP BY is combined with a set function to produce summary values for each of these sets.

A side effect of GROUP BY is that the output results are sorted by the grouping columns. This added benefit is not dictated by SQL standards, but is provided in UniVerse SQL. To override it, use an explicit ORDER BY.

The following examples use GROUP BY.

To get a count of the number of engagements booked at each location, enter:

```
>SELECT LOCATION_CODE, COUNT(*) FROM ENGAGEMENTS.T
SQL+GROUP BY LOCATION_CODE;
LOCATION_CODE    COUNT ( * )

CCLE001                    8
CDET001                    8
CDFW001                    8
.
.
.
WVGA001                    8
Press any key to continue...
```

To get a count of animals by type and the range of prices paid, enter:

```
>SELECT DESCRIPTION, COUNT(*), MIN(COST), MAX(COST)
SQL+FROM LIVESTOCK.T
SQL+GROUP BY DESCRIPTION;
DESCRIPTION    COUNT ( * )    MIN ( COST )    MAX ( COST )

Aardwolf                 2         5583.00         8977.00
Cacomistle               1        10078.00        10078.00
Camel                    4         6016.00         8661.00
Cheetah                  2         4094.00         4712.00
.
.
.
Shetland                13         1330.00         9924.00
Press any key to continue...
```

You can specify more than one grouping in the GROUP BY clause. For instance, to get a count of animals by type *and* use, enter:

```
>SELECT DESCRIPTION, USE, COUNT(*)
SQL+FROM LIVESTOCK.T
SQL+GROUP BY DESCRIPTION, USE;
DESCRIPTION    USE    COUNT ( * )

Aardwolf       Z                2
Cacomistle     Z                1
Camel          R                4
Cheetah        Z                2
Civet          Z                3
Coati          Z                2
```

```
Dhole           Z               1
Dog             P               1
Elephant        P               1
Elephant        R               1
.
.
.
Parrot          P               2
Puma            Z               2
Ratel           Z               1
Press any key to continue...
```

Restrictions on Grouping Rows

As this last example illustrates, the GROUP BY Clause allows you to see only one level of grouping at a time (that, is you cannot nest GROUP BY clauses). However, you can use certain keywords, such as BREAK ON and DET.SUP, to effectively produce multilevel totals.

A column listed in the GROUP BY clause must be an actual column, not a calculated one, and cannot be multivalued, "exploding" a multivalued column into discrete rows using the UNNEST feature changes the column to singlevalued, and you can then use it in a GROUP BY clause). Furthermore, any column in the select list must be a constant, a set function, a column listed in the GROUP BY clause, or an expression comprising some combination of these.

Null Values in Grouping Columns

Nulls are treated in a special way when they appear in a grouping column. Although SQL treats nulls as unknown values, and therefore each null could represent a *different* value, GROUP BY treats two null values found in a column as being identical, and places them into the same output grouping.

Selecting Groups (HAVING)

The HAVING Clause operates with grouped queries similarly to how the WHERE Clause operates with ungrouped queries, selecting or rejecting row groups depending on selection criteria. The selection criteria that can be used with HAVING are the same as those used with WHERE.

Instead of asking for a count of animals, enter the following to see only those animals numbering more than six:

```
>SELECT DESCRIPTION, COUNT(*) FROM LIVESTOCK.T
SQL+GROUP BY DESCRIPTION
SQL+HAVING COUNT(*) > 6;
DESCRIPTION    COUNT ( * )

Lion                    8
Shetland               13

2 records listed.
```

Another example of HAVING uses two grouping columns. Ask to see only those zoo and parade animals numbering more than one:

```
>SELECT DESCRIPTION, USE, COUNT(*) FROM LIVESTOCK.T
SQL+GROUP BY DESCRIPTION, USE
SQL+HAVING (USE = 'Z' OR USE = 'P') AND COUNT(*) > 1;
DESCRIPTION    USE    COUNT ( * )

Aardwolf       Z               2
Cheetah        Z               2
Civet          Z               3
Coati          Z               2
Fox            Z               2
Horse          P               3
.
.
.
Wolverine      Z               3

20 records listed.
```

The selection criteria in a HAVING clause must include at least one set function. Otherwise, move the search criteria to a WHERE clause and apply it to individual rows to get the same result. For instance, in the previous example, if you were not selecting on COUNT(*), you could rephrase the query as:

```
>SELECT DESCRIPTION, USE, COUNT(*) FROM LIVESTOCK.T
SQL+WHERE USE = 'Z' OR USE = 'P'
SQL+GROUP BY DESCRIPTION, USE;
DESCRIPTION    USE    COUNT ( * )

Aardwolf       Z                2
Cacomistle     Z                1
Cheetah        Z                2
Civet          Z                3
Coati          Z                2
Dhole          Z                1
Dog            P                1
Elephant       P                1
Ferret         Z                1
.
.
.
Mink           Z                2
Press any key to continue...
```

In theory, you could use a HAVING clause without a GROUP BY clause (making the entire table, in essence, a single group). However, this is not common practice.

Processing SQL Queries

Processing qualifiers affect or report on the processing of SQL queries. For example, processing qualifiers can:

- Show you how a statement will be processed
- Suppress the query optimizer
- Avoid lock delays

Showing How a Query Will Be Processed (EXPLAIN)

Use EXPLAIN in a SELECT statement to display information about how the statement will be processed, so that you can decide if you want to rewrite the query more efficiently. You can also use EXPLAIN in an INSERT, UPDATE, or DELETE statement, whenever it contains a WHERE clause or a query specification.

EXPLAIN lists the tables included in the query or WHERE clause, explains how data will be retrieved (that is, by table, select list, index lookup, or explicit ID), and explains how any joins will be processed. After each message, press **Q** to quit, or press any other key to continue the query.

If a client program uses EXPLAIN in a SELECT statement, the statement is not processed. Instead, an SQLSTATE value of IA000 is returned, along with the EXPLAIN message as the message text.

To see what the EXPLAIN display looks like, enter the following:

```
>SELECT ORDER.NO, "DATE", CUST.NO, "DESC", QTY
SQL+FROM UNNEST ORDERS ON PROD.NO, INVENTORY
SQL+WHERE ORDERS.PROD.NO = CAST(INVENTORY.PROD.NO AS INT)
SQL+EXPLAIN;


UniVerse/SQL: Optimizing query block 0
Tuple restriction: ORDERS.PROD.NO = value expression

Driver source: ORDERS
Access method: file scan

1st join primary:   ORDERS             est. cost:   73
         secondary: INVENTORY          est. cost:   42
         type:      cartesian join using scan of secondary file

Order No  Order Date  Customer No
```

```
Description...................    Qty.
10002     14 JUL 92        6518    Collapsible Felt Top Hat
1
10002     14 JUL 92         6518    White Classic Ring
1
10002     14 JUL 92         6518    Red Classic Ring
1
10002     14 JUL 92         6518    Blue Classic Ring
1
10006     22 APR 92         6518    Red Vinyl Stage Ball
3
10004     22 AUG 92         4450    Sure Balance Unicycle
1
10004     22 AUG 92        4450    Classic Polyethylene Club
9
10005     25 NOV 92         9874    Red Classic Ring
9
10003      07 MAR 92         9825    Red Juggling Bag
10
10003      07 MAR 92         9825    Blue Juggling Bag
10
10001     11 FEB 92         3456    Red Vinyl Stage Ball
7
10001     11 FEB 92         3456    Gold Deluxe Stage Torch
4
10001     11 FEB 92         3456    Sure Balance Unicycle
1
10007     06 JUL 92        9874    Classic Polyethylene Club
3

14 records listed.
```

Disabling the Query Optimizer (NO.OPTIMIZE)

The query optimizer tries to determine the most efficient way to process a SELECT statement (or an INSERT, UPDATE, or DELETE statement containing a WHERE clause or a query specification). Use NO.OPTIMIZE to disable the query optimizer when processing the WHERE clause.

To run the preceding example without using the query optimizer, enter the following:

```
>SELECT ORDER.NO, "DATE", CUST.NO, "DESC", QTY
SQL+FROM UNNEST ORDERS ON PROD.NO, INVENTORY
SQL+WHERE ORDERS.PROD.NO = CAST(INVENTORY.PROD.NO AS INT)
SQL+NO.OPTIMIZE;
```

Avoiding Lock Delays (NOWAIT)

Normally when a SELECT statement tries to access a row or table locked by another user or process, it waits for the lock to be released, then continues processing. Use the NOWAIT keyword to stop processing when a statement encounters a record or file lock. If the statement is used in a transaction, processing stops and the transaction is rolled back. The user ID of the user who owns the lock is returned to the terminal screen or to the client program.

If a SELECT statement with NOWAIT selects an I-descriptor or an EVAL expression that executes a UniVerse BASIC subroutine, the NOWAIT condition applies to all the SQL operations in the subroutine.

You cannot use NOWAIT in a subquery or a view definition.

***Note**: At isolation level 0 or 1, a SELECT statement never encounters the locked condition.*

If the query in the next example encounters a lock set by another user, it terminates immediately; it does not wait for the lock to be released:

```
>SELECT ORDER.NO, "DATE", CUST.NO, "DESC", QTY
SQL+FROM UNNEST ORDERS ON PROD.NO, INVENTORY
SQL+WHERE ORDERS.PROD.NO = CAST(INVENTORY.PROD.NO AS INT)
SQL+NO.OPTIMIZE NOWAIT;
```

Joining Tables

An important feature of SQL is table joins, the ability to retrieve information from more than one table. Thus far, the examples have referred to only one SQL table per query. The diagram of the Circus database shows that many of the tables are related to one another, as expected in a *relational* database.

For example, the ENGAGEMENTS.T table has a LOCATION_CODE that links to a LOCATIONS.T table, an ACT_NO that links to an ACTS.T table, and a RIDE _ID that links to a RIDES.T table. Likewise, the RIDES.T and ACTS.T tables link to the PERSONNEL.T, LIVESTOCK.T, and EQUIPMENT.T tables. And both the INVENTORY.T and EQUIPMENT.T tables link to the VENDORS.T table. All of which strongly implies that there are many times when you might want to query two or more of these tables in the same statement.

As with RetrieVe, you can obtain a similar result by using the TRANS function in an I-descriptor. However, in RetrieVe you can do so only if the relationship between the two files is defined in the dictionary, and the first file has the primary key of the other file in one of its fields.

In UniVerse SQL, you also can use I-descriptors, but it is much easier and simpler to use an impromptu join in your SELECT statement.

Technically, an SQL table join combines information from two or more tables on the basis of join conditions that describe the relationships among the tables. Before discussing table joins, let's look briefly at Cartesian "joins."

Cartesian Joins

If you refer to multiple tables in a SELECT statement that does not explicitly use a join condition among the tables, the output consists of rows representing every possible *combination* of rows from those tables. This is commonly called the *Cartesian product* (or simply the *product*) of the two tables. This combined output almost always is meaningless and misleading. Therefore querying multiple tables without specifying a join condition is not recommended, as shown in the following example:

```
>SELECT DESCRIPTION, COMPANY
SQL+FROM EQUIPMENT.T, VENDORS.T
SQL+ORDER BY DESCRIPTION;
DESCRIPTION.............    COMPANY.................

                           Associated Interests
```

```
                              Corporate Professionals
                              Silver Assemblies
                              Midwest Intercontinental
                              Financial Wares
                              City Manufacturers
                              .
                              .
                              .
Air Compressor                Commerce Exchange
Air Compressor                Eastern International
Air Compressor                Red Controls
Air Compressor                Independent Stocks
Air Compressor                Universal Devices
Air Compressor                Indiana Management
Air Compressor                Country Traders
Press any key to continue...
```

What you wanted was a list of equipment assets and the vendors from whom they were purchased. SQL combined *every* item of equipment with *every* vendor, producing an output result of over 5,000 rows (all the rows from the EQUIPMENT.T table multiplied by all the rows from the VENDORS.T table). The result is misleading because the list makes it appear as if an air compressor (and every other kind of equipment) was purchased from each vendor.

Joining Two Tables

Specifying a join condition between the EQUIPMENT.T and VENDORS.T tables would have produced the intended result. The two tables each have a column on which you can construct a join based on matching values: the VENDOR_CODE column of the EQUIPMENT.T table contains the vendor numbers that correspond to the values in the VENDOR_CODE column of the VENDORS.T table.

An equi-join is a condition based on the equalities between two columns in the two tables being joined. Rephrase the previous query to use an equi-join to get the intended result:

```
>SELECT DESCRIPTION, COMPANY
SQL+FROM EQUIPMENT.T, VENDORS.T
SQL+WHERE EQUIPMENT.T.VENDOR_CODE = VENDORS.T.VENDOR_CODE
SQL+ORDER BY DESCRIPTION;
DESCRIPTION.............    COMPANY.................

                              Illinois Operations
Air Compressor                London Treating
Amplifiers                    Reliable Wholesale
Balloon Vending Stand         Elite Salvage
Beer Keg Stand                Greek Cousins
Calliope                      Rembrandt Rentals
```

```
Cash Register                Beacon Components
Coffee/cookies Stand         European Plus
Computer                     Immediate Enterprises
Cooling System               Custom Group
Copier                       Miami Acceptances
.
.
.
Merry-Go-Round               Provencher Providers

Press any key to continue...
```

This time the output result has much fewer rows, and the information in each row is properly related. The WHERE clause names one column from each of the two tables listed in the FROM clause. It was necessary to qualify the column name VENDOR_CODE, which appears in both the EQUIPMENT.T and VENDORS.T tables, by its respective table name to indicate the table to which you are referring.

In addition to the equi-join on values common to both tables, you can add other join selection criteria, even criteria based on inequalities and using a relational operator other than "=". Assume that you have two versions of the VENDORS.T table, VEND1.T and VEND2.T, and you want to list those vendors in VEND1.T whose third address line is different in VEND2.T. First you join VEND1.T and VEND2.T on their VENDOR_CODE columns, respectively, and then select only those rows where ADR3 differs (is unequal) between the tables. Again, because both tables employ identical column names, the column names must be qualified:

```
>SELECT VEND1.T.VENDOR_CODE, VEND1.T.ADR3, VEND2.T.ADR3
SQL+FROM VEND1.T, VEND2.T
SQL+WHERE VEND1.T.VENDOR_CODE = VEND2.T.VENDOR_CODE AND
SQL+VEND1.T.ADR3 <> VEND2.T.ADR3;
```

UniVerse SQL processes standard table joins, also known as *inner joins*, in one of the following ways:

- If one of the columns is a primary key and you use an equi-join (=), UniVerse SQL retrieves the matching row directly, much like using the TRANS function in an I-descriptor. However, it is more efficient because there is no UniVerse BASIC code to execute.
- If neither of the columns is a primary key, UniVerse uses a secondary index to join the tables.
- If there is no index, UniVerse SQL tries to use a sort-merge-join.
- All other joins are processed using a Cartesian product. For example, for each row in the first table, the entire second table is scanned for matching rows. This is a slow process with large tables.

Qualifying Column Names

Related tables often share identical column names. For example, the Circus database has a vendor ID column named VENDOR_CODE, which appears in the VENDORS.T, INVENTORY.T, and EQUIPMENT.T tables. To write a query referring to two of these tables and refer to VENDOR_CODE, you need some method to indicate *which* VENDOR_CODE column in which table you mean.

In such situations, you must *qualify* any ambiguous column name by prefacing it with the appropriate table name and a period, for example, *tablename.columnname*. In a previous example, VENDOR_CODE and ADR3 are column names that appear in both the VEND1.T and VEND2.T tables. When they were used in the query, they were qualified with their respective table names, "VEND1.T" and "VEND2.T".

SQL offers a shorthand way of specifying table qualifiers through the use of *table aliases*. In a previous example, rather than entering "VEND1.T" and "VEND2.T" as part of each qualified column name, assign a shorter alias to each of the two tables and then use those aliases as table qualifiers. Specify an alias for a table immediately after its table name in the FROM clause:

FROM *tablename alias*…

For example, if you assigned "A" as the alias for VEND1.T and "B" as the alias for VEND2.T, enter the query as:

```
>SELECT A.VENDOR_CODE, A.ADR3, B.ADR3
SQL+FROM VEND1.T A, VEND2.T B
SQL+WHERE A.VENDOR_CODE = B.VENDOR_CODE AND A.ADR3 <> B.ADR3;
```

Such shorthand is valuable when you have identically named tables in different databases and systems, neither of which is your current database or system. Rather than entering long *table.column* names, just assign a single-character alias to each one.

Joining Three or More Tables

Joining several tables is really no different from joining two tables—just add a join condition for each pair of tables to be joined.

As an example, a new table, ACCTS.T, has been added to the database. It contains information concerning the accounts with each vendor and it is related to VENDORS.T through a vendor number stored in VENDOR_CODE. To see the balances due (AMOUNT_DUE) for any of the vendors from whom tents were purchased, enter:

```
>SELECT DESCRIPTION, COMPANY, AMOUNT_DUE
SQL+FROM EQUIPMENT.T, VENDORS.T, ACCTS.T
SQL+WHERE EQUIPMENT.T.VENDOR_CODE = VENDORS.T.VENDOR_CODE
SQL+AND VENDORS.T.VENDOR_CODE = ACCTS.T.VENDOR_CODE
SQL+AND DESCRIPTION LIKE '%Tent%';
```

UniVerse SQL joins the EQUIPMENT.T, VENDORS.T, and ACCTS.T tables, using vendor number as the match, and then retrieves the requested data.

Joining a Table to Itself

Sometimes it is useful to join a table to itself. This is called a *reflexive join* or a *self join*. For example, if each row in a PERSONNEL.T table contained not only the employee number (BADGE_NO) of that employee, but also the employee number of that employee's manager (MGR), you could join the PERSONNEL.T table to itself to look up the name of an employee's manager.

In a relational database, you usually expect to find a second table with data about managers, and then the situation would be a typical two-table join. But since there is no second table, imagine that there are *two* copies of the PERSONNEL.T table, one called EMPLOYEES.T and the other called MANAGERS.T. The MGR column of the EMPLOYEES.T table would be a foreign key pointing to the MANAGERS.T table. Use the query:

```
>SELECT EMPLOYEES.T.NAME, MANAGERS.T.NAME
SQL+FROM EMPLOYEES.T, MANAGERS.T
SQL+WHERE EMPLOYEES.T.MGR = MANAGERS.T.BADGE_NO;
```

Theoretically, this "duplicate table" approach is how UniVerse SQL joins a table to itself. Instead of physically duplicating the table, UniVerse SQL lets you refer to it by a different name using table aliases (see "Qualifying Column Names" on page 15). Rewrite the previous query to assign EMPLOYEES and MANAGERS as aliases:

```
>SELECT EMPLOYEES.NAME, MANAGERS.NAME
SQL+FROM PERSONNEL.T EMPLOYEES, PERSONNEL.T MANAGERS
SQL+WHERE EMPLOYEES.MGR = MANAGERS.BADGE_NO;
```

The change is minor except for the definition of the two table aliases. You could assign just one alias, PERSONNEL MANAGERS, for instance, and use the table's own name, PERSONNEL.T, for the other.

Analyzing a Table Join

Showing How a Query Will Be Processed (EXPLAIN) in the *UniVerse SQL Reference* provides an analysis of how UniVerse SQL processes a table join. To produce this analysis, add the keyword EXPLAIN. The report that is produced lists the tables in the query, indicates how the data will be retrieved, and shows the estimated I/O costs. Information is provided for each query block (SELECT clause, subquery, and so on) that specifies multiple tables. Using this information, you can decide if you want to continue the query, restate the query more efficiently, or bypass it altogether.

Take one of the previous queries and add the EXPLAIN keyword:

```
>SELECT DESCRIPTION, COMPANY, COST CONV 'MD2$', USE_LIFE
SQL+FROM EQUIPMENT.T, VENDORS.T
SQL+WHERE EQUIPMENT.T.VENDOR_CODE = VENDORS.T.VENDOR_CODE
SQL+EXPLAIN;
UniVerse/SQL: Optimizing query block 0
Tuple selection criteria:
TRUE

Driver source: EQUIPMENT.T
Access method: file scan

1st join primary:   EQUIPMENT.T        est. cost:   61
         secondary: VENDORS.T          est. cost:   191
         type:      hashed access (primary key)

UniVerse/SQL: Press any key to continue or 'Q' to quit
```

The analysis names the keys on which the equi-join is based and tells you the I/O costs of accessing each table and the type of access that will be used.

