IBM

UniVerse

SQL Reference

Version 10.3
February, 2009

Documentation Team: Claire Gustafson, Shelley Thompson, Anne Waite

Table of Contents

Preface

This book is a reference guide to UniVerse SQL. It is for application developers and system administrators who are familiar with UniVerse and with ANSI-standard SQL. It contains reference pages for all UniVerse SQL statements, data types, and SQL catalog tables. It is a companion volume to *UniVerse SQL User Guide* and *UniVerse SQL Administration for DBAs*.

Organization of This Manual

This manual contains the following chapters:

Chapter 1, "Introduction," introduces UniVerse SQL.

Chapter 2, "The SQL Catalog," describes the tables in the SQL catalog.

Chapter 3, "Data Types," describes UniVerse SQL data categories and data types.

Chapter 4, "UniVerse SQL in Client Programs," describes how to use UniVerse SQL in programs.

Chapter 5, "Triggers," describes how to use triggers.

Chapter 6, "UniVerse SQL Statements," describes all UniVerse SQL statements.

Appendix A, "UniVerse SQL Grammar," describes the UniVerse SQL grammar in Backus Naur Form.

Appendix B, "Reserved Words," lists the reserved words in UniVerse SQL.

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: `FLD1FVAL1VSUBV1SSUBV2`
V	Value mark. For example, the value mark (**V**) in the following string delimits elements VAL1 and SUBV1: `FLD1FVAL1VSUBV1SSUBV2`
S	Subvalue mark. For example, the subvalue mark (**S**) in the following string delimits elements SUBV1 and SUBV2: `FLD1FVAL1VSUBV1SSUBV2`
T	Text mark. For example, the text mark (**T**) in the following string delimits elements 4 and 5: `1F2S3V4T5`

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. The continuation lines are indented. 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 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 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 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 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 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

Introduction

1

UniVerse SQL is both a database language and a set of capabilities. Using SQL you can query and update data in UniVerse files as well as in SQL tables. Starting with Release 8.3.3 of UniVerse, you can use UniVerse SQL in client programs as well as interactively.

UniVerse SQL conforms to the ANSI/ISO 1989 standard established for SQL, enhanced to take advantage of the postrelational database structure of UniVerse. In contrast to first-normal-form (1NF) databases, which can have only one value for each row and column position (or cell), UniVerse is a nonfirst-normal-form (NF^2) database, which can hold more than one value in a cell. UniVerse also supports nested tables called associations, which are made up of a group of related multivalued columns in a table.

Programmatic SQL in UniVerse conforms to ANSI-1989 SQL with Integrity Enhancements (Level 1), and it includes many features from ANSI-1992 SQL and most features from the ODBC core level of SQL grammar.

UniVerse SQL Syntax Conventions

In this book the presentation of SQL syntax differs in some respects from the presentation of syntax in other UniVerse documentation. Because SQL statements can be complex and detailed, we first present an overview of syntax elements with brief descriptions of what each element does. For example, the overview of SELECT statement syntax first describes each type of clause you can use, without presenting complete details of each clause.

Subsequent sections describe each syntax element in complete detail. Because many of the same syntax elements, such as table and column specifications, conditional expressions, and so forth, are common to several SQL statements, they are treated separately, each in its own section, making them easier to understand and use. For example, since the SELECT, INSERT, UPDATE, and DELETE statements can all specify tables, you will find a separate section, "Table," that describes the table expression syntax to use in these four statements.

Examples Used in This Book

The examples used in this book are based on two databases:

- UniVerse demonstration database
- Circus database

The UniVerse Demonstration Database

The UniVerse demonstration database comprises three tables, CUSTOMERS, ORDERS, and INVENTORY, which are derived from the UniVerse demonstration database. For information about the demonstration database, see the INITIALIZE.DEMO command in *UniVerse User Reference*.

If you want to use these tables on your system, do the following:

1. Use the INITIALIZE.DEMO command to install copies of the UniVerse files in your account.
2. If you are defined in the SQL catalog as an SQL user with RESOURCE privilege, use the CREATE SCHEMA statement to make the account an SQL schema, then proceed to step 3.