Outer Joins

The outer join of one table to another differs from an inner join in that the resulting table may contain additional rows beyond what would be in the same tables joined by an inner join: one additional row is produced for each row in the first table specification (the outer, or left table) that does not meet the join condition against any row in the second table specification. An outer join is also known as a *left outer join*.

The next two queries illustrate the distinction between an inner join and an outer join. The first example shows the vendor name and description for every part sold by a vendor whose name starts with *H*. *AA* and *BB* are table aliases (also known as correlation names), used to simplify the language of the queries:

```
>SELECT AA.VENDOR_CODE, COMPANY, DESCRIPTION
SQL+FROM VENDORS.T AS AA INNER JOIN EQUIPMENT.T AS BB
SQL+ON AA.VENDOR_CODE = BB.VENDOR_CODE
SQL+WHERE COMPANY LIKE 'H%'
SQL+ORDER BY COMPANY;

VENDOR_CODE  COMPANY.................  DESCRIPTION............
        145  Hollywood Retail          Truck 246 YGN
         29  Houston Professionals     Truck 588 RWJ
         29  Houston Professionals     Security System
        146  Hub Sales                 Zoo Fencing
4 records listed.
```

This example also could have been written as the following ordinary join:

```
>SELECT AA.VENDOR_CODE, COMPANY, DESCRIPTION
SQL+FROM VENDORS.T AS AA, EQUIPMENT.T AS BB
SQL+WHERE AA.VENDOR_CODE = BB.VENDOR_CODE
SQL+AND COMPANY LIKE 'H%'
SQL+ORDER BY COMPANY;
```

The second example demonstrates an outer join, showing the same information as the first example, but also including vendors whose names start with *H* but who do not sell any parts:

```
>SELECT AA.VENDOR_CODE, COMPANY, DESCRIPTION
SQL+FROM VENDORS.T AS AA LEFT OUTER JOIN EQUIPMENT.T AS BB
SQL+ON AA.VENDOR_CODE = BB.VENDOR_CODE
SQL+WHERE COMPANY LIKE 'H%'
SQL+ORDER BY COMPANY;

VENDOR_CODE   COMPANY.................  DESCRIPTION............
        174   Harvard Consultants
        141   High Innovations
         92   Hill Marketing
        145   Hollywood Retail          Truck 246 YGN
         59   Hong Kong Imports
         29   Houston Professionals     Truck 588 RWJ
         29   Houston Professionals     Security System
        146   Hub Sales                 Zoo Fencing

8 records listed.
```

Selecting on Joined Tables

When retrieving information from multiple joined tables using the SELECT statement, the *tablename.** syntax is useful. In a SELECT list, the sublist *tablename.** means "all columns of *tablename*" and is equivalent to specifying each column in *tablename*.

The following specifics apply to using *tablename.**:

- If *tablename* is a given correlation name *corrname* in the FROM clause, then you cannot use *tablename.** as a sublist. Use *corrname.** instead.
- A sublist of the form *schemaname.tablename.** is supported, but only if *schemaname.tablename* appears in the FROM clause.
- A sublist of the form *tablename_assocname.** is supported, but only if *tablename_assocname* appears in the FROM clause.
- A sublist of the form *filename.**, where *filename* is not a table, is supported if *filename* appears in the FROM clause. In this case, the asterisk means "fields defined by the @ phrase."

Using UNION to Combine SELECT Statements

Combine two or more SELECT statements into a single result table using the UNION operator. When a set of SELECT statements is joined by a UNION operator, it collectively is called a query expression.

A query expression that contains the keyword UNION must satisfy the following rules:

- INQUIRING is not allowed in the FROM clause.
- Field modifiers are not allowed.
- The only allowed field qualifiers are AS, FMT, CONV, DISPLAYNAME, and DISPLAYLIKE. Except for AS, these field qualifiers must appear in the first SELECT of the query.

To specify that duplicate rows not be removed in the result table, add ALL to the query. If you do not specify ALL, duplicate rows are removed.

Use query expressions as interactive SQL queries and programmatic SQL queries, and in the CREATE VIEW statement. Query expressions cannot be used as a subquery or in the INSERT statement.

SQL processes SELECT statements joined by UNION from left to right. Specify a different processing order by using parentheses. You cannot enclose the entire query expression in parentheses, however.

In addition, column names, column headings, formats, and conversions used in the result table are taken from the first SELECT statement. All SELECT statements combined using the UNION operator must specify the same number of result columns. Corresponding columns among the SELECT statements must belong to the same data category (character, number, date, or time).

This example uses UNION to show all personnel and all act locations with telephone numbers in the 617 area code:

```
>SELECT NAME DISPLAYNAME 'NAME or LOCATION', PHONE FROM
PERSONNEL.T
SQL+WHERE PHONE LIKE '617%'
SQL+UNION
SQL+SELECT NAME, PHONE FROM LOCATIONS.T WHERE PHONE LIKE '617%'
SQL+ORDER BY 1;

NAME or LOCATION..........  PHONE........

Anderson, Suzanne           617/451-1910
Bacon, Roger                617/562-3322
Boston Properties, Inc.     617/565-5859
Carter, Joseph              617/360-6667
Palumbo, Mark               617/541-5373

5 records listed.
```

DISPLAYNAME is used in the first SELECT to provide a sensible column heading, and an ORDER BY clause is used to sort the output alphabetically.

Subqueries

The concept of subqueries, the ability to use a query within another query, is what originally gave the name “structured” to Structured Query Language. Subqueries are a powerful SQL feature that:

- Permits the writing of queries that more closely parallel their English-language equivalent
- Aids querying by letting you reduce a complex query into “bite-sized” pieces
- Enables you to construct queries that cannot be written in any other way

Subqueries often provide an alternative to two separate SELECT statements or a SELECT statement with a multitable join.

A subquery is a SELECT statement (called the *inner* SELECT) that is nested in another SELECT statement (called the *outer* SELECT), or in an INSERT, DELETE, or UPDATE statement. Like all SELECTs, a subquery SELECT must have a SELECT clause and a FROM clause. It optionally can include WHERE, GROUP BY, and HAVING clauses. When used, the subquery is enclosed in parentheses and is part of the WHERE or HAVING clause of the outer SELECT:

SELECT… WHERE… [ALL | ANY | IN | EXISTS] (SELECT *subquery***)**

Subqueries can be nested to a depth of nine levels. A subquery SELECT differs in several aspects from a regular SELECT in that it:

- Can specify only one select item (column), except in the case of EXISTS
- Cannot include the ORDER BY clause, INQUIRING, or any field modifiers, field qualifiers, or report qualifiers

A less frequent use of a subquery is including it as part of a HAVING clause (see “Using Subqueries with HAVING” on page 31).

Correlated and Uncorrelated Subqueries

Subqueries are classified as *correlated* or *uncorrelated*. A subquery is correlated when the results it produces depend on the results of the outer SELECT statement that contains it. All other subqueries are considered uncorrelated.

Correlated Subqueries

A correlated subquery is executed repeatedly, once for each value produced by the outer SELECT. For example, this correlated subquery lists the 10 youngest employees on staff:

```
>SELECT DOB, NAME FROM PERSONNEL.T MAINSEL
SQL+WHERE 10 >
SQL+(SELECT COUNT(DOB) FROM PERSONNEL.T SUBQ
SQL+WHERE SUBQ.DOB > MAINSEL.DOB)
SQL+AND DOB IS NOT NULL
SQL+ORDER BY DOB;
DOB.......    NAME.....................

02/03/73      Wang, Isabel
08/06/73      Dickinson, Timothy
08/29/73      Ellsworth, Leonard
10/26/73      Friedrich, Linda
12/10/73      Young, Pamela
12/15/73      Schultz, Mary Lou
01/03/74      Giustino, Carol
09/06/74      Kozlowski, Bill
10/04/74      Parker, Leslie
12/04/74      Torres, Stephen

10 records listed.
```

This subquery is correlated because the value it produces depends on MAINSEL.DOB, the value produced by the outer SELECT. The subquery must execute once for every row that the outer SELECT considers. Within the subquery, COUNT(DOB) returns a value to the outer SELECT.

Uncorrelated Subqueries

An uncorrelated subquery is executed only once, and the result of executing the subquery is the return of no value, one value, or a set of values to the outer SELECT. Uncorrelated subqueries probably are the most common, and are shown in the next examples.

Subquery Test Types

Subqueries always are part of the selection criteria in the WHERE Clause or HAVING Clause of which they are a part and fall into three basic types:

- **Comparison test (=, <>, #, <, <=, >, >=):** Compares an expression to the results of the outer SELECT statement. Quantified comparisons, which use the keyword ANY or ALL, are an extension of this.
- **Match test (IN):** Determines whether a value is included in the results of the inner SELECT statement.
- **Existence test (EXISTS):** Determines whether any rows were selected by the inner SELECT statement.

Comparison Test (=, <>, #, <, <=, >, >=)

A subquery comparison test uses the same operators as in the examples of simple comparisons. It compares the value of each row selected by the outer SELECT with the *single* value produced by the subquery.

Unless you use ANY or ALL (see "Quantified Comparisons" on page 26), the subquery in a comparison test must produce only a single row. If it produces multiple rows, SQL reports an error; try using the match test instead. If the subquery produces no rows or a null value, a null will be returned. For example, list the animals with an estimated life span longer than animal 67 (which has an estimated life span of 16 years):

```
>SELECT ANIMAL_ID, DESCRIPTION, EST_LIFE FROM LIVESTOCK.T
SQL+WHERE EST_LIFE > (SELECT EST_LIFE FROM LIVESTOCK.T
SQL+WHERE ANIMAL_ID = 67) ORDER BY ANIMAL_ID;
ANIMAL_ID    DESCRIPTION    EST_LIFE

        2    Mink                17
        3    Otter               18
        4    Lion                18
       16    Lion                18
       23    Tiger               18
       .
       .
       .
       80    Puma                19
       81    Tiger               17

25 records listed.
```

The processing sequence produced by this query follows. First, the (SELECT EST_LIFE FROM LIVESTOCK.T WHERE ANIMAL_ID = 67) subquery produces a single row consisting of the EST_LIFE value of the row where ANIMAL_ID equals 67. Then the outer SELECT compares the EST_LIFE value of every row of the LIVESTOCK.T table to that value and selects any rows whose value is higher. This is an example of an uncorrelated subquery.

In another example, list the inventory items that have a quantity on hand that is less than the average quantity on hand for all the inventory:

```
>SELECT DESCRIPTION, QOH FROM INVENTORY.T
SQL+WHERE QOH < (SELECT AVG(QOH) FROM INVENTORY.T);
DESCRIPTION.............    QOH..

Mustard                        125
French Fries, Frozen            51
Crabcakes                       87
.
.
.
Large Cat Chow                 127
Bird Seed                       94

22 records listed.
```

The subquery (SELECT AVG(QOH)FROM INVENTORY.T) is evaluated first, calculating the average QOH for the entire inventory and producing a single row containing that average. Then the QOH in each row of the INVENTORY.T table is compared to this value, and those rows with a QOH below the average are output.

This example of a subquery comparison uses two different tables to list the engagements booked for Houston:

```
>SELECT LOCATION_CODE, DATE FROM ENGAGEMENTS.T
SQL+WHERE LOCATION_CODE =
SQL+(SELECT LOCATION_CODE FROM LOCATIONS.T
SQL+WHERE ADR3 LIKE 'HOUSTON%');
LOCATION_CODE    DATE......

CIAH001          10/03/95
CIAH001          10/04/95
CIAH001          01/17/97
CIAH001          09/16/92
CIAH001          01/18/97
CIAH001          12/28/94
CIAH001          09/17/92
CIAH001          12/29/94

8 records listed.
```

Quantified Comparisons

The keywords ALL and ANY can be used to modify a subquery comparison to make it into a quantified comparison. Subqueries in *quantified* comparisons can return more than one row.

Using the ALL Quantifier

The ALL quantifier probably is the most commonly used quantifier. Here, the test value is compared to each value in the column of values produced by the subquery, one at a time. If all individual comparisons return a true result, or if no values are returned, the result is true.

To obtain a list of every inventory item that has a cost higher than the cost of all the animal chows, enter:

```
>SELECT DESCRIPTION, COST FROM INVENTORY.T
SQL+WHERE COST > ALL (SELECT COST FROM INVENTORY.T
SQL+WHERE DESCRIPTION LIKE '%Chow%');
DESCRIPTION.............    COST........

Cookies                             98.32
Cola                               102.83
Franks                              99.92
Egg Rolls                          103.80

4 records listed.
```

It turns out that animal feed runs from $11 for elephant chow up to $96 for domestic dog chow; you can see in the results a list of all items that cost more than $96. A good way to think of the ALL test is to read the previous statement as “Select those rows where COST is greater than *all* of the ‘animal chow’ COSTs.”

Using the ANY Quantifier

Use the ANY quantifier (or its synonym SOME) to determine if a subquery comparison is true for at least one of the values returned by the subquery. The value being tested is compared to each value in the subquery results, one at a time. If *any* comparison is true, a true result is returned. If the subquery returns no values, the result is false.

Look at the result if you change the quantifier in the previous example from ALL to ANY:

```
>SELECT DESCRIPTION, COST FROM INVENTORY.T
SQL+WHERE COST > ANY (SELECT COST FROM INVENTORY.T
SQL+WHERE DESCRIPTION LIKE '%Chow%');
DESCRIPTION.............    COST........

Mustard                             91.52
French Fries, Frozen                34.95
Crabcakes                           28.53
.
```

```
.
.
Cheese Slices                          88.21
Ice Bags                               69.23

43 records listed.
```

A longer list is returned because it contains any item whose cost exceeds any one of the animal chows, and since $11 is the cost of the elephant chow, the query selected any item (including any other animal chows) costing more than $11. Think of the previous SQL statement as reading "Select those rows where COST is greater than *any one* of the 'animal chow' COSTs."

Using ANY can be tricky, especially when used with the inequality operator (<> or #). To see a list of trucks that were *not* bought from a company whose name begins with *H*, enter:

```
>SELECT DESCRIPTION, COMPANY FROM EQUIPMENT.T, VENDORS.T
SQL+WHERE EQUIPMENT.T.VENDOR_CODE = VENDORS.T.VENDOR_CODE
SQL+AND DESCRIPTION LIKE '%Truck%'
SQL+AND EQUIPMENT.T.VENDOR_CODE <> ANY
SQL+(SELECT VENDORS.T.VENDOR_CODE FROM VENDORS.T
SQL+WHERE COMPANY LIKE 'H%');
DESCRIPTION.............    COMPANY.................

Truck 897 M X X             Beacon Components
Truck 102 T I U             King Finishing
Truck 413 X H K             Veterans Advisers
Truck 665 B C C             Ohio Treating
Truck 588 R W J             Houston Professionals
Truck 243 Y G N             Hollywood Retail
Truck 821 N H Y             Boston Equipment
Truck 212 A Q S             Accurate Surplus

8 records listed.
```

You get *all* trucks selected, because the *subquery* produces a two-row table that contains the vendor codes for two different companies that begin with *H*. Because the vendor code being tested always fails to match at least one of the two vendor codes in the subquery result, the <> comparison always tests true, even when the source *does* begin with an *H*, and all trucks, including those purchased from Houston Professionals and Hollywood Retail, appear in the result. The correct query is:

```
>SELECT DESCRIPTION, COMPANY FROM EQUIPMENT.T, VENDORS.T
SQL+WHERE EQUIPMENT.T.VENDOR_CODE = VENDORS.T.VENDOR_CODE
SQL+AND DESCRIPTION LIKE '%Truck%'
SQL+AND NOT (EQUIPMENT.T.VENDOR_CODE = ANY
SQL+(SELECT VENDORS.T.VENDOR_CODE FROM VENDORS.T
SQL+WHERE COMPANY LIKE 'H%'));
DESCRIPTION.............    COMPANY..................

Truck 897 M X X             Beacon Components
Truck 102 T I U             King Finishing
Truck 413 X H K             Veterans Advisers
Truck 665 B C C             Ohio Treating
Truck 821 N H Y             Boston Equipment
Truck 212 A Q S             Accurate Surplus

6 records listed.
```

This time any trucks bought from companies with names beginning with *H* were not selected.

Changing ANY to EXISTS

You can always turn an ANY test into an EXISTS test by moving the comparison inside the subquery. Doing so helps avoid the kinds of confusion being discussed. Take the current example and rewrite it as an EXISTS:

```
>SELECT DESCRIPTION, COMPANY FROM EQUIPMENT.T, VENDORS.T
SQL+WHERE EQUIPMENT.T.VENDOR_CODE = VENDORS.T.VENDOR.CODE
SQL+AND DESCRIPTION LIKE '%Truck%'
SQL+AND NOT EXISTS (SELECT VENDORS.T.VENDOR_CODE FROM VENDORS.T
SQL+WHERE COMPANY LIKE 'H%'
SQL+AND EQUIPMENT.T.VENDOR_CODE = VENDORS.T.VENDOR_CODE);
DESCRIPTION.............    COMPANY..................

Truck 897 M X X             Beacon Components
Truck 102 T I U             King Finishing
Truck 413 X H K             Veterans Advisers
Truck 665 B C C             Ohio Treating
Truck 821 N H Y             Boston Equipment
Truck 212 A Q S             Accurate Surplus

6 records listed.
```

Again, the correct results are returned.

Match Test (IN)

Using the IN Keyword in the *UniVerse SQL Reference* compares the test value to a column of data values produced by the subquery and returns a true result if the test value matches any of the values in the column. Therefore, the IN keyword is equivalent to = ANY.

List the engagements that are booked into sites having at least 6,000 seats:

```
>SELECT ENGAGEMENTS.T.LOCATION_CODE, DATE, SEATS
SQL+FROM ENGAGEMENTS.T, LOCATIONS.T
SQL+WHERE ENGAGEMENTS.T.LOCATION_CODE =
SQL+LOCATIONS.T.LOCATION_CODE
SQL+AND ENGAGEMENTS.T.LOCATION_CODE IN
SQL+(SELECT LOCATION_CODE FROM LOCATIONS.T
SQL+WHERE SEATS >= 6000)
SQL+ORDER BY ENGAGEMENTS.T.LOCATION_CODE, DATE;
LOCATION_CODE    DATE......    SEATS

CIND001          02/13/93       8000
CIND001          02/14/93       8000
CIND001          02/18/94       8000
CIND001          02/19/94       8000
CIND001          08/19/94       8000
CIND001          08/20/94       8000
CIND001          10/04/94       8000
CIND001          10/05/94       8000
CMIL001          03/04/93       6000
CMIL001          03/05/93       6000
.
.
.
WVGA001          02/12/97       8000
WVGA001          02/13/97       8000

72 records listed.
```

You also could have entered this as:

```
>SELECT ENGAGEMENTS.T.LOCATION_CODE, DATE, SEATS
SQL+FROM ENGAGEMENTS.T, LOCATIONS.T
SQL+WHERE ENGAGEMENTS.T.LOCATION_CODE =
SQL+LOCATIONS.T.LOCATION_CODE
SQL+AND ENGAGEMENTS.T.LOCATION_CODE = ANY
SQL+(SELECT LOCATION_CODE FROM LOCATIONS.T
SQL+WHERE SEATS >= 6000)
SQL+ORDER BY ENGAGEMENTS.T.LOCATION_CODE, DATE;
```

Existence Test (EXISTS)

The EXISTS test, sometimes called an existential qualifier, checks to see if a subquery produces any rows of results at all. If the subquery select criteria produces results, a true result is returned; if not, a false result is returned.

Because the EXISTS test does not actually *use* the results of the subquery, the rule about a subquery returning only a single column of results is waived, and you can use a SELECT *. In fact, there is no reason to use anything else.

To list the engagements scheduled in Washington state, enter:

```
>SELECT ENGAGEMENTS.T.LOCATION_CODE, DATE, ADR3
SQL+FROM ENGAGEMENTS.T, LOCATIONS.T
SQL+WHERE ENGAGEMENTS.T.LOCATION_CODE =
SQL+LOCATIONS.T.LOCATION_CODE
SQL+AND EXISTS (SELECT * FROM LOCATIONS.T
SQL+WHERE ENGAGEMENTS.T.LOCATION_CODE =
SQL+LOCATIONS.T.LOCATION_CODE
SQL+AND ADR3 LIKE '% WA %');
LOCATION_CODE    DATE......    ADR3.....................

WSEA001          12/07/95      SEATTLE WA 96030
WSEA001          12/08/95      SEATTLE WA 96030
WSEA001          06/04/95      SEATTLE WA 96030
WSEA001          06/05/95      SEATTLE WA 96030
WSEA001          02/16/95      SEATTLE WA 96030
WSEA001          02/17/95      SEATTLE WA 96030
WSEA001          02/21/96      SEATTLE WA 96030
WSEA001          02/22/96      SEATTLE WA 96030

8 records listed.
```

This is one of those instances where it would be simpler to use a table join instead of a subquery:

```
>SELECT ENGAGEMENTS.T.LOCATION_CODE, DATE, ADR3
SQL+FROM ENGAGEMENTS.T, LOCATIONS.T
SQL+WHERE ENGAGEMENTS.T.LOCATION_CODE =
SQL+LOCATIONS.T.LOCATION_CODE
SQL+AND ADR3 LIKE '% WA %';
```

Using Subqueries with HAVING

Although subqueries are more commonly found in WHERE clauses, you also can use them in HAVING clauses. An example of how you would use a subquery in a HAVING clause is a case where you list all the animals, grouped by use, with an estimated life span greater than the average estimated life span of all the animals:

```
>SELECT USE, NAME, EST_LIFE FROM LIVESTOCK.T
SQL+GROUP BY USE, NAME, EST_LIFE
SQL+HAVING EST_LIFE > (SELECT AVG(EST_LIFE) FROM LIVESTOCK.T);
USE    NAME......    EST_LIFE

P      Bassar              18
P      Bauchi              18
P      Bussa               18
P      Foula               16
.
.
.
Z      Mongo               15
Press any key to continue...
```

This subquery computed the average estimated life span (14.26 years) of all the animals in the LIVESTOCK.T table, then compared the estimated life of each animal to that average, selecting those that are greater than the average.

Chapter

4

Selecting on Multivalued Columns

This chapter explains how to use UniVerse SQL to access and manipulate data stored in UniVerse's multivalued columns. It also discusses dynamic normalization.

The real world operates in ways that are difficult to represent in a database. Real-world data often is multilayered—people have multiple charge accounts, more than one dependent, several telephone numbers at which they can be reached, and a long string of places they have lived, schools they have attended, and companies for which they have worked.

Most conventional relational databases can hold only a singlevalue in each *cell* (the intersection of a column and a row). Multiple values are consigned to a separate table and linked to their associated rows using a common key field. UniVerse is designed to handle multivalued columns, which enable a single cell to hold an array of values, with each value separated from the next by a delimiter called a value mark.

Uses for Multivalued Columns

One common use of a multivalued column is to store a where-used list. Suppose you have a particular part that is a component of many different products. A multivalued column in a PARTS record would contain the primary key values of all the PRODUCT records for products that include that part. As another example, you might have a number of people working on a particular project. A multivalued column in a PROJECT record would contain the employee IDs of all the people working on that project. In the Circus database, an example of a where-used list is the multivalued column EQUIP_CODE in the VENDORS.T table, which lists the IDs of the equipment purchased from each vendor.

Another use for multivalued columns is to store a small number of alternate pieces of information. A PHONE column could contain more than one phone number per row: a primary contact number, alternate numbers, a fax number, and even an Internet address. In the Circus database, the three different kinds of vaccination given the circus animals are stored in a group of multivalued columns in the LIVESTOCK.T table.

In conventional database management systems, you must create your own secondary tables to handle these relatively common situations. Sometimes it may be beneficial to store such information in separate tables, for ease of updating, for example. However, such separation often created extra work for the implementors and unnecessary overhead for the system.

With UniVerse SQL, you have the best of both worlds. You can use multivalued columns *and* related tables, choosing the approach best suited to your needs.

Associations

You can group related multivalued columns together as an *association*. This means that, in each row, the first value in one multivalued column of an association has a one-to-one relationship to the first values in all the other associated columns, all the second position values in each associated column have the same relationship, and so forth. An association, therefore, is an array of columns containing related multivalues and, in effect, can be thought of as a *nested table* or a *table within a table*.

To extend one of the earlier examples, suppose that people are assigned to a project for a finite period of time. In UniVerse, you could create three associated multivalued columns in the PROJECT.T table for the employee ID, his or her start date on the project, and the scheduled release date. It is necessary to define these three columns as an association, because the start and release dates are not just a pile of dates, but each one is associated with an employee.

Although not required, many associations are generated specifying a key. An association key, when combined with the primary key of the base table (or the @ID column if the table has no primary key), can be thought of as the primary key of the "table within a table" that the association represents. If you designate only one column as the key, that column automatically has the column constraint ROWUNIQUE. The *depth* of an association (number of nested association rows within a particular base table row) is determined by the maximum number of values in any of its key columns, or in all of its columns if there is no association key.

Multivalued Columns in the Sample Database

The Circus database has many examples of multivalued columns. INVENTORY.T, for example, shows how such columns are used.

In INVENTORY.T, multivalued columns store an order history for each inventory item. The ordering information is stored in the columns named VENDOR_CODE and ORDER_QTY, which record the vendor from which the item was purchased and the quantity purchased. These columns are related to one another through an association called ORDERS_ASSOC, which tells the system that the first value in the VENDOR_CODE column for a row is associated with the first value in the ORDER_QTY column for that row.

To see the orders for hot dog buns, enter:

```
>SELECT DESCRIPTION, VENDOR_CODE, ORDER_QTY
SQL+FROM INVENTORY.T
SQL+WHERE DESCRIPTION = 'Hot Dog Buns';
DESCRIPTION.............    VENDOR_CODE    ORDER_QTY

Hot Dog Buns                        43          300
                                   231          700
                                    63          700
                                    13          500
                                   210          700
                                    36          500
                                    34          500
                                    67          900
1 records listed.
```

The output shows that eight orders have been placed for hot dog buns. In UniVerse, data about all eight orders can be stored in a *single* row of the INVENTORY.T table, the row for hot dog buns.

If this database were stored in a conventional relational database management system, the orders data would be stored in a separate ORDERS.T table, with each row in the table linked to a corresponding row in the INVENTORY.T table by an ITEM_CODE that matches the ITEM_CODE of the INVENTORY.T table. The SELECT would be more complex:

```
>SELECT DESCRIPTION, VENDOR_CODE, ORDER_QTY
SQL+FROM INVENTORY.T, ORDERS.T
SQL+WHERE ORDERS.T.ITEM_CODE = INVENTORY.T.ITEM_CODE
SQL+AND DESCRIPTION = 'Hot Dog Buns';
```

Selection Criteria and Multivalued Columns

You can select rows based on multivalued columns in much the same way as you use singlevalued columns, but multivalued columns offer a few more options. With multivalued columns, you can use several types of clauses to select the data you want to see. The WHERE clause also is used with singlevalued columns, but the remaining three clauses are used solely with multivalued columns:

Clause	Effect
WHERE Clause	Selects rows where *at least one* of the values in the multivalued column matches the selection criteria.
WHERE EVERY	Selects rows where *all* of the values in the multivalued column match the selection criteria.
WHEN Clause	Further determines which of the multivalues in the selected rows are to be actually *displayed* in the output.
UNNEST Clause	Explodes the multivalued association values so that each such value is combined with the other data in the row to form a complete and separate record.

Selection Criteria with Multivalued Columns

It is almost impossible to understand these different clauses and how they work in combination from just a line or two of explanation. The best way to learn about them is to observe their different effects on the output results, as shown in the following sections. You are encouraged to experiment with different combinations of these clauses to fine tune your output.

To show how these clauses work, the next several examples start with a simple request and build upon it. To list the animals that have *any* booster shots due before 1996, enter:

```
>SELECT NAME, VAC_TYPE, VAC_NEXT
SQL+FROM LIVESTOCK.T
SQL+WHERE VAC_NEXT <= '12/31/95';
NAME......    VAC_TYPE    VAC_NEXT..

Bussa         R           10/11/96
              P           04/25/95
              L           05/05/96
Warri         R           09/03/95
              P           05/05/96
              L           11/25/96
```

```
Ekiti          R            08/25/95
               P            08/04/95
               L            03/11/95
Gboko          R            01/03/95
.
.
.
Wurno          R            02/08/96
Press any key to continue...
```

Using just the WHERE clause, you get a listing of all the animals who have any booster shots scheduled before 1996. But all the multivalues in any selected row are treated as one entity, with no attempt to distinguish between those values that satisfy the WHERE clause and those that don't. For example, Bussa has three booster shots scheduled, one during 1995, and two in 1996. The output shows the data for all three boosters because you asked to see all rows that have at least one booster scheduled before 1996. If this is confusing, think of using WHERE alone as really asking for WHERE ANY.

WHERE may be what you want in some cases, but at other times you may want to see *only* those boosters that match your criteria. In the latter case, to extract a subset of the multivalued data selected, use a WHEN clause. List the animals that have any booster shots due before the end of 1995, and show the particulars for *only* those shots:

```
>SELECT NAME, VAC_TYPE, VAC_NEXT
SQL+FROM LIVESTOCK.T
SQL+WHERE VAC_NEXT <= '12/31/95'
SQL+WHEN VAC_NEXT <= '12/31/95';
NAME......     VAC_TYPE     VAC_NEXT..

Bussa          P            04/25/95
Warri          R            09/03/95
Ekiti          R            08/25/95
               P            08/04/95
               L            03/11/95
Gboko          R            01/03/95
.
.
.
Imese          P            12/03/95
               L            12/03/95
Press any key to continue...
```

The effect of this version is to first select the rows with at least one booster scheduled before 1996, and from those rows extract only the applicable data. This time, in Imese's case, you see only the two boosters that are scheduled before 1996, because the WHEN clause filters the multivalues selected by the WHERE clause.

You can achieve almost the same result by using an UNNEST clause to "explode" each set of multivalues into discrete rows, so that the WHERE clause can operate on each exploded row as if it were singlevalued. For example:

```
>SELECT NAME, VAC_TYPE, VAC_NEXT
SQL+FROM UNNEST LIVESTOCK.T ON VAC_ASSOC
SQL+WHERE VAC_NEXT <= '12/31/95';
NAME......    VAC_TYPE    VAC_NEXT..

Bussa         P           04/25/95
Warri         R           09/03/95
Ekiti         R           08/25/95
Ekiti         P           08/04/95
Ekiti         L           03/11/95
Gboko         R           01/03/95
Gboko         P           08/19/95
Gboko         L           12/17/95
Marone        R           01/22/95
Marone        P           05/08/95
Marone        L           08/31/95
.
.
.
Imese         P           12/03/95
Press any key to continue...
```

In this query, the UNNEST clause creates a virtual table containing a row for each multivalue and then the WHERE clause selects on those rows. Again, the output shows only those vaccination values that satisfy the date criterion. The difference between the output of this example and that of the previous one is that the data in columns not included in the association (NAME) is repeated on each line.

***Note**: You cannot use UNNEST and WHEN clauses that both refer to the same association because an UNNEST clause changes the multivalued columns in the association to singlevalued, and the WHEN clause does not operate on singlevalued columns.*

If you use a WHERE EVERY clause, there is no problem with seeing extraneous vaccination information, because the query selects only those rows where *all* booster shots satisfy the date criterion.

List those animals with *all three booster shots* scheduled before 1996:

```
>SELECT NAME, VAC_TYPE, VAC_NEXT
SQL+FROM LIVESTOCK.T
SQL+WHERE EVERY VAC_NEXT <= '12/31/95';
NAME......    VAC_TYPE    VAC_NEXT..

Ekiti         R           08/25/95
              P           08/04/95
```

```
                L           03/11/95
Gboko           R           01/03/95
                P           08/19/95
                L           12/17/95
Marone          R           01/22/95
                P           05/08/95
                L           08/31/95
Moundon         R           06/12/95
                P           03/27/95
                L           03/30/95
Namoda          R           09/17/95
.
.
.
Kribi           R           08/13/95
Press any key to continue...
```

Now the output lists only those animals that have all three booster shots scheduled before 1996.

The preceding examples give you some idea of the variety of outputs you can get when selecting on multivalued columns. The following sections explain each option in greater detail.

Using WHERE

In dealing with multivalued columns, you must understand the distinction between the function of the WHERE Clause and the function of the WHEN clause, which is a UniVerse SQL enhancement that is used exclusively with multivalued columns. Remember that WHERE *retrieves rows*, and WHEN *selectively displays multivalues* in those selected rows.

When you want to retrieve rows where one or more of the values in a multivalued column satisfies the criteria, use the WHERE clause, which operates somewhat like it did with singlevalued columns. As with singlevalued columns, you can use any of the other comparison operators or any keywords such as BETWEEN, IN, SAID, LIKE, IS NULL, and NOT.

List the employees who have any dependents born since 2/1/89:

```
>SELECT NAME, DEP_DOB, DEP_NAME FROM PERSONNEL.T
SQL+WHERE DEP_DOB > '2/1/89';
NAME.....................    DEP_DOB...    DEP_NAME..

Nelson, Lisa                 09/27/69      Robert
                             04/13/94      Brian
Niederberger, Brian          11/16/68      Darlene
```

```
                                06/30/94      Marion
Torres, Stephen                 10/26/76      Patricia
                                11/27/94      Cecilia
Osborne, Paul                   03/07/58      Evelyn
                                10/13/92      Russell
                                01/30/93      Harold
Perry, Patricia                 02/18/62      Brian
.
.
.
Morse, Carol                    09/06/67      Leonard
                                06/06/90      Jacqueline
Press any key to continue...
```

The next query does a table join using the multivalued column EQUIP_CODE in the PERSONNEL.T table and the singlevalued column EQUIP_CODE in the EQUIPMENT.T table to obtain a list of employees who have had experience running the hot dog stand:

```
>SELECT NAME FROM PERSONNEL.T, EQUIPMENT.T
SQL+WHERE PERSONNEL.T.EQUIP_CODE = EQUIPMENT.T.EQUIP_CODE
SQL+AND DESCRIPTION LIKE 'Hot Dog%';
NAME.....................

Morse, Leonard
King, Nathaniel
Ford, Hope
Milosz, Charles
Anderson, Suzanne

5 records listed.
```

In this case, `PERSONNEL.T.EQUIP_CODE = EQUIPMENT.T. EQUIP_CODE` is the join condition between PERSONNEL.T and EQUIPMENT.T, as expected. But no join condition is necessary between each employee in the PERSONNEL.T table and his or her list of equipment operating skills (the multivalued column, EQUIP_CODE) because it is part of a "table within a table" (the association EQUIP_ASSOC, which comprises the multivalued columns EQUIP_CODE and EQUIP_PAY).

Using EVERY

To select only those rows where *every* value in a multivalued column meets the selection criteria, you can add the keyword EVERY to the WHERE clause.

This example uses EVERY to list those inventory items for which all order quantities are 700 units or less. The output lists only those rows where *every* multivalue satisfies the selection criterion.

```
>SELECT ITEM_TYPE, DESCRIPTION, ORDER_QTY FROM INVENTORY.T
SQL+WHERE EVERY ORDER_QTY <= 700;
ITEM_TYPE    DESCRIPTION.............    ORDER_QTY

D            Mustard                           400
R            French Fries, Frozen              600
R            Cookies                           500
.
.
.
B            Film                              500
                                               200
                                               400
                                               600
Press any key to continue...
```

WHERE EVERY selects a row for which the association containing the multivalued column has no association rows (that is, the multivalued column has no values). When a set is empty, every one of its values meets any selection criteria, whatever they are.

Using WHEN

When you use just the WHERE clause, the output result contains all of the values in the multivalued column, even though you may be interested in only some of them.

In the example that asked for a list of animals who had any booster shots scheduled before 1996, the output listed the VAC_TYPE and VAC_NEXT for *all* of the animal's shots, even those that were not scheduled before 1996. To see only those shots scheduled during the period requested, add a WHEN clause:

```
>SELECT NAME, VAC_TYPE, VAC_NEXT
SQL+FROM LIVESTOCK.T
SQL+WHERE VAC_NEXT < '12/31/95'
SQL+WHEN VAC_NEXT < '12/31/95';
```

The WHEN Clause limits output from multivalued columns to only those values that meet specified criteria, without having to unnest the columns first.

As another example, ask for a list of all locations and the fees charged by their government agencies:

```
>SELECT DESCRIPTION FMT '30L', GOV_AGENCY, GOV_FEE
SQL+FROM LOCATIONS.T;
DESCRIPTION..................... GOV_AGENCY................ GOV_FEE.....

Detroit State Fair Ground     Health Inspector               3231.00
                              Sales Tax Authority            1504.00
                              Police, Paid Detail             615.00
                              Labor Inspector                4045.00
                              Alcohol Tobacco Firearms       1772.00
                              Weights And Measures           3274.00
                              Food & Ag (Animal Health)      3082.00
                              Fire Marshal                   4659.00
                              Environmental Permitting        761.00
                              Zoning Board                   2218.00
Houston State Fair Ground  Health Inspector                  3931.00
                              Sales Tax Authority            2673.00
                              Police, Paid Detail            3931.00
                              Labor Inspector                2756.00
                              Alcohol Tobacco Firearms       4582.00
                              Weights And Measures           2325.00
                              Food & Ag (Animal Health)      3523.00
                              Fire Marshal                   3109.00
                              Environmental Permitting       4596.00
                              Zoning Board                   2094.00
Milwaukee State Fair GroundHealth Inspector                 1295.00
Press any key to continue...
```

WHEN is most commonly used with a WHERE clause. Recall that WHERE affects retrieval, and WHEN affects output results.