   If you do not have RESOURCE Privilege, or if you are not an SQL user, have your database administrator (DBA) register you as an SQL user and make the account into a schema with you as the schema's owner.
3. Use the CONVERT.SQL command to turn the UniVerse files into tables:

   ```
   CONVERT.SQL CUSTOMERS CREATE GEN
   CONVERT.SQL ORDERS CREATE GEN
   CONVERT.SQL INVENTORY CREATE GEN
   ```

The Circus Database

The Circus database is a more complex database based on the activities of a travelling circus. This database comprises 10 tables and is fully described in *UniVerse SQL User Guide*. To install the Circus database on your system, complete the following steps:

1. Create a UniVerse account or choose an existing UniVerse account to contain the Circus database.

2. If you are defined in the SQL catalog as an SQL user with RESOURCE privilege, use the CREATE SCHEMA statement to make the account an SQL schema, then proceed to step 3.

   If you do not have RESOURCE Privilege, or if you are not an SQL user, have your database administrator (DBA) use the SETUP.DEMO.SCHEMA command to register you as an SQL user and make the account into a schema called DEMO_*username* with you as the schema's owner.

3. Log on to the new schema and use the MAKE.DEMO.TABLES command to create the tables and load data into them. The table names all have the suffix .T . You are the owner of the tables.

If you want to drop these tables later, use the REMOVE.DEMO.TABLES command.

Chapter

The SQL Catalog

2

This chapter describes the following:

- The structures of all SQL catalog tables
- How to find and fix SQL catalog inconsistencies

What Is the SQL Catalog?

The SQL catalog is a schema containing six tables that define the database. These tables describe the following:

- Each schema: its name, owner, and full path
- Each table: its name, owner, number of columns, size, and so on
- Each view: its name and the SELECT statement that creates it
- Each column: its name, table, data type, size, whether null values are allowed, and so on
- Each association of multivalued columns: its name, table, order, and so on
- Each user: the schemas and tables they own, and the database and table privileges they have

The UniVerse installation process creates the CATALOG schema and the six tables that make up the SQL catalog in the UV account directory. The path is uvhome*/sql/catalog* (where *uvhome* is the path of the UV account directory on your system). The UniVerse administrator determines the owner of the directory and tables that make up the SQL catalog. On UNIX systems, the SQL catalog owner can be either *uvsql*, *root*, or *uvadm*. On Windows platforms, any member of the Administrators group is an owner of the SQL catalog. The default owner ID on Windows platforms is NT AUTHORITY\SYSTEM.

Structure of the SQL Catalog

The following sections describe the SQL catalog tables:

- UV_ASSOC
- UV_COLUMNS
- UV_SCHEMA
- UV_TABLES
- UV_USERS
- UV_VIEWS

UV_ASSOC

The UV_ASSOC table describes all associations of multivalued columns in all tables in the database. It has the following columns:

Column Name	Data Type	Description
ASSOC_SCHEMA	CHAR(18)	Name of the schema where the association's table is located.
ASSOC_NAME	CHAR(18)	Name of the association.
ORDERING	CHAR(10)	One of the following: LAST: Association rows are added after existing association rows. FIRST: Association rows are added before existing association rows. IN_COL_BY: Association rows are inserted according to the sequence indicated in the ORDER_TYPE column. PRESERVING: Association rows are added according to the position specified by @ASSOC_ROW. The positions of the association rows are stable. UNORDERED: Same as LAST.
ORDER_COLUMN	CHAR(18)	If ORDERING is IN_COL_BY, this column contains the name of the column whose data determines the order in which to insert association rows.

UV_ASSOC Columns

Column Name	Data Type	Description
ORDER_TYPE	CHAR(2)	If ORDERING is IN_COL_BY, the order type is one of the following: AL: Association rows are inserted in ascending order, left-justified. AR: Association rows are inserted in ascending order, right-justified. DL: Association rows are inserted in descending order, left-justified. DR: Association rows are inserted in descending order, right-justified.
EMPTY_ROW	CHAR(3)	*Reserved for future use*.
SICA	CHAR(254)	*Reserved for future use*.

UV_ASSOC Columns (Continued)

ASSOC_SCHEMA and ASSOC_NAME form the primary key.

UV_COLUMNS

The UV_COLUMNS table describes all columns of all tables in the database. It has the following columns:

Column Name	Data Type	Description
TABLE_SCHEMA	CHAR(18)	Name of the schema where the column's table is located.
TABLE_NAME	CHAR(18)	Name of the table containing the column.
COLUMN_NAME	CHAR(18)	Name of the column.
IN_ASSOCIATION	CHAR(18)	Name of the association of multivalued columns to which the column belongs.
AMC	INT	AMC is a multivalued column that always contains two values. The first value is the column number, representing the position of the column in the table as defined in the CREATE TABLE statement. Columns making up the primary key are together considered column 0. The second value is either: A number representing the position of the column in the primary key, starting with 1. 0 for all non-primary-key columns.
ACOL_NO	INT	ACOL_NO is a multivalued column. If the column is not part of an association or a primary key, ACOL_NO is empty. If the column is part of an association or a primary key, ACOL_NO has two values: The first value is either: 0 for primary key and association key columns.
ACOL_NO (continued)		A number representing the position of the column in the association, starting with 1.