The next example retrieves the rows from LOCATIONS.T for those towns where at least one government agency charges a fee of over $5,200:

```
>SELECT DESCRIPTION FMT '30L', GOV_AGENCY, GOV_FEE
SQL+FROM LOCATIONS.T
SQL+WHERE GOV_FEE > 5200;
DESCRIPTION.....................  GOV_AGENCY................ GOV_FEE.....

Milwaukee State Fair Ground    Health Inspector               1295.00
                               Sales Tax Authority            3217.00
                               Police, Paid Detail            2394.00
                               Labor Inspector                4262.00
                               Alcohol Tobacco Firearms       1916.00
                               Weights And Measures            793.00
                               Food & Ag (Animal Health)      5219.00
                               Fire Marshal                   1440.00
                               Environmental Permitting       3649.00
                               Zoning Board                    645.00
Minneapolis State Fair Ground  Health Inspector               5321.00
                               Sales Tax Authority            3433.00
                               Police, Paid Detail            2783.00
                               Labor Inspector                3603.00
                               Alcohol Tobacco Firearms       5455.00
                               Weights And Measures           1844.00
                               Food & Ag (Animal Health)       868.00
```

```
                              Fire Marshal                  1422.00
                              Environmental Permitting      1428.00
                              Zoning Board                  5019.00
Los Angeles State Fair Ground Health Inspector              2604.00
                              Sales Tax Authority           4681.00
                              Police, Paid Detail           4289.00
                              Labor Inspector               3175.00
                              Alcohol Tobacco Firearms      3684.00
                              Weights And Measures          1103.00
                              Food & Ag (Animal Health)     5473.00
                              Fire Marshal                  4194.00
Press any key to continue...
```

The result lists *all* the government fees charged at those locations. If you want the output to include *only* those fees over $5,200, add a WHEN clause that echoes the WHERE clause's selection criteria:

```
>SELECT DESCRIPTION FMT '30L', GOV_AGENCY, GOV_FEE
SQL+FROM LOCATIONS.T
SQL+WHERE GOV_FEE > 5200
SQL+WHEN GOV_FEE > 5200;
DESCRIPTION....................  GOV_AGENCY.............  GOV_FEE.....

Milwaukee State Fair Ground     Food & Ag (Animal Health)       5219.00
Minneapolis State Fair Ground   Health Inspector                5321.00
                                Alcohol Tobacco Firearms        5455.00
Los Angeles State Fair Ground   Food & Ag (Animal Health)       5473.00
Cleveland State Fair Ground     Police, Paid Detail             5360.00
                                Environmental Permitting        5240.00
Dallas State Fair Ground        Alcohol Tobacco Firearms        5288.00
New Orleans State Fair Ground   Sales Tax Authority             5277.00
                                Fire Marshal                    5207.00
Topsfield Fair                  Fire Department                 5430.00
Seattle State Fair Ground       Health Inspector                5235.00
Golden Gate Exposition Center   Labor Inspector                 5347.00
Atlanta State Fair Ground       Police, Paid Detail             5226.00
Chicago State Fair Ground       Police, Paid Detail             5326.00
                                Fire Marshal                    5449.00
Savannah State Fair Ground      Police, Paid Detail             5330.00
Reno State Fair Ground          Food & Ag (Animal Health)       5462.00

13 records selected.  17 values listed.
```

You can use the WHEN clause alone, without a corresponding WHERE clause, to suppress display of certain multivalues. To *list* all of the rows in the LOCATIONS.T table, but not *display* any multivalued agency fees that are below $5,200, enter:

```
>SELECT DESCRIPTION FMT '30L', GOV_AGENCY, GOV_FEE
SQL+FROM LOCATIONS.T
SQL+WHEN GOV_FEE >= 5200;
DESCRIPTION....................  GOV_AGENCY.............    GOV_FEE.....

Detroit State Fair Ground
Houston State Fair Ground
Milwaukee State Fair Ground     Food & Ag (Animal Health)         5219.00
Minneapolis State Fair Ground   Health Inspector                  5321.00
                                Alcohol Tobacco Firearms          5455.00
Springfield State Fair Ground
```

```
Washington State Fair Ground
Los Angeles State Fair Ground   Food & Ag (Animal Health)        5473.00
Cleveland State Fair Ground     Police, Paid Detail              5360.00
                                Environmental Permitting         5240.00
Dallas State Fair Ground        Alcohol Tobacco Firearms         5288.00
New Orleans State Fair Ground   Sales Tax Authority              5277.00
                                Fire Marshal                     5207.00



Chicago State Fair Ground       Police, Paid Detail              5326.00
                                Fire Marshal                     5449.00
Hartford State Fair Ground
Press any key to continue...
```

Using UNNEST

When retrieving data from multivalued columns, the output consists of one line for each selected row, with the values of any multivalued columns listed on successive lines:

```
>SELECT ANIMAL_ID, NAME, VAC_TYPE, VAC_DATE
SQL+FROM LIVESTOCK.T
SQL+WHERE NAME = 'Ekiti';
ANIMAL_ID    NAME......    VAC_TYPE    VAC_DATE..

       32    Ekiti         R           08/25/92
                           P           08/04/92
                           L           03/11/92

1 records listed.
```

Even though it may look like three rows of output, what you are actually seeing is a single output row displayed on three *lines*, which is confirmed by the `1 records listed.` message. Sometimes you want to treat each association row as a separate row.

There are several reasons to do this. One reason was mentioned earlier, when UNNEST Clause was used as an alternative to the WHERE/WHEN clause combination to produce output of only those multivalues that satisfied the selection criterion. Another reason is to include the values from the other (nonmultivalued) columns in the table on *every* line of the output.

The UNNEST clause “unnests” or explodes associated table rows containing multivalued data and produces a separate row for each multivalue. Unnesting is performed before anything else, and the unnested columns are treated as singlevalued columns for the remainder of the processing of the query.

The UNNEST clause names the table, and either an association name or the name of a multivalued column (column *aliases* cannot be used) within the association. Its syntax is as follows:

FROM UNNEST *tablename* **ON** { *association_name* | *columnname* }

Add an UNNEST clause to the previous query and see the difference in the output result:

```
>SELECT ANIMAL_ID, NAME, VAC_TYPE, VAC_DATE
SQL+FROM UNNEST LIVESTOCK.T ON VAC_ASSOC
SQL+WHERE NAME = 'Ekiti';
ANIMAL_ID    NAME......    VAC_TYPE    VAC_DATE..

       32    Ekiti         R           08/25/92
       32    Ekiti         P           08/04/92
       32    Ekiti         L           03/11/92

3 records listed.
```

Getting a totally separate and complete row for each multivalue in the association VAC_ASSOC, with the common single-column data (ANIMAL_ID and NAME) replicated on each line, provides several potential advantages:

- By repeating the data from the singlevalued columns on each line, the output may be rendered more readable in certain circumstances.
- By retrieving a virtual row for each value in a multivalued column, you can treat the retrieved rows as if they contained all singlevalued columns. An earlier example used UNNEST to cause a WHERE clause to select on multivalued columns as if they were singlevalued, without needing to add a WHEN clause:

```
>SELECT NAME, VAC_TYPE, VAC_NEXT
SQL+FROM UNNEST LIVESTOCK.T ON VAC_ASSOC
SQL+WHERE VAC_NEXT < '12/31/95';
```

The next query does a table join using the multivalued column EQUIP_CODE in the PERSONNEL.T table and the singlevalued column EQUIP_CODE in the EQUIPMENT.T table to obtain a list of the equipment that Daniel Hanson has used:

```
>SELECT NAME, PERSONNEL.T.EQUIP_CODE, DESCRIPTION
SQL+FROM UNNEST PERSONNEL.T ON EQUIP_ASSOC, EQUIPMENT.T
SQL+WHERE NAME LIKE 'Hanson, Daniel%'
SQL+AND PERSONNEL.T.EQUIP_CODE = EQUIPMENT.T.EQUIP_CODE;
NAME.....................    EQUIP_CODE
DESCRIPTION..............

Hanson, Daniel                        28    Truck 897 M X X
Hanson, Daniel                        39    Zoo Fencing
Hanson, Daniel                        42    Desk Credenza Sets
Hanson, Daniel                        26    Truck 243 Y G N

4 records listed.
```

Use the same join to determine which pieces of equipment each of the staff members has used:

```
>SELECT NAME, PERSONNEL.T.EQUIP_CODE, DESCRIPTION
SQL+FROM UNNEST PERSONNEL.T ON EQUIP_ASSOC, EQUIPMENT.T
SQL+WHERE PERSONNEL.T.EQUIP_CODE = EQUIPMENT.T.EQUIP_CODE;
NAME.....................    EQUIP_CODE
DESCRIPTION..............

Sunshine, Susie                       12    Beer Keg Stand
Irwin, Rebecca                         4    Lucky Dip Stand
Hanson, Daniel                        28    Truck 897 M X X
Hanson, Daniel                        39    Zoo Fencing
Hanson, Daniel                        42    Desk Credenza Sets
Hanson, Daniel                        26    Truck 243 Y G N
Vaughan, Randall                      50    Video Cameras
Vaughan, Randall                       1    Souvenir Stand
Nelson, Lisa                          48    Copier

Bailey, Cheryl                        42    Desk Credenza Sets
.
.
.
Kozlowski, Nicholas                   43    Feeding Buckets
Kozlowski, Nicholas                    1    Souvenir Stand
Press any key to continue...
```

The following example shows the use of a multivalued column in a subquery to get a list of those inventory items that had order quantities higher than 1.5 times the average of all order quantities:

```
>SELECT DESCRIPTION, ORDER_QTY
SQL+FROM UNNEST INVENTORY.T ON ORDERS_ASSOC
SQL+WHERE ORDER_QTY >
SQL+(SELECT (AVG(ORDER_QTY) * 1.5) FROM INVENTORY.T);
DESCRIPTION.............    ORDER_QTY

Crabcakes                          800
Crabcakes                          900
Jerky                              900
Handbills                          800
Handbills                          900
Horse Feed                         900
Ticket Stock                       900
.
.
.
Popcorn                            800
Press any key to continue...
```

Although all of the UNNEST examples have used the *association* name connected with the multivalued columns in its ON phrase, they could just as well have used the name of one of the multivalued columns within the association. But even if multivalued columns are unassociated, the “table within a table” principle still holds. As mentioned earlier, the VENDORS.T table contains a pure where-used list in the multivalued column EQUIP_CODE. You can treat EQUIP_CODE as though it were in an association.

List the IDs of the equipment that was purchased from Utopia Professionals:

```
>SELECT VENDORS.T.EQUIP_CODE
SQL+FROM UNNEST VENDORS.T ON EQUIP_CODE, EQUIPMENT.T
SQL+WHERE COMPANY LIKE 'Utopia Professionals%'
SQL+AND VENDORS.T.EQUIP_CODE = EQUIPMENT.T.EQUIP_CODE;
EQUIP_CODE

        49
        34
        27

3 records listed.
```

Using Set Functions

The set functions (AVG, COUNT, MAX, MIN, and SUM), described in Chapter 2, "Using SELECT Statements," are as applicable to multivalued columns as they are to singlevalued ones.

As an example, each row of the ENGAGEMENTS.T table represents a booking at a particular location (LOCATION_CODE) on a particular date (DATE). One association of multivalued columns in that table represents entrance gates, GATE_NUMBER represents the gate number (1 through 20), and GATE_TICKETS records the number of tickets sold at that gate. There is also a third column, GATE_REVENUE, which records the revenues for the gate.

1. To see the ticket sales by gate for the performance in East Atlanta on 3/18/94, enter:

```
>SELECT LOCATION_CODE, DATE, GATE_NUMBER, GATE_TICKETS
SQL+FROM ENGAGEMENTS.T
SQL+WHERE LOCATION_CODE LIKE 'EATL%' AND DATE = '03/18/94';
LOCATION_CODE    DATE......    GATE_NUMBER    GATE_TICKETS

EATL001          03/18/94                1             163
                                         2             615
                                         3             133
                                         4             774
                                         5             261
                                         6            1006
                                         7             479
                                         8             594
                                         9             504
                                        10             888
                                        11             419
                                        12            1192
                                        13             653
                                        14             677
                                        15             303
                                        16             471
                                        17             435
                                        18             305
                                        19             577
                                        20             115

1 records listed.
```

2. To calculate the average ticket sales per gate on that date, just replace the column specifications with the AVG function:

```
>SELECT AVG(GATE_TICKETS) CONV 'MD0'
SQL+FROM ENGAGEMENTS.T
SQL+WHERE LOCATION_CODE LIKE 'EATL%' AND DATE = '03/18/94';
AVG ( GATE_TICKETS )

                528

1 records listed.
```

3. Average ticket sales more selectively in two ways. First, ask for the average ticket sales for gates 1 through 5 on that day:

```
>SELECT AVG(GATE_TICKETS) CONV 'MD0'
SQL+FROM ENGAGEMENTS.T
SQL+WHERE LOCATION_CODE LIKE 'EATL%' AND DATE = '03/18/94'
SQL+WHEN GATE_NUMBER < 6;
AVG ( GATE_TICKETS )

                389

1 records listed.
```

   Ask for the average ticket sales for gate 3 for *all* performance dates in Atlanta:

```
>SELECT AVG(GATE_TICKETS) CONV 'MD0'
SQL+FROM ENGAGEMENTS.T
SQL+WHERE LOCATION_CODE LIKE 'EATL%'
SQL+AND GATE_TICKETS > 0 WHEN GATE_NUMBER = 3;
AVG ( GATE_TICKETS )

                306

1 records listed.
```

   Note that it was necessary to include the `AND GATE_TICKETS > 0` condition because some engagement dates are in the future and no tickets have been sold yet. Including such dates would produce a lower-than-actual average result (assuming that every gate sells some tickets at each engagement).

4. Ask for the average ticket sales by gate for all performance dates:

```
>SELECT GATE_NUMBER, AVG(GATE_TICKETS) CONV 'MD0'
SQL+FROM UNNEST ENGAGEMENTS.T ON GATES_ASSOC
SQL+WHERE LOCATION_CODE LIKE 'EATL%'
SQL+AND GATE_TICKETS > 0
SQL+GROUP BY GATE_NUMBER;
GATE_NUMBER    AVG ( GATE_TICKETS )

          1                     352
          2                     356
          3                     306
```

```
             4                    570
             5                    364
             6                    588
             7                    318
             8                    462
             9                    430
            10                    509
            11                    337
            .
            .
            .
            20                    220

20 records listed.
```

One reason for using UNNEST here is that GROUP BY can refer only to a singlevalued column. Including the UNNEST clause explodes the multivalues in GATE_TICKETS into discrete rows, making it appear to be a singlevalued column. If you try to do a GROUP BY on a multivalued column without unnesting it first, you see the error message `GROUP BY columns must be single valued`.

Subqueries on Nested Tables

Subqueries allow you to ask questions such as how sales on one date or for one gate compare to calculated averages. Suppose you want to know which gates at the 3/18/94 performance in East Atlanta sold more tickets than the average of all gates on that date. You must begin with the outer SELECT that lists the gates and their ticket counts, but you cannot just add a selection condition. For example:

```
>SELECT LOCATION_CODE, DATE, GATE_NUMBER, GATE_TICKETS
SQL+FROM ENGAGEMENTS.T
SQL+WHERE LOCATION_CODE LIKE 'EATL%' AND DATE = '03/18/94'
SQL+AND GATE_TICKETS > AVG(GATE_TICKETS);
UniVerse/SQL: Set functions may not be specified directly in the
WHERE clause
```

You cannot specify a set function directly in a WHERE clause, but even if you could, this phraseology would not be precise enough. Instead, perform the comparison against an inner SELECT whose result is the appropriate average:

```
>SELECT LOCATION_CODE, DATE, GATE_NUMBER, GATE_TICKETS
SQL+FROM ENGAGEMENTS.T
SQL+WHERE LOCATION_CODE LIKE 'EATL%' AND DATE = '03/18/94'
SQL+WHEN GATE_TICKETS >
SQL+   (SELECT AVG(GATE_TICKETS)
SQL+      FROM ENGAGEMENTS.T
SQL+      WHERE LOCATION_CODE LIKE 'EATL%' AND
SQL+      DATE = '03/18/94');
LOCATION_CODE    DATE......    GATE_NUMBER    GATE_TICKETS

EATL001          03/18/94                2             615
                                         4             774
                                         6            1006
                                         8             594
                                        10             888
                                        12            1192
                                        13             653
                                        14             677
                                        19             577

1 records selected.
```

For another example, suppose that you want to know the dates on which ticket sales for gate 12 were above the average of ticket sales *for that gate* on all dates in East Atlanta (remember to exclude from the average all dates for which tickets have not yet been sold):

```
>SELECT LOCATION_CODE, GATE_NUMBER, DATE, GATE_TICKETS
SQL+FROM UNNEST ENGAGEMENTS.T ON GATE_NUMBER
SQL+WHERE LOCATION_CODE LIKE 'EATL%'
SQL+AND GATE_NUMBER = 12
SQL+AND GATE_TICKETS >
SQL+   (SELECT AVG(GATE_TICKETS)
SQL+      FROM UNNEST ENGAGEMENTS.T ON GATE_NUMBER
SQL+      WHERE LOCATION_CODE LIKE 'EATL%'
SQL+      AND GATE_NUMBER = 12
SQL+      AND GATE_TICKETS > 0);
LOCATION_CODE    GATE_NUMBER    DATE......    GATE_TICKETS

EATL001                  12    03/18/94              1192
EATL001                  12    05/24/94               514

2 records listed.
```

You might use WHEN GATE_NUMBER = 12 in the subquery, but because you cannot use WHEN in a subquery, the alternative is to unnest the association rows.

You can perform other combinations of set functions in a similar fashion. The UNNEST operation is not necessary in a statement such as SELECT AVG (GATE_TICKETS); however, UNNEST is essential in the outer SELECT.

Using Dynamic Normalization

Dynamic normalization is a UniVerse extension that explodes multivalued columns (associated or unassociated) so that they appear as singlevalued. In other words, dynamic normalization allows you to process a nonfirst-normal-form (NF^2) table as if it were a first-normal-form (1NF) table and, in effect, performs an UNNEST on an association or multivalued column within that table. A major use of dynamic normalization is to enable client (such as UniVerse Call Interface, or UCI) applications to treat NF^2 tables and associations as if they were 1NF structures, although you can also use dynamic normalization in interactive SQL queries and in UniVerse BASIC programs using the UniVerse BASIC SQL Client Interface.

When you want to dynamically normalize a table, use the construct *tablename_association* or *tablename_mvcolname* instead of *tablename*, where *mvcolname* is an unassociated multivalued column. The result is a virtual table, containing the table's primary key (or @ID column if the table has no primary key) plus the columns of the association or the unassociated column, with an individual row generated for each value entry in the association or column. Dynamic normalization is similar to UNNEST in that the virtual 1NF table contains only singlevalued data and you use WHERE instead of WHEN to select from the table.

For example, this query references a virtual table consisting of ANIMAL_ID (the primary key of the LIVESTOCK.T table) and the four columns of the VAC_ASSOC association, exploded. To use a WHERE clause to show all booster shots due during September 1995, enter:

```
>SELECT * FROM LIVESTOCK.T_VAC_ASSOC
SQL+WHERE VAC_NEXT BETWEEN '9/1/95' AND '9/30/95'
SQL+ORDER BY ANIMAL_ID;
ANIMAL_ID    VAC_TYPE  VAC_DATE  VAC_NEXT  VAC_CERT

        1    p         09/08/92  09/08/95  498062
        4    L         09/02/92  09/02/95  814600
        4    R         09/08/92  09/08/95  242744
        9    R         09/17/92  09/17/95  716025
       24    R         09/03/92  09/03/95  745200
       26    L         09/28/92  09/28/95  391399
       70    R         09/02/92  09/02/95  821955
       71    R         09/07/92  09/07/95  12329

8 records listed.
```

You also can dynamically normalize unconverted UniVerse files and SQL tables. Use the construct *filename_association* or *filename_mvcolname*.

When referring to an association or unassociated multivalued column through dynamic normalization, you can reference a virtual column called @ASSOC_ROW whose value is a numeric position within a multiset. For example, to see an order number and the first item from each order, enter:

```
>SELECT ORDNO. ITEM FROM ORDERS_DETAIL WHERE @ASSOC_ROW = 1;
```

Dynamic normalization is useful when adding values to or modifying values in multivalued columns in an existing base table row.

Chapter

5

Modifying Data

This chapter explains how to use the UniVerse SQL statements INSERT, UPDATE, and DELETE to add, update, and delete data in tables and files.

You also can modify tables and files in any of the following ways:

- Using the UniVerse ReVise process
- Writing a UniVerse BASIC program
- Using the UniVerse Editor
- Using client software such as interCALL and UniVerse ODBC

Before performing any of these operations on a table, you must have permission to do so. This raises the issue of *database security.* Additionally, be aware that *data integrity* imposes certain restrictions on the values you can enter into certain columns.

Database Security and UniVerse SQL

Before you can change a table or file, you must have been granted the appropriate privileges. You should be aware that *data integrity* imposes certain restrictions on the values you legitimately can enter into certain columns. How to grant UniVerse SQL database privileges is covered in detail in *UniVerse SQL Administration for DBAs*. This section provides a basic understanding of database security.

There are three layers to database security:

- An operating system layer
- A UniVerse layer
- A UniVerse SQL layer

Operating System Security

File permissions on a UniVerse user's files and directories are set when the user is added to the system and when the user's UniVerse account is created. Default file permissions are set by the *umask* environment variable in the user's *.profile* file or in a UniVerse account's LOGIN paragraph. These default file permissions determine permissions for all files and directories subsequently created by the user.

Use UniAdmin or the UniVerse System Administration menus to set and change permissions for a UniVerse account. Files can be protected in a number of ways. Permission must be granted to the owner, to the owner's group, and/or to others outside the group before a file can be written to, read from, and/or executed. Either the system administrator or the owner of a file can change these permissions at will. All such security is based on password protection, which prevents unauthorized users from logging on to an account and gaining access to its protected files.

More detailed information about operating system security is in the documentation for your operating system.

UniVerse Security

In addition to operating system security, another way of controlling user actions is to edit the contents of the VOC file for an account, then setting file permissions to prevent users from changing the VOC file. Because a VOC file contains all of the commands and verbs that a user can execute, removing certain entries from the file prevents users from executing them.

Also, VOC entries that point to remote items provide a further mechanism for controlling access to certain commands. By specifying a user-supplied subroutine in field 4 of remote-type VOC entries, you can set a flag that permits or restricts access to the remote item.

UniVerse SQL Security

With UniVerse SQL you have a third layer of security beyond operating system security and basic UniVerse security. The UniVerse SQL GRANT statement assigns database privileges and user privileges on tables and views.

Only a DBA (database administrator) can grant database privileges to users. The three levels of database privileges, from lowest level to highest, are as follows:

- CONNECT registers a user as a UniVerse SQL user. Granting CONNECT privilege to users allows them to create new tables and alter, delete, and grant and revoke privileges on tables they own.
- RESOURCE includes all capabilities of the CONNECT privilege, plus the power to create schemas. RESOURCE privilege can be granted only to those users who already have CONNECT privilege.

- DBA includes all capabilities of the RESOURCE privilege, plus the ability to create schemas and tables for other users, grant privileges on any table to any user, revoke privileges on any table from any user, and the ability to perform SELECT, INSERT, UPDATE, and DELETE operations on any table.

  Whenever you create a table, you are the only user with privileges on it, except for users with DBA privilege. You then can grant any or all of the following table privileges to others:

  - SELECT Privilege
  - INSERT Privilege
  - UPDATE Privilege
  - DELETE Privilege
  - REFERENCES Privilege
  - ALTER Privilege

Whether you can modify data in a specific table depends on whether you have operating system permissions for the table, UniVerse access to the needed commands and verbs, and the necessary UniVerse SQL table privileges.

Data Integrity

Completeness, consistency, and accuracy are extremely important in any application. As discussed in *UniVerse SQL Administration for DBAs*, UniVerse SQL helps ensure all three standards by providing a number of data integrity constraints that are or can be imposed on database INSERT and UPDATE operations.

Consequently, some values that you might attempt to enter or change will not be accepted, because they violate certain rules. To give you an idea of how UniVerse SQL maintains data integrity, here are a few things you are not allowed to do in the Circus database:

- Enter a null value as a LOCATION_CODE, VENDOR_CODE, ITEM_CODE, BADGE_NO, or any other record ID column. This is a *Required Data Violation* because null values are not allowed in record ID or primary key columns.
- Enter a nonunique value into any of these columns. This is an *Entity Integrity Violation* because all values in record ID or primary key columns must be unique.
- Enter a VENDOR_CODE value in the EQUIPMENT.T table that doesn't match a value in the VENDOR_CODE column of the VENDORS.T table. This is a *Referential Integrity Violation* because VENDOR_CODE in the EQUIPMENT.T table is defined as referencing the VENDOR_CODE in the VENDORS.T table.

Data integrity is further enforced by the CHECK column constraint, which specifies that a value to be inserted into a column must meet certain criteria, and by the CONV format specification, which requires that data entered for a column be convertible.

Transaction Processing

To be able to use the individual UniVerse SQL statements for modifying data, you need some understanding of transaction processing.

Transaction processing ensures database integrity by guaranteeing that unless a sequence of commands is completed successfully its effects are cancelled. Four transaction statements in UniVerse BASIC (BEGIN TRANSACTION, COMMIT, ROLLBACK, and END TRANSACTION) provide the ability to define and control transactions.

As a simple example, assume that you have a UniVerse BASIC program that removes an act from your roster and, consequently, updates any references to that act in the ENGAGEMENTS.T table. If you are not using transaction processing and you remove the act from the ACTS.T table just when the system crashes but before the program has had a chance to remove all references to the act in ENGAGEMENTS.T, the database would be out of synch (because ENGAGEMENTS.T is referring to an act that no longer exists).

With transaction management, the two operations—removing the act from ACTS.T and removing any references to the act from ENGAGEMENTS.T—are defined as a unified transaction sequence. If the execution sequence is not completed, as happened here, the database is “rolled back” (much like reversing a film or videotape) to the beginning of the transaction, restoring both tables to their state before the transaction began. However, if the entire sequence had executed successfully, a COMMIT would have been executed instead, recording the changes permanently in the database.

An interactive UniVerse SQL session is always considered to be in transaction mode, even though you never specifically define a transaction. Each transaction you enter defaults to an autocommit mode (that is, its results are *always* recorded in the database). In effect, each UniVerse SQL interactive statement constitutes a transaction by itself.

The remainder of this chapter discusses each UniVerse SQL data modification statement (INSERT, UPDATE, and DELETE), with special emphasis on handling multivalued columns.

Avoiding Lock Delays (NOWAIT)

Normally when an INSERT, UPDATE, or DELETE statement tries to access a row or table locked by another user or process, it waits for the lock to be released, then continues processing. Use the NOWAIT keyword to stop processing when a statement encounters a record or file lock. If the statement is used in a transaction, processing stops and the transaction is rolled back. The user ID of the user who owns the lock is returned to the terminal screen or to the client program.

Inserting Data (INSERT)

Databases are a reflection of reality. As such, adding a new employee, stock item, ride, vendor, or animal will require adding one or more rows of data somewhere in the database. In the context of the Circus database, this could be anything like booking a new engagement date, purchasing a new animal or concession, hiring a new staff member, or signing up a new vendor.

INSERT is the statement you use for adding *new* rows to a table (UPDATE is for changing values in *existing* rows). In its most basic form, INSERT names the table where the row is to be inserted and specifies the columns to be filled and the values to be inserted in those columns.

Before you can add a row to a table, you must know the names of the columns in the table and their format. If you do not have a copy of the table layout, one way to obtain this is to display the dictionary entries for the table. If you just purchased a new truck and want to add a new row to the EQUIPMENT.T table, retrieve the dictionary for EQUIPMENT.T:

```
>SELECT * FROM DICT EQUIPMENT.T;
Type &
Field......... Field. Field........ Conversion.. Column..... Output Depth &
Name.......... Number Definition... Code........ Heading.... Format Assoc..

EQUIP_CODE     D    0               MD0                       5R     S
@ID            D    0                          EQUIPMENT      5R     S
@KEY           PH     EQUIP_CODE

VENDOR_CODE    D    1               MD0                       5R     S
VENDOR_REF     D    2                                         10L    S
DESCRIPTION    D    4                                         25T    S
DEPRECIATION   D    3                                         1L     S
TAX_LIFE       D    7               MD0                       5R     S
VOLTS          D    8               MD0                       5R     S
COST           D    5               MD22                      12R    S
USE_LIFE       D    6               MD0                       5R     S
PURCHASE_DATE  D    9               D2/                       10L    S
@REVISE        PH     VENDOR_CODE
                      VENDOR_REF
                      DEPRECIATION
                      DESCRIPTION
                      COST USE_LIFE
                      TAX_LIFE
                      VOLTS
Press any key to continue...
```

This display shows that the EQUIPMENT.T table contains 10 singlevalued columns. The specs on the new truck include:

Make and Model	1992 Mack Truck Model 4500L
Purchased From	Century Group (Vendor #90)
Cost	$16,725.00
Date Purchased	December 15, 1994

Naming the Table and Specifying the Columns

To construct the INSERT statement, first name the table:

```
>INSERT INTO EQUIPMENT.T...
```

You then specify a *column list*, naming the columns into which you want to insert values. For example, if you want to fill in just the record ID, description, vendor source, cost, and date of purchase, enter the following column list:

```
>INSERT INTO EQUIPMENT.T
SQL+(EQUIP_CODE, DESCRIPTION, VENDOR_CODE,
SQL+COST, PURCHASE_DATE) VALUES (...
```

Sometimes you can eliminate the column list. In the case of an SQL table, if you omit the column list, you must supply a value for *every* column in the table, in the order in which the columns were defined in the original CREATE TABLE statement or in a later ALTER TABLE statement. So, if you want to insert a full row of data in the EQUIPMENT.T table, you would omit the list of columns and enter only the values:

```
>INSERT INTO EQUIPMENT.T VALUES (...
```

In the case of a UniVerse *file*, however, if you are inserting a row and do not include a column list, there must be an @INSERT phrase in the file dictionary. This @INSERT will define the columns to be filled and their order.

Supplying the Values

Finally, supply the values to go in those columns in the form of a *value list*:

```
>INSERT INTO EQUIPMENT.T
SQL+(EQUIP_CODE, DESCRIPTION, VENDOR_CODE,
SQL+COST, PURCHASE_DATE)
SQL+VALUES (61, '1992 Mack Truck Model 4500L', 90,
SQL+16725.00, '12/15/94');
UniVerse/SQL: 1 record inserted.
```

Think of the column list and value list as being paired, with each column name in the column list matched in a one-to-one correspondence to its value in the value list.

Remember that database conventions require that the record ID (in this case, EQUIP_CODE) is set to a unique value. In this table, record IDs are assigned sequentially, and 60 was the last number used. Therefore, assign 61 as the record ID of the new row.

All values must be listed in the same order as the column list and must conform to any format conventions or constraints that apply to their corresponding columns.

To double-check the new entry, enter:

```
>SELECT * FROM EQUIPMENT.T
SQL+WHERE EQUIP_CODE = 61;

EQUIP_CODE....    61
VENDOR_CODE...    90
VENDOR_REF....
DEPRECIATION..
DESCRIPTION... 1992 Mack Truck Model
               4500L
COST..........     16725.00
USE_LIFE......
TAX_LIFE......
VOLTS.........
PURCHASE_DATE. 12/15/94

1 records listed.
```

Any column not included in the column list will be set to a default value, if one was supplied in the table definition; otherwise, it will be set to null (or to an empty string if it is a UniVerse file). You can specify a null value for any column, particularly if you want to override its default value.

For example, if you wanted to enter the basic information for a new vendor but there is no third line of address (ADR3) and you do not have a contact name (CONTACT) as yet, your INSERT statement would omit the names of these two columns and also omit their respective values:

```
>INSERT INTO VENDORS.T
SQL+(VENDOR_CODE, COMPANY, ADR1, ADR2, TERMS, PHONE, FAX,
SQL+EQUIP_CODE, ITEM_CODE)
SQL+VALUES (233, 'New Age Plastics', '7300 Huntington Avenue',
SQL+'Boston MA 02116', 'Net 60', '617-555-3243',
SQL+'617-555-3246', 16, 44);
UniVerse/SQL: 1 record inserted.
```

Alternatively, you could enter the INSERT statement by retaining the names of the two columns and specifying NULL as their values:

```
>INSERT INTO VENDORS.T (VENDOR_CODE, COMPANY, ADR1, ADR2,
SQL+ADR3, TERMS, CONTACT, PHONE, FAX, EQUIP_CODE, ITEM_CODE)
SQL+VALUES (233, 'New Age Plastics', '7300 Huntington Avenue',
SQL+'Boston MA 02116', NULL, 'Net 60', NULL, '617-555-3243',
SQL+'617-555-3246', 16, 44);
UniVerse/SQL: 1 record inserted.
```

Using Expressions in Value Lists

Instead of an explicit value, you can use an expression in a value list. Although there is no particular reason for doing this in the Circus database, it is useful in other situations. In an application such as a retail operation, you might want to record both the retail price and the discounted price of each item. Use an INSERT such as:

```
>INSERT IN SALES.T VALUES (..., 10.50, 10.50 * 0.75,...);
```

Inserting Multivalues into a New Row

When specifying values to be inserted into multivalued columns when adding a new row, the only clause affected is the *values list*. Separate the values by commas and enclose the values for each multivalued column in angle brackets.

Suppose you placed four orders for a new stock item, corn dogs, and you want to reflect this in the INVENTORY.T table. Specifications for the new stock item are:

Column	Values
Type	R
Description	Corn Dogs
Quantity on Hand	825
Cost	$50.95
Price	$78.00
Vendors	79, 52, 95, 67
Order Quantities	150, 200, 350, 125

These specifications translate into the statement:

```
>INSERT INTO INVENTORY.T (ITEM_CODE, ITEM_TYPE, DESCRIPTION,
SQL+QOH, COST, PRICE, VENDOR_CODE, ORDER_QTY)
SQL+VALUES (46, 'R', 'Corn Dogs', 825, 50.95, 78.00,
SQL+<79, 52, 95, 67>, <150, 200, 350, 125>);
UniVerse/SQL: 1 record inserted.
```

As was the case with EQUIPMENT.T, the record ID of INVENTORY.T is also a sequentially assigned number (the last number assigned was 45). Now retrieve this row to make sure it has been stored properly:

```
>SELECT * FROM INVENTORY.T WHERE ITEM_CODE = 46;
ITEM_CODE...    46
ITEM_TYPE... R
DESCRIPTION. Corn Dogs
QOH.........   825
COST........        50.95
PRICE.......        78.00
VENDOR_CODE ORDER_QTY
         79       150
         52       200
         95       350
         67       125

1 records listed.
```

Inserting Multivalues into an Existing Row

Sometimes you want to add values to an association within an already existing row, such as adding a new vaccination entry for one of the animals.

You might assume that making a change to an *existing* row would be handled as an update and use an UPDATE statement. But because a table association is a "table within a table," use an INSERT statement instead and use dynamic normalization so that you can operate on the association as if it were a 1NF table.

For example, to add a new vaccination entry, you would apply dynamic normalization to the vaccination association (VAC_ASSOC). Take animal 73, which has records of three vaccinations:

```
>SELECT ANIMAL_ID, VAC_TYPE, VAC_DATE, VAC_NEXT, VAC_CERT
SQL+FROM LIVESTOCK.T WHERE ANIMAL_ID = 73;
ANIMAL_ID    VAC_TYPE    VAC_DATE..    VAC_NEXT..    VAC_CERT

       73    R           06/12/92      06/12/95      80782
             P           03/27/92      03/27/95      252906
             L           03/30/92      03/30/95      469618

1 records listed.
```

To add a new vaccination entry for vaccination type D, enter an INSERT statement that uses the LIVESTOCK.T_VAC_ASSOC table association. The virtual table consists of the primary key of the table (ANIMAL_ID) plus the four fields defined in VAC_ASSOC. Because the INSERT fills in all columns in this virtual table, there is no need to list the column names in the statement:

```
>INSERT INTO LIVESTOCK.T_VAC_ASSOC
SQL+VALUES (73, 'D', '08/03/95', '08/03/97', '800971');
UniVerse/SQL: 1 record inserted.
```

Now if you look at the row again, you see that the new vaccination entry has been added:

```
>SELECT VAC_TYPE, VAC_DATE, VAC_NEXT, VAC_CERT
SQL+FROM LIVESTOCK.T WHERE ANIMAL_ID = 73;
ANIMAL_ID    VAC_TYPE    VAC_DATE..    VAC_NEXT..    VAC_CERT

       73    R           06/12/92      06/12/95      80782
             P           03/27/92      03/27/95      252906
             L           03/30/92      03/30/95      469618
             D           08/03/95      08/03/97      800971

1 records listed.
```

The new row was added to the end of existing association rows, but such order is not guaranteed. However, an option of the CREATE TABLE statement allows some measure of control over where new rows are positioned in an association. The choices are LAST, which is the default, FIRST, and in order by the values in a specified column of the association.

Inserting Multiple Rows

Rather than add a single row of data to a table using a single-row INSERT statement, you can add multiple rows of data using a variation of the INSERT statement. A multirow INSERT is a variation of the INSERT statement that takes its values from the database, rather than from the INSERT statement. Since the data values to be inserted are taken from tables rather than from the INSERT statement itself, the form of the value list is a SELECT statement.

One use of a multirow INSERT is to copy selected rows and columns from one table to another, perhaps for the purpose of archiving old data. Suppose you create an abbreviated ENGAGEMENTS.T table, calling it OLD_ENGAGEMENTS.T, and use it to archive past bookings. Then, at the end of each year, you copy that year's engagements into the table. You can find the CREATE TABLE statement for generating this table in Appendix A, "The Sample Database," (note that, unlike the original table, the new table has no associations defined because they would impose unwanted constraints on the data). The statement for copying all 1994 dates from ENGAGEMENTS.T to OLD_ ENGAGEMENTS.T is:

```
>INSERT INTO OLD_ENGAGEMENTS.T (LOCATION_CODE, DATE,
SQL+GATE_REVENUE, RIDE_REVENUE, CONC_REVENUE)
SQL+SELECT LOCATION_CODE, DATE, GATE_REVENUE,
SQL+RIDE_REVENUE, CONC_REVENUE
SQL+FROM ENGAGEMENTS.T
SQL+WHERE DATE BETWEEN '01/01/94' AND '12/31/94';
UniVerse/SQL: 14 records inserted.
```

Then use a DELETE statement (see "Deleting Data (DELETE)" on page 25) to remove those rows from the ENGAGEMENTS.T table.

Other uses for multirow insertions are as follows:

- Combining data from two or more tables into a single table. One reason you might do this is to perform complex analyses on a large amount of data that is scattered among several tables. Among the advantages of doing so are the elimination of extraneous data and multitable joins (thereby speeding up retrieval), "freezing" the data at a particular point in time, and performing the analysis without affecting the production database.
- Using the SAMPLE keyword to create a test database.
- Doing statistical joins that require intermediate results.

For example, you might want to combine booking data from ENGAGEMENTS.T, ACTS.T, and PERSONNEL.T for the first quarter of 1995. Then analyze what acts were used for which dates, who comprised the staff, and how much they were paid. To load the table using an INSERT statement, enter:

```
>INSERT INTO NEWTAB.T (ENG_ID, ENG_DATE, ACT_DESC, EMP_ID,
SQL+EMP_NAME, ACT_PAY)
SQL+SELECT LOCATION_CODE, DATE, DESCRIPTION,
SQL+BADGE_NO, NAME, ACT_PAY
SQL+FROM ENGAGEMENTS.T, ACTS.T, PERSONNEL.T
SQL+WHERE PERSONNEL.T.ACT_NO = ACTS.T.ACT_NO
SQL+AND OPERATOR = BADGE_NO
SQL+AND DATE BETWEEN '1/1/95' AND '3/31/95';
```

Now you have a combined table that contains just the data with which you want to work. You also can query and refine without disturbing your production data.

A subquery in an INSERT statement cannot include field modifiers (AVERAGE, BREAK ON, BREAK SUPPRESS, CALCULATE, PERCENT, and TOTAL), field qualifiers (CONVERSION, FORMAT, and so forth), report qualifiers, processing qualifiers (except SAMPLE and SAMPLED), or the ORDER BY clause.

Updating Data (UPDATE)

UPDATE modifies the values of one or more columns in one or more selected rows of a table. The UPDATE statement specifies the table to be updated, the columns to be modified, and the rows to be selected.

Updating Values in a Single Row

In its simplest form, UPDATE modifies a single column value in a specific row. For example, to increase the depreciable life for equipment item 28 from 3 to 5 years, enter:

```
>UPDATE EQUIPMENT.T
SQL+SET TAX_LIFE = 5
SQL+WHERE EQUIP_CODE = 28;
UniVerse/SQL: 1 record updated.
```

In addition to setting a column to a literal value or expression, you also can do any of the following:

- SET *column* = NULL
- SET *column* = USER
- SET *column* = DEFAULT
- SET *column* = CURRENT_DATE
- SET *column* = CURRENT_TIME

Perhaps you want to update more than one column of the row. To update the quantity on hand, cost, and price for inventory item 13, enter:

```
>UPDATE INVENTORY.T
SQL+SET QOH = 65, COST = 38.94, PRICE = 50.76
SQL+WHERE ITEM_CODE = 13;
UniVerse/SQL: 1 record updated.
```

Updating Values in Multivalued Columns

You can update values in a multivalued column of a row by specifying a multivalued literal, much as you do when you *insert* values into a multivalued column. When using UPDATE on a multivalued column, supply the same number of values as are currently in the association's key column (the number of values in the association key of an association is called the depth of association).

For example, you might move the dates of the next scheduled vaccinations ahead three months for animal 74. First find out the depth of the VAC_ASSOC association—how many entries exist in the VAC_TYPE column (the association key)—by entering:

```
>SELECT * FROM LIVESTOCK.T WHERE ANIMAL_ID = 74;

ANIMAL_ID...    74
NAME........ Doba
DESCRIPTION. Hyena
USE......... Z
DOB......... 08/21/84
ORIGIN...... Kenya
COST........     10229.00
EST_LIFE....  15
VAC_TYPE VAC_DATE.. VAC_NEXT.. VAC_CERT
R        06/18/93   09/17/96   957640
P        05/24/92   05/24/95   573198
L        06/08/92   06/08/95   772270

1 records listed.
```

Since there are three vaccination entries for this row, include exactly three values in the update:

```
>UPDATE LIVESTOCK.T
SQL+SET VAC_NEXT = <'12/17/96', '08/24/95', '09/08/95'>
SQL+WHERE ANIMAL_ID = 74;
UniVerse/SQL: 1 record updated.
```

An update of this type replaces *all* the values in the column for the selected row. To update just one of the multivalues, add a WHEN clause to identify the value to be changed, as explained under "Using WHEN with UPDATE" on page 20.

Alternatively, update each date individually using dynamic normalization and issuing three UPDATE statements, one for each of the date values to be changed:

```
>UPDATE LIVESTOCK.T_VAC_ASSOC
SQL+SET VAC_NEXT = '12/17/96'
SQL+WHERE ANIMAL_ID = 74 AND VAC_NEXT = '09/17/96';
UniVerse/SQL: 1 record updated.

>UPDATE LIVESTOCK.T_VAC_ASSOC
SQL+SET VAC_NEXT = '08/24/95'
SQL+WHERE ANIMAL_ID = 74 AND VAC_NEXT = '05/24/95';
UniVerse/SQL: 1 record updated.

>UPDATE LIVESTOCK.T_VAC_ASSOC
SQL+SET VAC_NEXT = '09/08/95'
SQL+WHERE ANIMAL_ID = 74 AND VAC_NEXT = '06/08/95';
UniVerse/SQL: 1 record updated.
```

Using WHEN with UPDATE

Be very careful using WHERE with multivalued columns when updating a table. For example, take the statement:

```
>UPDATE ACTS.T SET EQUIP_CODE = 33
SQL+WHERE ACT_NO = 2 AND EQUIP_CODE = 32;
UniVerse/SQL: 1 record updated.
```

You might think that you are changing only those EQUIP_CODE values equal to 32 for act 2. Based on how the SELECT statement works, remember that by using just a WHERE clause, you retrieve *all* of the multivalues in the EQUIP_CODE column for act 2 (even though only one of them is equal to 32). You added a WHEN clause to see only the values equal to 32.

Similarly, what you are doing here is changing *all* of the EQUIP_CODE values for act 2 (assuming that at least one of them is equal to 32). To change *only* those values equal to 32, enter:

```
>UPDATE ACTS.T SET EQUIP_CODE = 33
SQL+WHERE ACT_NO = 2 WHEN EQUIP_CODE = 32;
UniVerse/SQL: 1 record updated.
```

Updating Globally

Most of the time, an UPDATE affects more than one row of a table. For instance, if you do not specify a WHERE clause in an UPDATE statement, the update applies to *all* rows of the table, in effect doing a bulk update of the table—something you will rarely want to do. For example, to change the payment terms in *all* rows of VENDORS.T to net 30 days, use the statement:

```
>UPDATE VENDORS.T SET TERMS = 'Net 30 Days';
UniVerse/SQL: 232 records updated.
```

Sometimes, you *do* want to change several selected rows, which you can accomplish by doing a global search-and-change. Expanding the example under "Updating Values in a Single Row" on page 18, increase the depreciable tax life of *any* equipment purchased since January 1993 and having a tax life of three years:

```
>UPDATE EQUIPMENT.T SET TAX_LIFE = 5
SQL+WHERE PURCHASE_DATE > '1/1/93' AND TAX_LIFE = 3;
UniVerse/SQL: 2 records updated.
```

Use any form of the WHERE clause that is valid in a SELECT statement:

```
>UPDATE EQUIPMENT.T SET TAX_LIFE = 5
SQL+WHERE TAX_LIFE IS NULL;
UniVerse/SQL: 1 record updated.

>UPDATE EQUIPMENT.T SET TAX_LIFE = 5, USE_LIFE = 10
SQL+WHERE EQUIP_CODE IN (23, 29, 34, 41);
UniVerse/SQL: 4 records updated.
```

Add the keyword REPORTING to your UPDATE statement to get a list of the primary keys of those rows affected:

```
>UPDATE EQUIPMENT.T SET TAX_LIFE = 5
SQL+WHERE TAX_LIFE IS NULL REPORTING;
UniVerse/SQL: Record "37" updated.
UniVerse/SQL: 1 record updated.
```

Using an Expression as the SET Value

You can use an *expression* as part of the SET clause. For example, to unload the overstock and make room for incoming merchandise, you decide to give a 20% discount on all inventory items, of which you have more than 600. Issue the statement:

```
>UPDATE INVENTORY.T SET PRICE = PRICE * .8
SQL+WHERE QOH > 600;
UniVerse/SQL: 1 record updated.
```

Using Subqueries in the WHERE Clause

It is not uncommon to find a subquery in the WHERE clause of an UPDATE statement. In fact, a subquery can be useful in determining which rows to update in a table, based on data contained in one or more other tables.

As one example, suppose that an employee, Hope Saarinen, has quit. To remove her name from any ride assignments, enter:

```
>UPDATE RIDES.T SET OPERATOR = 0,
SQL+WHERE OPERATOR = (SELECT BADGE_NO FROM PERSONNEL.T
SQL+WHERE NAME LIKE '%Saarinen, Hope%');
UniVerse/SQL: 2 records updated.
```

You have searched the PERSONNEL.T table to find Hope Saarinen, found her ID, and then removed the ID from any rows in the RIDES.T table that match her ID.

As another example, all vendors with whom you have placed more than three orders have offered to stretch their payment terms from the current net 30 days to a more lenient net 60 days. To update the VENDORS.T table to reflect this new development, enter:

```
>UPDATE VENDORS.T SET TERMS = 'Net 60 Days'
SQL+WHERE 3 < (SELECT COUNT(*) FROM INVENTORY.T
SQL+WHERE VENDORS.T.VENDOR_CODE = INVENTORY.T.VENDOR_CODE);
UniVerse/SQL: 4 records updated.
```

This is an example of a correlated subquery, as explained under "Correlated and Uncorrelated Subqueries" on page 23. VENDOR_CODE in the subquery is an outer reference, referring to VENDOR_CODE in the VENDORS.T table that is being updated.

Selecting Records for Updating

The FOR UPDATE clause locks all selected rows with update record locks (READU) or exclusive file locks (FX) until the end of the current transaction. This lets client programs update or delete the selected rows later within the same transaction, without being delayed by locks held by other users. You can also use the FOR UPDATE clause in an interactive SELECT statement.

The syntax of the FOR UPDATE clause is:

```
SELECT [ALL | DISTINCT] column_specifications
    FROM table_specification
    [WHERE clause]
    FOR UPDATE [OF column[,column]...]
```

The OF clause limits the acquiring of update record locks or file locks to those tables or files containing the named columns. It is useful only in a join where data is selected from two or more tables.

You cannot use the FOR UPDATE clause in:

- A subquery
- A view definition
- A trigger program

You cannot use the FOR UPDATE clause if the SELECT statement includes:

- The UNION operator
- Set functions
- A GROUP BY clause
- A HAVING clause

The current isolation level determines which locks are set when a SELECT statement includes the FOR UPDATE clause.

A file lock is set instead of record locks when a table or file already has the maximum number of record locks allowed by your system. The MAXRLOCK configurable parameter determines the maximum number of record locks.

***Note**: The FOR UPDATE clause has no effect on locks set by a subquery. Rows, tables, and files selected by a subquery are given shared record locks appropriate to the current isolation level.*

This example selects one column from each of two tables for update. It sets READU locks on all rows selected from the ORDERS table and sets READL locks on all rows selected from the CUSTOMER table.

```
>SELECT ORDERS.CUSTNO, CUSTOMER.CUSTID
SQL+FROM ORDERS, CUSTOMER
SQL+WHERE ORDERS.CUSTNO = CUSTOMER.CUSTID
SQL+FOR UPDATE OF ORDERS.CUSTNO;
```

Deleting Data (DELETE)

Deleting data rows from a table is just as common as inserting data. When an entity no longer exists, this must be reflected in the database by removing any rows that represent that entity.

The DELETE statement is structured like the UPDATE statement, and includes a FROM clause naming the table and a WHERE clause for selecting the rows to be deleted. And, like UPDATE, DELETE can operate on just a single row, multiple rows, or all the rows of a table. Also, you can add the REPORTING keyword to see a list of the primary key values (or @ID values if the table has no primary key) for the rows that were deleted.

```
>DELETE FROM EQUIPMENT.T
SQL+WHERE VENDOR_CODE = 110 REPORTING;
UniVerse/SQL: Record "18" deleted.
UniVerse/SQL: 1 record deleted.
```

When you delete a row in this way, you delete all of the multivalues associated with it. Also note that with the referential constraint on EQUIP_CODE in this table, this deletion would not be allowed until that constraint is removed.

Deleting Multivalues from a Row

To delete one or more values from a multivalued column in a row, you must use dynamic normalization so that you can operate on the association row containing the multivalue as if it were a 1NF row.

Recall how dynamic normalization works with a SELECT statement. If you use dynamic normalization to retrieve all the vaccination data for animal 74, you get one row for each vaccination row:

```
>SELECT * FROM LIVESTOCK.T_VAC_ASSOC WHERE ANIMAL_ID = 74;
ANIMAL_ID    VAC_TYPE    VAC_DATE..    VAC_NEXT..    VAC_CERT

       74    R           06/18/93      06/17/96      957640
       74    P           05/24/92      05/24/95      573198
       74    L           06/08/92      06/08/95      772270

3 records listed.
```

To delete the association row for the L vaccination, construct the DELETE statement just as if the association row to be deleted were a row in the table, but use *tablename_association* instead of just *tablename*. Remember to use WHERE instead of WHEN to specify the selection criteria because the virtual table produced by *tablename_association* contains only singlevalued columns.

To delete the vaccination data for vaccination type L in the previous example, enter:

```
>DELETE FROM LIVESTOCK.T_VAC_ASSOC
SQL+WHERE ANIMAL_ID = 74 AND VAC_TYPE = "L";
UniVerse/SQL: 1 record deleted.
```

If you check the table row afterward, you find that all data for vaccination type L has been deleted:

```
>SELECT ANIMAL_ID, VAC_TYPE, VAC_DATE, VAC_NEXT, VAC_CERT
SQL+FROM LIVESTOCK.T WHERE ANIMAL_ID = 74;
ANIMAL_ID    VAC_TYPE    VAC_DATE..    VAC_NEXT..    VAC_CERT

       74    R           06/18/93      06/17/96      957640
             P           05/24/92      05/24/95      573198

1 records listed.
```

Deleting All Rows in a Table

Obviously, omitting the WHERE clause in a DELETE statement can have disastrous effects unless you want to remove a table completely. You do not want to do that to the Circus database. But if you did, and wanted to erase all the data in the EQUIPMENT.T table, enter:

```
>DELETE FROM EQUIPMENT.T;
UniVerse/SQL: 61 records deleted.
```

Then all rows of EQUIPMENT.T would be gone. Note that while all the data has been erased, the EQUIPMENT.T table would still exist in the database, and you could add new rows to it at any time. DROP TABLE is the statement that deletes the table itself.

Deleting Individual Rows

DELETE can be a dangerous statement in any form, even when you remember to include a WHERE clause. It is recommended that you first do a SELECT using the same WHERE clause you will be using in the DELETE to make sure those are the rows you intended.

You may remember that, as one example of using INSERT, you copied all of the 1994 rows from ENGAGEMENTS.T to OLD_ENGAGEMENTS.T for archiving. It would make sense to delete those same rows from ENGAGEMENTS.T.

But before you do so, use a SELECT statement to make sure your WHERE clause selects the rows you want:

```
>SELECT LOCATION_CODE, DATE FROM ENGAGEMENTS.T
SQL+WHERE DATE BETWEEN '1/1/94' AND '12/31/94';
LOCATION_CODE    DATE......

WREN001          01/10/94
WREN001          01/11/94
EMIA001          05/21/94
.
.
.
CSPR001          05/08/94
Press any key to continue...
```

After you are sure that the listed rows are the rows you want to delete, reenter the statement, changing it as shown:

```
>DELETE FROM ENGAGEMENTS.T
SQL+WHERE DATE BETWEEN '1/1/94' AND '12/31/94';
UniVerse/SQL: 53 records deleted.
```

Using Triggers

You can augment and regulate the modification of data in tables by creating a trigger for the table. A trigger specifies actions to perform before or after the execution of certain database modification events. You can define up to six triggers for a table. The names of all triggers and their corresponding UniVerse BASIC programs are stored in the table's SICA.

Use the CREATE TRIGGER statement to create a trigger for a table. You must be the table's owner or have ALTER Privilege on the table, or you must be a DBA to create a trigger.

You can set a trigger to fire (execute) *before* an INSERT, UPDATE, or DELETE event changes data. A BEFORE trigger can examine the new data and determine whether to allow the INSERT, UPDATE, or DELETE event to proceed; if the trigger rejects a data change, UniVerse rolls back the entire transaction. You can also set triggers to fire *after* an INSERT, UPDATE, or DELETE event, for example, to change related rows, audit database activity, and print or send messages.

Using Alternate Dictionaries

All three data modification statements—INSERT, UPDATE, and DELETE—can specify an alternate file dictionary instead of the table's primary file dictionary. As with the SELECT statement, choose this option by adding a USING DICT clause to the statement. For example, to add a new truck to the EQUIPMENT.T table using the alternate dictionary EQUIP_D1, enter:

```
>INSERT INTO EQUIPMENT.T USING DICT EQUIP_D1
SQL+(EQD_CODE, DESCRIPTION, VENDOR_CODE,
SQL+COST, PURCHASE_DATE)
SQL+VALUES (68, '1995 Renault Truck', 90,
SQL+21575.00, '05/23/95');
UniVerse/SQL: 1 record inserted.
```

Alternate dictionaries provide different ways of looking at the same data. For example, one dictionary might use American-style dates, while another might use European-style dates. A table used by several departments within a company might have separate dictionaries because each department uses different terminology to refer to the columns.

Chapter

6

Establishing and Using Views

This chapter discusses *views*. In simplest terms, a view is a *virtual table*. More precisely, a view is the *definition* of a virtual table.

A view is represented in the database as metadata, derives its data from one or more "real" tables (called *base tables* in this context) or even other views, and has a user-defined name. The metadata for a view describes the virtual table in terms of columns and rows of one or more base (physical) tables or views. A view behaves much like a real table. Up through Release 8.3.3 of UniVerse, views are read-only. With Release 9.3.1 and later, some views also are updatable.

Views are useful in a number of ways:

- Data security. Views act as a mask to limit user access to certain rows and columns in the "real" tables.
- Appearance. Views can vary the appearance of a database, presenting a facade that is most familiar to each user.
- Convenience. Instead of naming the same columns and specifying the same complex selection criteria repeatedly in your SELECT statements, you can define a view (which is actually a stored query) once, and reuse it often.

While this chapter discusses views alone, they are closely related to tables. Discussion of how to create tables appears in *UniVerse SQL Administration for DBAs*.

Examples of Views

A view is used commonly for security reasons. Giving a user limited access to the PERSONNEL.T table, for example, allows that user to look at nonsensitive information such as name and address, but hides such data as pay and date of birth. Do not grant the user privileges on the PERSONNEL.T table at all, but instead create a view that encompasses only the columns you want him or her to see, then grant privileges on that view.

Views are always expressed in terms of a query specification (SELECT statement) that defines the data to be included in the view. The general syntax for the CREATE VIEW statement is as follows:

CREATE VIEW *viewname* [(*columnnames*)] **AS SELECT…**

For example, enter:

```
>CREATE VIEW GENERAL1
SQL+AS SELECT NAME, ADR1, ADR2, ADR3
SQL+FROM PERSONNEL.T;
Creating View "GENERAL1"
Adding Column NAME
Adding Column ADR1
Adding Column ADR2
Adding Column ADR3
>GRANT SELECT ON GENERAL1 TO jimc;
Granting privilege(s).
```

To use a view for the sake of convenience, define one that extracts data about big orders and major vendors:

- The ITEM_CODE, DESCRIPTION, and ORDER_QTY columns of the INVENTORY.T table and the COMPANY column of the VENDORS.T table
- Only those rows whose order quantity equals or exceeds 800

The following is an example of a joined view, because it joins the INVENTORY.T and VENDORS.T tables. It also selects data on the basis of rows as well as columns.

```
>CREATE VIEW MAJOR_SUPPLIERS
SQL+AS SELECT INVENTORY.T.ITEM_CODE, DESCRIPTION, ORDER_QTY,
SQL+COMPANY FROM UNNEST INVENTORY.T ON VENDOR_CODE, VENDORS.T
SQL+WHERE INVENTORY.T.VENDOR_CODE = VENDORS.T.VENDOR_CODE
SQL+AND ORDER_QTY >= 800;
Creating View "MAJOR_SUPPLIERS"
Adding Column ITEM_CODE
Adding Column DESCRIPTION
Adding Column ORDER_QTY
Adding Column COMPANY
```

Refer to a view in a SELECT statement as you would any other table, keeping in mind that whatever you ask for is always superimposed on the selection criteria defined in the view:

```
>SELECT COMPANY, ORDER_QTY, DESCRIPTION
SQL+FROM MAJOR_SUPPLIERS
SQL+ORDER BY COMPANY;
COMPANY.................    ORDER_QTY    DESCRIPTION..........

African Environmental              900    T-shirts
Amalgamated Academy                800    Large Cat Chow
Brown Assets                       800    Taffy
California Sourcing                900    Cola
Cincinnati Solutions               900    Sawdust
Citizens Division                  800    Nachos
City Manufacturers                 800    Fried Clams
Cleveland Center                   900    Jerky
Commerce Exchange                  900    Pretzels
Convenient Promotions              800    Cola
.
.
.
Precision Exports                  800    Pretzels
Prime Automation                   800    Handbills
Press any key to continue...
```

This means:

- Retrieve all data selected by MAJOR_SUPPLIERS
  - Inventory item number, inventory description, order quantity, and vendor company
  - For only those items ordered in quantities of 800 or greater
- Display just the vendor company, order quantity, and description, sorted

Once you have created a view, you can perform all sorts of queries on it, without repeating the column names and selection criteria you have defined in the view.

Creating Views

Creating a view involves using a SELECT statement on one or more tables to define the columns and rows to be included in a view, then assigning the view a name. For security reasons, you must have SELECT privilege on all of the tables making up the view. You cannot base a view on a UniVerse file.

A view behaves much like a table and has a file dictionary associated with it. The characteristics of the columns in the view are taken from their definitions in the base tables and cannot be overridden. Thus, in the CREATE VIEW statement you cannot include certain field modifiers (AVERAGE, BREAK ON, BREAK SUPPRESS, CALCULATE, PERCENT, or TOTAL), field qualifiers (SINGLEVALUED, MULTIVALUED, ASSOC, or ASSOCIATION), or any report qualifiers or processing qualifiers. Also, you cannot include an ORDER BY clause in a CREATE VIEW statement. Instead, any ordering must be specified in the SELECT statements that addresses the view.

Using views, you can:

- Select columns in tables and other views
- Select rows in tables and other views
- Select a combination of columns and rows
- Add DISPLAYNAME, CONVERSION, and FORMAT qualifiers
- Add derived columns
- Summarize rows in tables and other views

Column-Based (Vertical) Views

You can define a view that permits access to only certain columns in a table. To build upon the earlier example using the PERSONNEL.T table, the Human Resources and Payroll departments might need access to the entire PERSONNEL.T table, but others—such as ride supervisors, ticket booth managers, and concession supervisors—have a much more limited “need to know.” For example:

- A ride supervisor might need to know only whether Joe Davis has ever worked rides and his hourly rate.

- A concessionaire might need to access the INVENTORY.T table to check on the number of hot dogs in stock and their price, but does not need to know the wholesale cost.

These and other cases can be handled by creating a *vertical*, or column-based, view of the tables in question. Specify in the column list portion of the SELECT statement the columns to include in the view:

```
>CREATE VIEW RIDE_OP_INFO
SQL+AS SELECT BADGE_NO, NAME, RIDE_ID, RIDE_PAY
SQL+FROM PERSONNEL.T;
Creating View "RIDE_OP_INFO"
Adding Column BADGE_NO
Adding Column NAME
Adding Column RIDE_ID
Adding Column RIDE_PAY
Adding association RIDES_ASSOC
>SELECT * FROM RIDE_OP_INFO ORDER BY BADGE_NO;
BADGE_NO    NAME.....................    RIDE_ID    RIDE_PAY..

       1    Nelson, Suzanne                    5         15.93
                                              14         10.93
                                               7         15.21
       3    Grant, Nancy                      13         12.09
                                               1         11.01
                                               4         15.95
       4    Giustino, Carol                    5         10.67
       .
       .
       .
Press any key to continue...
```

Then, if you grant the ride supervisors SELECT privilege on this view (but not on the PERSONNEL.T table itself), they can obtain the information they need without accessing sensitive personnel data. Note that a view inherits any association from the table or view from which it is derived if the view definition includes all columns in that association, as was the case with RIDES_ASSOC in the previous example.

In a sense, a vertical view slices a table into vertical strips (columns), and then combines them into a private table containing only those columns.

Row-Based (Horizontal) Views

You can restrict access to certain rows of a table. This is called a *horizontal* view. Using the PERSONNEL.T table once again, allow access only to records for nonmanagement personnel (employees in any position earning less than $15 per hour):

```
>CREATE VIEW NON_MANAGEMENT AS SELECT *
SQL+FROM PERSONNEL.T
SQL+WHERE EVERY EQUIP_PAY < 15.00 AND EVERY EQUIP_PAY <> 0
SQL+AND EVERY ACT_PAY < 15.00 AND EVERY ACT_PAY <> 0
SQL+AND EVERY RIDE_PAY < 15.00 AND EVERY RIDE_PAY <> 0;
Creating View "NON_MANAGEMENT"
Adding Column BADGE_NO
Adding Column DOB
.
.
.
Adding Column RIDE_PAY
Adding association DEP_ASSOC
Adding association EQUIP_ASSOC
Adding association ACTS_ASSOC
Adding association RIDES_ASSOC
>SELECT NAME, EQUIP_PAY, ACT_PAY, RIDE_PAY FROM NON_MANAGEMENT
SQL+ORDER BY NAME;
NAME.....................    EQUIP_PAY.    ACT_PAY...    RIDE_PAY..

Bailey, Cheryl                     8.24         13.75         12.90
                                  14.40
Carr, Stephen                      9.33          9.41          9.15
                                                                10.70
Clark, Kelly                       8.55         10.79         12.94
                                  13.25         14.86         11.68
                                  13.51          8.16         12.08
                                   8.90                        8.38
Dickinson, Cecilia                13.46          8.17          8.84
                                   8.84         13.96
                                   9.67
Dickinson, Timothy                 9.94         13.23         12.57
.
.
.
Hanson, Allen                     12.70         13.65         10.12
                                                10.79          8.44
                                                              13.28
Press any key to continue...
```

You also could create a view of all personnel who have no dependents:

```
>CREATE VIEW NO_DEPENDENTS AS SELECT *
SQL+FROM PERSONNEL.T
SQL+WHERE EVERY DEP_NAME = '';
Creating View "NO_DEPENDENTS"
Adding Column BADGE_NO
Adding Column DOB
Adding Column BENEFITS
Adding Column NAME
.
.
.
Adding Column RIDE_PAY
Adding association DEP_ASSOC
Adding association EQUIP_ASSOC
Adding association ACTS_ASSOC
Adding association RIDES_ASSOC
>SELECT NAME, DEP_NAME FROM NO_DEPENDENTS ORDER BY NAME;
NAME.....................    DEP_NAME..

Astin, Jocelyn
Bacon, Roger
Bennett, Nicholas
Bowana, Keltu
Burrows, Alan
.
.
.
Press any key to continue...
```

Combined Vertical and Horizontal Views

You can create a view that combines both vertical and horizontal views. This was the case in the earlier example of the MAJOR_SUPPLIERS view, which specified both column names and selection criteria:

```
>CREATE VIEW MAJOR_SUPPLIERS
SQL+AS SELECT INVENTORY.T.ITEM_CODE, DESCRIPTION, ORDER_QTY,
SQL+COMPANY FROM UNNEST INVENTORY.T ON VENDOR_CODE, VENDORS.T
SQL+WHERE INVENTORY.T.VENDOR_CODE = VENDORS.T.VENDOR_CODE
SQL+AND ORDER_QTY >= 800;
```

Slicing a table in both directions is common, and is used for both security and convenience.

Column Names and Derived Columns

Previous examples omitted the optional *columnname,* which assigns view-specific names to the columns. When included, the number of columns listed in *columnnames* must be the same as the number of columns in the SELECT clause.

The *columnnames* option is required for any virtual (derived) column that is part of the view. The *columnnames* option is also useful when you want to supply alternate names for the columns.

Assigning Column Names in a View

The *columnnames* option is a list of alternate names and does not include any field modifiers or qualifiers. Qualifiers, if used, must be supplied in the column specifications of the SELECT clause. If you do not supply *columnnames*, the names of the view columns will be the same as the column names of the tables from which the view is derived.

If the view definition includes derived columns, aggregate functions, or multiple columns of the same name, you *must* include the *columnnames* option in the CREATE VIEW statement.

In the previous example, to assign different column names in the view, enter:

```
>CREATE VIEW MAJOR_SUPPLIERS (INVENTORY_#, ITEM_DESCRIPTION,
SQL+ORDER_QUANTITY, VENDOR_NAME)
SQL+AS SELECT INVENTORY.T.ITEM_CODE, DESCRIPTION, ORDER_QTY,
SQL+COMPANY FROM INVENTORY.T, VENDORS.T
SQL+WHERE INVENTORY.T.VENDOR_CODE = VENDORS.T.VENDOR_CODE
SQL+AND ORDER_QTY >= 800;
```

A direct correspondence exists between the first name listed in *columnnames* and the first column specified in the SELECT clause: INVENTORY_# pairs with ITEM_CODE, ITEM_DESCRIPTION pairs with DESCRIPTION, and so on.

Derived, or Calculated, Columns in a View

Often a view should contain one or more virtual or derived columns, that is, data that doesn't exist in the tables from which the view is derived but is calculated from those columns.

Because such columns do not have defined names, supply names in the *columnnames* portion of the CREATE VIEW statement. For example, to create a view based on the INVENTORY.T table that includes an additional column showing inventory markup (COST / PRICE), enter:

```
>CREATE VIEW INVENTORY1 (DESCRIPTION, WHOLESALE_COST,
SQL+RETAIL_PRICE, MARKUP)
SQL+AS SELECT DESCRIPTION, COST, PRICE, (COST / PRICE)
SQL+FROM INVENTORY.T;
Creating View "INVENTORY1"
Adding Column DESCRIPTION
Adding Column WHOLESALE_COST
Adding Column RETAIL_PRICE
Adding Column MARKUP
```

Note once again that the number of names in *columnnames* matches the number of columns in the SELECT statement.

Summarized Views

Another way to use a view is to summarize data contained in tables, again for either convenience or security. As an example of convenience, people are rarely interested in details but want summaries. A properly defined view can provide an appropriate summary without coding it in a SELECT statement. In terms of security, for example, it might be okay for someone to look at the average employee salary but not individual salaries.

The GROUP BY clause is the most common way to produce a summarized result. To create a view that summarizes the average cost per animal by use category, enter:

```
>CREATE VIEW AVG_COST
SQL+(USE_CATEGORY, AVERAGE_COST)
SQL+AS SELECT USE, AVG(COST)
SQL+FROM LIVESTOCK.T GROUP BY USE;
Creating View 'AVG_COST'
Adding Column USE_CATEGORY
Adding Column AVERAGE_COST
>SELECT * FROM AVG_COST;
USE_CATEGORY    AVERAGE_COST...
P               6105.45
R               6529.73
Z               6112.16

3 records listed.
```

As another example, to create a view that allows someone to find out the average hourly rate for ride operators (but not their individual hourly rates), enter:

```
>CREATE VIEW RIDE_AVG_RATE (AVERAGE_RATE)
SQL+AS SELECT AVG(RIDE_PAY) COL.HDG 'AV RIDE PAY'
SQL+FROM PERSONNEL.T WHERE RIDE_PAY > 0
SQL+AND RIDE_PAY IS NOT NULL;
Creating View "RIDE_AVG_RATE"
Adding Column AVERAGE_RATE
>SELECT * FROM RIDE_AVG_RATE;
AV RIDE PAY

      11.74

1 records listed.
```

Note that this view defines a single result row (average rate) that has no one-to-one correspondence to a row in the source table, PERSONNEL.T. Also note that any qualifiers, such as COL.HDG as shown here, are specified as part of the SELECT statement, not in *columnnames*, because it is the SELECT statement that defines the view.

Updating Views

You can now use an INSERT, DELETE, or UPDATE statement to modify some views. A view is updatable if the user has appropriate rights on the view, and when:

- The FROM clause identifies one table
- The view does not include the keyword DISTINCT
- The table reference identifies either a base table or an updatable view
- There is no subquery into the same table
- There is no GROUP BY, HAVING, WHEN, or UNNEST clause
- Dynamic normalization has not been performed

INSERT is the statement you use for adding *new* data to a view (UPDATE is for changing values in *existing* views). In its most basic form, INSERT names the view where the data is to be inserted and specifies the columns to be filled and the values to be inserted in those columns.

UPDATE modifies the values of one or more columns in one or more selected rows of a view. The UPDATE statement specifies the view to be updated, the columns to be modified, and the rows to be selected.

Deleting data rows from a view is just as common as inserting data. When an entity no longer exists, this must be reflected in the database by removing any rows that represent that entity.

The DELETE statement is structured like the UPDATE statement, and includes a FROM clause naming the view and a WHERE clause for selecting the rows to be deleted. And, like UPDATE, DELETE can operate on just a single row, multiple rows, or all the rows of a table.

You can query the UV_TABLES table in the SQL catalog to find out whether a table is a base table or a view, and to obtain a list of views that are derived from a base table or view.

Views created before Release 9.3.1 of UniVerse are read-only. They must be recreated from the bottom up to be updatable in Release 9.3.1 or later.

For more information about using INSERT, UPDATE, and DELETE, see Chapter 5, “Modifying Data.”

Dropping Views

Remove a view in the same way that you remove a table, through a DROP VIEW statement:

DROP VIEW *viewname* [**CASCADE**];

Issuing a DROP VIEW statement deletes the view from the SQL catalog, deletes its associated file dictionary, and revokes all user privileges on the view. If a view has other views derived from it, you must include the keyword CASCADE to drop those dependent views.

To drop the RIDE_AVG_RATE view created previously, enter:

```
>DROP VIEW RIDE_AVG_RATE;
Dropping View RIDE_AVG_RATE
```

Listing Information About a View

Because a view behaves much like a real table, views also comprise:

- A file dictionary
- A data file (which is empty)
- A SICA (security and integrity constraints area) region

As with tables, you can examine all of these as sources of information about the view.

To see the contents of the file dictionary of a table or view, use the UniVerse LIST command with the DICT keyword:

```
>LIST DICT AVG_COST
DICT AVG_COST    03:44:18pm  10 Jan 1995  Page    1

               Type &
Field......... Field. Field........ Conversion.. Column......Output Depth &
Name.......... Number Definition... Code........ Heading.....Format Assoc..

USE_CATEGORY   D    1                                        1L     S
AVERAGE_COST   D    2               MD22                     12R    S
@REVISE        PH     USE_CATEGORY
                      AVERAGE_COST
@              PH     ID.SUP
                      USE_CATEGORY
                      AVERAGE_COST

4 records listed.
```

Alternatively, you could enter:

```
>SELECT * FROM DICT AVG_COST;
```

To print the dictionary, use either the RetrieVe PRINT.DICT command or the UniVerse SQL SELECT statement:

```
>PRINT.DICT AVG_COST
>SELECT * FROM DICT AVG_COST LPTR;
```

To see the contents of a view's data, use the UniVerse SQL SELECT statement. Note that you cannot use RetrieVe commands on the view itself.

```
>SELECT * FROM AVG_COST;
```

Finally, to see the information in a view's SICA region, use the following command:

```
>LIST.SICA AVG_COST
LIST.SICA AVG_COST 11:35:41AM  02 May 1995   Page    1
========================================
Sica Region for View "AVG_COST"

  Schema:           CIRCUS
  Revision:         2
  Checksum is:      8263
    Should Be:      8263
  Size:             304
  Creator:          719
  Total Col Count:  2
    Key Columns:    0
    Data Columns:   2
  Check Count:      0
  Permission Count:0
  History Count:    0

  Query specification: SELECT USE , AVG ( COST ) FROM
   LIVESTOCK.T GROUP BY USE
  Underlying Tables: CIRCUS.LIVESTOCK.T
    WITH CHECK OPTION:      No

  Data for Column "USE_CATEGORY"

    Position:       1
    Key Position:   0
    Multivalued:    No
    Not Null:       No
    Not Empty:      No
    Unique:         No
    Row Unique:     No
    Primary Key:    No
    Default Type:   None
    Data Type:      CHARACTER
Press any key to continue...
```

Privileges and Views

Privileges work with views as they do with tables, but practically speaking, the only table privilege applicable to views is the SELECT privilege:

- When you create a view, you are the owner of that view and, as such, automatically have SELECT privilege on it.
- You can access a view only if you are its creator or have otherwise been granted SELECT privilege on it.
- You can grant the SELECT privilege on your view to other users, provided that you own the tables comprising the view, or the owners of the tables have granted you SELECT privilege WITH GRANT OPTION.

Because views are based on tables and other views, one consideration unique to views is that before you can create a view, you must have SELECT privilege on all of its underlying tables and views.

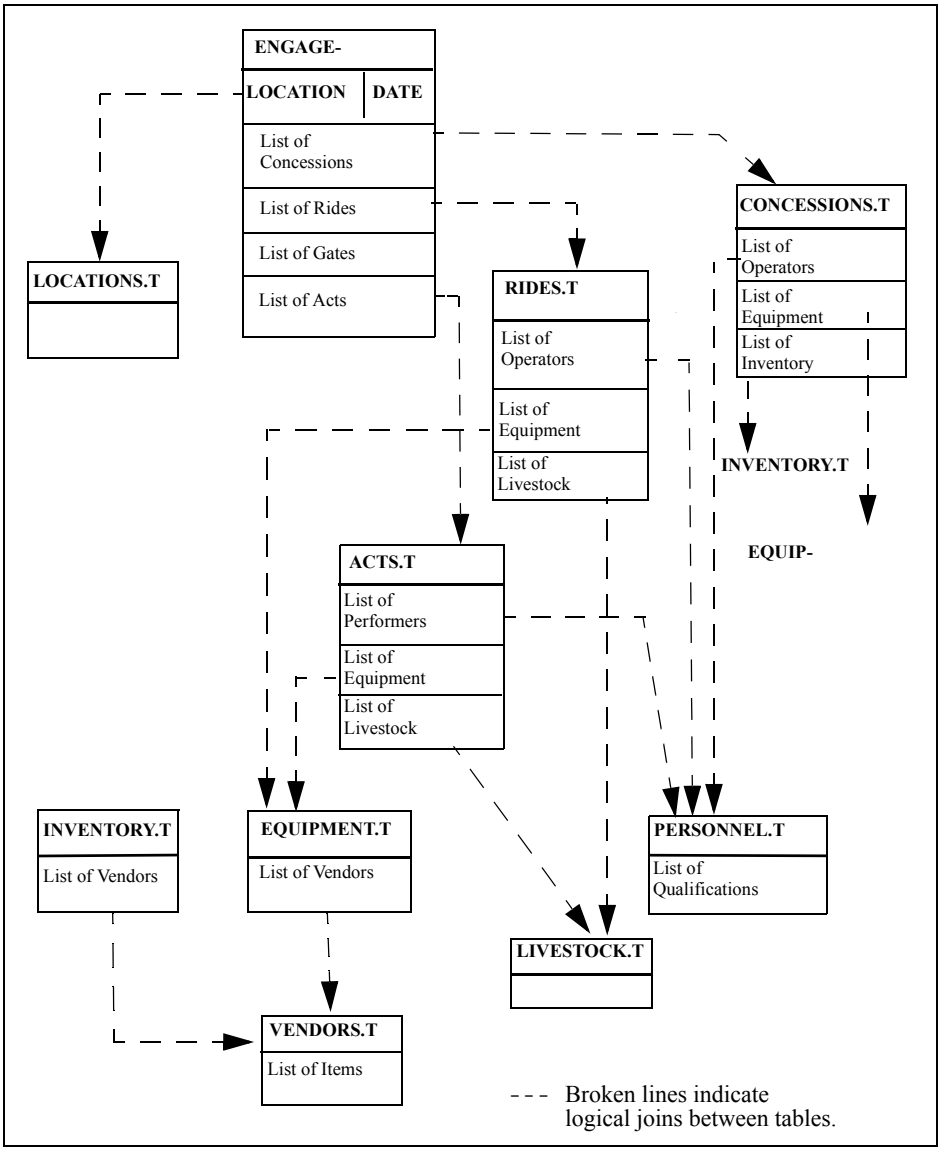
Appendix

A

The Sample Database

This appendix presents the structure of Circus, the sample database used for the examples in this manual. The descriptions of the tables are presented alphabetically in the form of their CREATE TABLE statements.

The Circus database is illustrated in the following illustration.



The Sample Database

ACTS.T Table

```
CREATE TABLE ACTS.T (
         ACT_NO             INT FMT '5R' PRIMARY KEY,
         DESCRIPTION        VARCHAR FMT '6T',
         DURATION           INT FMT '5R',
         OPERATOR           INT FMT '5R' MULTIVALUED,
         ANIMAL_ID          INT FMT '5R' MULTIVALUED
                            REFERENCES LIVESTOCK.T,
         EQUIP_CODE         INT FMT '5R' MULTIVALUED
                            REFERENCES EQUIPMENT.T
         );
```

CONCESSIONS.T Table

```
CREATE TABLE CONCESSIONS.T (
         CONC_NO            INT FMT '5R' PRIMARY KEY,
         DESCRIPTION        VARCHAR FMT '25T',
         OPERATOR           INT FMT '5R' MULTIVALUED
                            REFERENCES PERSONNEL.T,
         EQUIP_CODE         INT FMT '5R' MULTIVALUED
                            REFERENCES EQUIPMENT.T,
         ITEM_CODE          INT FMT '5R' MULTIVALUED
                            NOT NULL ROWUNIQUE
                            REFERENCES INVENTORY.T,
         QTY                INT FMT '5R' MULTIVALUED,
         ASSOCIATION        STOCK (ITEM_CODE KEY, QTY)
         );
```

ENGAGEMENTS.T Table

```
CREATE TABLE ENGAGEMENTS.T (
         LOCATION_CODE     CHAR(7) FMT '7L',
         "DATE"            DATE CONV 'D2/',
         "TIME"            TIME CONV 'MTH',
         ADVANCE           DEC(9,2) FMT '12R',
         GATE_NUMBER       INT FMT '5R' MULTIVALUED
                           NOT NULL ROWUNIQUE,
         GATE_REVENUE      DEC(9,2) FMT '12R'
         MULTIVALUED,
         GATE_TICKETS      INT FMT '5R' MULTIVALUED,
         ACT_NO            INT FMT '5R' MULTIVALUED
                           NOT NULL ROWUNIQUE
                           REFERENCES ACTS.T,
         RIDE_ID           INT FMT '5R' MULTIVALUED
                           NOT NULL ROWUNIQUE
                           REFERENCES RIDES.T,
         RIDE_REVENUE      DEC(9,2) FMT '12R'
         MULTIVALUED, RIDE_TICKETSINT FMT '5R'
         MULTIVALUED,
         CONC_ID           INT FMT '5R' MULTIVALUED
                           NOT NULL ROWUNIQUE
                           REFERENCES CONCESSIONS.T,
         CONC_REVENUE      DEC(9,2) FMT '12R'
         MULTIVALUED,
         CONC_TICKETS      INT FMT '5R' MULTIVALUED,
         LABOR             INT FMT '5R',
         PAY               DEC(5,2) FMT '10R',
         ASSOCIATION       GATES_ASSOC (GATE_NUMBER KEY,
                           GATE_REVENUE, GATE_TICKETS),
         ASSOCIATION       CONCS_ASSOC (CONC_ID KEY,
                           CONC_REVENUE, CONC_TICKETS),
         ASSOCIATION       RIDES_ASSOC (RIDE_ID KEY,
                           RIDE_REVENUE, RIDE_TICKETS),
         PRIMARY KEY       (LOCATION_CODE, "DATE")
         );
```

EQUIPMENT.T Table

```
CREATE TABLE EQUIPMENT.T (
         EQUIP_CODE         INT FMT '5R' PRIMARY KEY,
         VENDOR_CODE        INT FMT '5R'
                            REFERENCES VENDORS.T,
         VENDOR_REF         VARCHAR FMT '10L',
         DEPRECIATION       CHAR(1) FMT '1L',
         DESCRIPTION        VARCHAR FMT '25T',
         COST               DEC(9,2) FMT '12R',
         USE_LIFE           INT FMT '5R',
         TAX_LIFE           INT FMT '5R',
         VOLTS              INT FMT '5R',
         PURCHASE_DATE      DATE CONV 'D2/'
         );
```

INVENTORY.T Table

```
CREATE TABLE INVENTORY.T (
         ITEM_CODE          INT FMT '5R' PRIMARY KEY,
         ITEM_TYPE          CHAR(1) FMT '1L',
         DESCRIPTION        VARCHAR FMT '25T',
         QOH                INT FMT '5R',
         COST               DEC(9,2) FMT '12R',
         PRICE              DEC(9,2) FMT '12R',
         VENDOR_CODE        INT FMT '5R' MULTIVALUED
                            NOT NULL ROWUNIQUE
                            REFERENCES VENDORS.T,
         ORDER_QTY          INT FMT '5R' MULTIVALUED,
         ASSOCIATION        ORDERS_ASSOC (VENDOR_CODE
         KEY,
                            ORDER_QTY)
         );
```

LIVESTOCK.T Table

```
CREATE TABLE LIVESTOCK.T (
         ANIMAL_ID          INT FMT '5R' PRIMARY KEY,
         NAME               VARCHAR FMT '10T',
         DESCRIPTION        VARCHAR FMT '10T',
         USE                CHAR(1) FMT '1L',
         DOB                DATE CONV 'D2/',
         ORIGIN             VARCHAR FMT '12T',
         COST               DEC(9,2) FMT '12R',
         EST_LIFE           INT FMT '3R',
         VAC_TYPE           CHAR(1) FMT '1L' MULTIVALUED
                            NOT NULL ROWUNIQUE,
         VAC_DATE           DATE CONV 'D2/' MULTIVALUED,
         VAC_NEXT           DATE CONV 'D2/' MULTIVALUED,
         VAC_CERT           VARCHAR FMT '6L' MULTIVALUED,
         ASSOCIATION        VAC_ASSOC (VAC_TYPE KEY,
                            VAC_DATE, VAC_NEXT, VAC_CERT)
         );
```

LOCATIONS.T Table

```
CREATE TABLE LOCATIONS.T (
         LOCATION_CODE      CHAR(7) FMT '7L' PRIMARY KEY,
         DESCRIPTION        VARCHAR FMT '25T',
         NAME               VARCHAR FMT '25T',
         ADR1               VARCHAR FMT '25T',
         ADR2               VARCHAR FMT '25T',
         ADR3               VARCHAR FMT '25T',
         PHONE              VARCHAR FMT '12L',
         FAX                VARCHAR FMT '8L',
         ACRES              INT FMT '5R',
         SEATS              INT FMT '5R',
         PARKS              INT FMT '5R',
         MEDIA_NAME         VARCHAR FMT '25L' MULTIVALUED
                            NOT NULL ROWUNIQUE,
         MEDIA_CONTACT      VARCHAR FMT '25L'
         MULTIVALUED,
         MEDIA_PHONE        VARCHAR FMT '12L'
         MULTIVALUED,
         MEDIA_FAX          VARCHAR FMT '8L' MULTIVALUED,
         GOV_AGENCY         VARCHAR FMT '25L' MULTIVALUED
                            NOT NULL ROWUNIQUE,
         GOV_CONTACT        VARCHAR FMT '25L'
         MULTIVALUED,
         GOV_PHONE          VARCHAR FMT '12L'
         MULTIVALUED,
         GOV_FAX            VARCHAR FMT '8L' MULTIVALUED,
         GOV_FEE            DEC(9,2) FMT '12R'
         MULTIVALUED,
         GOV_CHECK          VARCHAR FMT '5L' MULTIVALUED,
         GOV_RATE           DEC(3,3) FMT '7R'
         MULTIVALUED,
         ASSOCIATION        MEDIA_ASSOC (MEDIA_NAME KEY,
                            MEDIA_CONTACT, MEDIA_PHONE,
                            MEDIA_FAX),
         ASSOCIATION        GOV_ASSOC (GOV_AGENCY KEY,
                            GOV_CONTACT, GOV_PHONE,
         GOV_FAX,
                            GOV_FEE, GOV_CHECK, GOV_RATE)
         );
```

PERSONNEL.T Table

```
CREATE TABLE PERSONNEL.T (
         BADGE_NO           INT FMT '5R' PRIMARY KEY,
         DOB                DATE CONV 'D2/',
         BENEFITS           VARCHAR FMT '10T',
         NAME               VARCHAR FMT '25T',
         ADR1               VARCHAR FMT '25T',
         ADR2               VARCHAR FMT '25T',
         ADR3               VARCHAR FMT '25T',
         PHONE              VARCHAR FMT '12L',
         DEP_NAME           VARCHAR FMT '10T' MULTIVALUED
                            NOT NULL ROWUNIQUE,
         DEP_DOB            DATE CONV 'D2/' MULTIVALUED,
         DEP_RELATION       VARCHAR FMT '5L' MULTIVALUED,
         EQUIP_CODE         INT FMT '5R' MULTIVALUED
                            NOT NULL ROWUNIQUE
                            REFERENCES EQUIPMENT.T,
         EQUIP_PAY          DEC(5,2) FMT '10R'
         MULTIVALUED,
         ACT_NO             INT FMT '5R' MULTIVALUED
                            NOT NULL ROWUNIQUE
                            REFERENCES ACTS.T,
         ACT_PAY            DEC(5,2) FMT '10R'
         MULTIVALUED,
         RIDE_ID            INT FMT '5R' MULTIVALUED
                            NOT NULL ROWUNIQUE
                            REFERENCES RIDES.T,
         RIDE_PAY           DEC(5,2) FMT '10R'
         MULTIVALUED,
         ASSOCIATION        DEP_ASSOC (DEP_NAME KEY,
         DEP_DOB,
                            DEP_RELATION),
         ASSOCIATION        EQUIP_ASSOC (EQUIP_CODE KEY,
                            EQUIP_PAY),
         ASSOCIATION        ACTS_ASSOC (ACT_NO KEY,
         ACT_PAY),
         ASSOCIATION        RIDES_ASSOC (RIDE_ID KEY,
                            RIDE_PAY)
         );
```

RIDES.T Table

```
CREATE TABLE RIDES.T (
         RIDE_ID            INT FMT '5R' PRIMARY KEY,
         DESCRIPTION        VARCHAR FMT '20T',
         OPERATOR           INT FMT '5R' MULTIVALUED,
         ANIMAL_ID          INT FMT '5R' MULTIVALUED
                            REFERENCES LIVESTOCK.T
         EQUIP_CODE         INT FMT '5R' MULTIVALUED
                            REFERENCES EQUIPMENT.T
         );
```

VENDORS.T Table

```
CREATE TABLE VENDORS.T (
         VENDOR_CODE        INT FMT '5R' PRIMARY KEY,
         COMPANY            VARCHAR FMT '25T',
         ADR1               VARCHAR FMT '25T',
         ADR2               VARCHAR FMT '25T',
         ADR3               VARCHAR FMT '25T',
         TERMS              VARCHAR FMT '10T',
         CONTACT            VARCHAR FMT '25T',
         PHONE              VARCHAR FMT '12L',
         FAX                VARCHAR FMT '8L',
         EQUIP_CODE         INT FMT '5R' MULTIVALUED,
         ITEM_CODE          INT FMT '5R' MULTIVALUED
                            NOT NULL ROWUNIQUE,
         LEAD_TIME          INT FMT '5R' MULTIVALUED,
         ASSOCIATION        PROD_ASSOC (ITEM_CODE KEY,
                            LEAD_TIME)
         );
```

Glossary

1NF	See **first normal form**.
account	User accounts are defined at the operating system level. Each user account has a user name, a user ID number, and a home directory. UniVerse accounts are defined in the UV.ACCOUNT file of the UV account. Each UniVerse account has a name and resides in a directory that contains special UniVerse files such as the VOC, &SAVEDLISTS&, and so on. See also **schema**.
aggregate functions	See **set functions**.
alias	A name assigned to a table, column, or value expression that lasts for the duration of the statement. See also **correlation name**.
ANSI	American National Standards Institute. A U.S. organization charged with developing American national standards.
association	A group of related multivalued columns in a table. The first value in any association column corresponds to the first value of every other column in the association, the second value corresponds to the second value, and so on. An association can be thought of as a nested table.
association depth	For any base table row, the number of values in the association key columns determines the association depth. If an association does not have keys, the column with the most association rows determines the association depth.
association key	The values in one or more columns of an association that uniquely identify each row in the association. If an association does not have keys, the @ASSOC_ROW keyword can generate unique association row identifiers.

association row	A sequence of related data values in an association. A row in a nested table.
authority	See **database privilege**.
BASIC SQL Client Interface	The UniVerse BASIC application programming interface (API) that lets application programmers write client programs using SQL function calls to access data in SQL server databases.
BNF	Backus Naur Form. A notation format using a series of symbols and production rules that successively break down statements into their components. Appendix A, "The Sample Database," shows UniVerse SQL syntax in BNF.
Boolean	See **logical values**, **three-valued logic**.
Cartesian product	All possible combinations of rows from specified tables.
CATALOG schema	The schema that contains the SQL catalog.
cell	The intersection of a row and a column in a table. In UniVerse SQL, cells can contain more than one value. Such values are often called *multivalues*. See also **multivalued column**.
character string	A set of zero or more alphabetic, numeric, and special characters. Character strings must be enclosed in single quotation marks.
check constraint	A condition that data to be inserted in a row must meet before it can be written to a table.
client	A computer system or program that uses the resources and services of another system or program (called a server).
column	A set of values occurring in all rows of a table and representing the same kind of information, such as names or phone numbers. A field in a table. See also **multivalued column**, **row**, **cell**, **table**.
comparison operator	See **relational operator**.
concurrency control	Methods, such as locking, that prevent two or more users from changing the same data at the same time.
CONNECT	The database privilege that grants users access to UniVerse SQL. Users with CONNECT privilege are registered in the SQL catalog. See also **registered users**.

connecting columns	Columns in one or more tables that contain similar values. In a join, the connecting column enables a table to link to another table or to itself.
constant	A data value that does not change. See also **literal**.
constraint	See **integrity constraint**.
correlated subquery	A subquery that depends on the value produced by an outer query for its results.
correlation name	A name assigned to a table, column, or value expression, that can be used in a statement as a qualifier or as the name of an unnamed column.
DBA	Database administrator. DBA is the highest-level database privilege. Like superuser, a user with DBA privilege has complete access to all SQL objects in the database.
DBMS	Database management system.
DDL	Data definition language.
DML	Data manipulation language.
database privilege	Permission to access SQL database objects. See also **CONNECT**, **RESOURCE**, **DBA**, **privilege**.
default value	The value inserted into a column when no value is specified.
depth	See **association depth**.
dynamic normalization	A mechanism for letting DML statements access an association of multivalued columns or an unassociated multivalued column as a virtual first-normal-form table.
effective user name	In a BASIC program, the user specified in an AUTHORIZATION statement; otherwise, the user who is logged in as running the program.
empty string	A character string of zero length. This is not the same as the null value.
expression	See **value expression**.
field	See **column**.
first normal form	The name of a kind of relational database that can have only one value for each row and column position (or cell). Its abbreviation is 1NF.
foreign key	The value in one or more columns that references a primary key or unique column in the same or in another table. Only values in the referenced column can be included in the foreign key column. See also **referential constraint**.

identifier	The name of a user or an SQL object such as a schema, table, or column.
inclusive range	The range specified with the BETWEEN keyword that includes the upper and lower limits of the range.
integrity constraint	A condition that data to be inserted in a row must meet before it can be written to a table.
isolation level	A mechanism for separating a transaction from other transactions running concurrently, so that no transaction affects any of the others. There are five isolation levels, numbered 0 through 4.
join	Combining data from more than one table.
join column	A column used to specify join conditions.
key	A data value used to locate a row.
keyword	A word, such as SELECT, FROM, or TO, that has special meaning in UniVerse SQL statements.
literal	A constant value. UniVerse SQL has four kinds of literal: character strings, numbers, dates, and times.
logical values	Value expressions can have any of the following logical values: true (1), false (0), or unknown (NULL).
multivalued column	A column that can contain more than one value for each row in a table. See also **cell**, **association**.
NF^2	See **nonfirst-normal form**.
nested query	See **subquery**.
nested sort	A sort within a sort.
nested table	See **association**.
nonfirst-normal form	The name of a kind of relational database that can have more than one value for a row and column position (or cell). Its abbreviation is NF^2. Thus, the UniVerse nonfirst-normal-form database can be thought of as an *extended relational database*.
NT AUTHORITY \SYSTEM	On Windows platforms, the user name of the database administrator (DBA) who owns the SQL catalog.

null value	A special value representing an unknown value. This is not the same as 0 (zero), a blank, or an empty string.
ODBC	Open Database Connectivity. A programming language interface for connecting to databases.
outer query	A query whose value determines the value of a correlated subquery.
outer table	The first table specified in an outer join expression.
owner	The creator of a database object such as a schema or table. The owner has all privileges on the object.
parameter marker	In a programmatic SQL statement, a single ? (question mark) used in place of a constant. Each time the program executes the statement, a value is used in place of the marker.
permissions	See **privilege**.
precision	The number of significant digits in a number. See also **scale**.
primary key	The value in one or more columns that uniquely identifies each row in a table.
primary key constraint	A column or table constraint that defines the values in specified columns as the table's primary keys. Primary keys cannot be null values and must also be unique. If a table has no primary key, the @ID column functions as an implicit primary key.
privilege	Permission to access, use, and change database objects. See also **database privilege**, **table privilege**.
programmatic SQL	A dialect of the UniVerse SQL language used in client programs that access SQL server databases. Programmatic SQL differs from interactive SQL in that certain keywords and clauses used for report formatting are not supported.
qualifier	An identifier prefixed to the name of a column, table, or alias to distinguish names that would otherwise be identical.
query	A request for data from the database.
record	See **row**.
referenced column	A column referenced by a foreign key column. See also **referential constraint**.
referencing column	A foreign key column that references another column. See also **referential constraint**.

referential constraint	A column or table constraint that defines a dependent relationship between two columns. Only values contained in the referenced column can be inserted into the referencing column. See also **foreign key**.
reflexive join	A join that joins a table to itself. Both join columns are in the same table.
registered users	Users with CONNECT privilege, whose names are listed in the SQL catalog. Registered UniVerse SQL users can create and drop tables, grant and revoke privileges on tables on which they have privileges, and so on.
relational operator	An operator used to compare one expression to another in a WHERE, WHEN, or HAVING clause, or in a check constraint. Relational operators include = (equal to), > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to), and <> (not equal to).
RESOURCE	Second-highest level database privilege. A user with RESOURCE privilege can create schemas.
root	On UNIX systems, the user name of the database administrator (DBA) who owns the SQL catalog if *uvsql* or *uvadm* is not the owner.
row	A sequence of related data elements in a table; a record. See also **column**, **cell**, **table**.
rowunique constraint	A column or table constraint requiring that values in the cells of specified multivalued columns must be unique in each cell. Values need not be unique throughout each column, but only in each row of each column.
scale	The number of places to the right of the decimal point in a number. See also **precision**.
schema	A group of related tables and files contained in a UniVerse account directory and listed in the SQL catalog.
security constraint	A condition that users must meet before they can perform a specified action on a table.
server	A computer system or program that provides resources and services to other systems or programs (called clients).
set functions	Arithmetic functions that produce a single value from a group of values in a specific column. Set functions include AVG, COUNT, COUNT(*), MAX, MIN, and SUM. Set functions can be used only in the SELECT and HAVING clauses of the SELECT statement.

SICA	Security and integrity constraints area. This is an area of each table where data structure, privileges, and integrity constraints are defined and maintained.
SQL	A language for defining, querying, modifying, and controlling data in a relational database.
SQL catalog	A set of tables that describe all SQL objects, privileges, and users in the system: UV_ASSOC, UV_COLUMNS, UV_SCHEMA, UV_TABLES, UV_USERS, and UV_VIEWS. The SQL catalog is located in the CATALOG schema.
SQL Client Interface	See **BASIC SQL Client Interface**.
statement	An SQL command that defines, manipulates, or administers data.
string	See **character string**.
subquery	A SELECT statement that nests within a WHERE, WHEN, or HAVING clause.
table	A matrix of rows and columns containing data. See also **column**, **row**, **cell**.
table privilege	Permission to read or write to a table. These include SELECT, INSERT, UPDATE, DELETE, ALTER, and REFERENCES. See also **privilege**.
temporary name	See **alias**.
three-valued logic	An extension of Boolean logic that includes a third value, unknown (NULL), in addition to the Boolean values true (1) and false (0). See also **logical values**.
transaction	A strategy that treats a group of database operations as one unit. The database remains consistent because either all or none of the operations are completed.
transaction management	A strategy that either completes or cancels transactions so that the database is never inconsistent.
trigger	A BASIC program associated with a table, executed (“fired”) when some action changes the table’s data.
UCI	Uni Call Interface. A C-language application programming interface (API) that lets application programmers write client programs using SQL function calls to access data in UniVerse databases.
unique constraint	A column or table constraint requiring that values in specified columns must contain unique values.

unnested table	The result of unnesting, or exploding, an association of multivalued columns to produce a separate row for each set of associated multivalues. Unnested data is treated as singlevalued.
user privilege	See **database privilege**.
uvadm	On UNIX systems, the user name of the database administrator (DBA). *uvadm* is the owner of the SQL catalog if *uvsql* or *root* is not the owner.
uvsql	On UNIX systems, the user name of the database administrator (DBA). *uvsql* is the owner of the SQL catalog if *root* or *uvadm* is not the owner.
value expression	One or more literals, column specifications, and set functions, combined with arithmetic operators and parentheses, that produce a value when evaluated.
view	A derived table created by a SELECT statement that is part of the view's definition.
wildcard	Either of two characters used in pattern matches. The _ (underscore) represents any single character. The % (percent sign) represents any number of characters.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W