UV_COLUMNS Columns

Column Name	Data Type	Description
		The second value is one of the following: For primary key columns, a number representing the position of the column in the primary key, starting with 1. For association key columns, a number representing the position of the column in the association, starting with the number following the last primary key column. 0 for columns that are not part of the association key.
MULTI_VALUE	CHAR(1)	S: Column is single-valued. M: Column is multivalued.
DATA_TYPE	CHAR(8)	Type of data in the column (see Chapter 3, "Data Types," for information about data types).
CHAR_MAX_LENGTH	INT	Maximum column length of a column whose data type is CHAR or VARCHAR.
NUMERIC_PRECISION	INT	Precision of a column whose data type is FLOAT.
NUMERIC_PREC_RADIX	INT	Precision of a column whose data type is DECIMAL or NUMERIC.
NUMERIC_SCALE	INT	Scale of a column whose data type is DECIMAL or NUMERIC.
NULLABLE	CHAR(3)	YES: Column can contain NULLs. NO: Column can't contain NULLs.
COL_DEFAULT	CHAR(254)	Default column value.
REMARKS	CHAR(254)	*Reserved for future use.*

UV_COLUMNS Columns (Continued)

TABLE_SCHEMA, TABLE_NAME, and COLUMN_NAME form the primary key.

UV_SCHEMA

The UV_SCHEMA table describes all the schemas in the database. It has the following columns:

Column Name	Data Type	Description
SCHEMA_NAME	CHAR(18)	Name of the schema.
OWNER	CHAR(18)	User ID number of the schema's owner.
PATH	CHAR(254)	Full path of the account directory where the schema resides.

UV_SCHEMA Columns

SCHEMA_NAME is the primary key.

UV_TABLES

The UV_TABLES table describes all the tables in the database. It has the following columns:

Column Name	Data Type	Description
TABLE_SCHEMA	CHAR(18)	Name of the schema where the table is located.
TABLE_NAME	CHAR(18)	Name of the table.
OWNER	INT	User ID number of the owner of the table.
TABLE_TYPE	CHAR(18)	One of the following: BASE TABLE: The table is nonderived. VIEW: The table is a view derived from one or more base tables or views. ASSOCIATION: The table is an association derived from corresponding multivalued columns in a base table. SYSTEM TABLE: The table is part of the SQL catalog.
BASE_TABLE	CHAR(18)	Name of the base table of an association.
COLUMNS	CHAR(18)	Names of the columns in the table. COLUMNS is a multivalued column.
VIEWS	CHAR(18)	Names of views derived from this table. VIEWS is a multivalued column.
PATH	CHAR(254)	Full path of the table.
DICT_PATH	CHAR(254)	Full path of the table dictionary.
ASSOCIATIONS	CHAR(18)	Names of the associations of multivalued columns in the table. ASSOCIATIONS is a multivalued column.
REMARKS	CHAR(254)	*Reserved for future use.*

UV_TABLES Columns

TABLE_SCHEMA and TABLE_NAME form the primary key.

UV_USERS

The UV_USERS table describes all UniVerse SQL users, the schemas and tables they own, and all their database and table privileges. It has the following columns:

Column Name	Data Type	Description
NAME	CHAR(18)	User name as defined by the operating system.
DBAUTH	CHAR(3)	YES: User has DBA database privilege. NO: User does not have DBA database privilege.
RESOURCEAUTH	CHAR(3)	YES: User has RESOURCE database privilege. NO: User does not have RESOURCE database privilege.
AUTHOR	CHAR(18)	User name of DBA who registered the user in the SQL catalog.
SCHEMAS	CHAR(18)	For each table in the TABLES column, the name of the schema containing the table. SCHEMAS is a multivalued column.
TABLES	CHAR(18)	Names of tables the user owns. TABLES is a multivalued column.
PERM_SCHEMAS	CHAR(18)	For each table in the PERM_TABLES column, the name of the schema containing the table. PERM_SCHEMAS is a multi-valued column.
PERM_TABLES	CHAR(18)	Names of tables for which the user has privi-leges. PERM_TABLES is a multivalued column.

UV_USERS Columns

NAME is the primary key. The table has two associations of multivalued columns: SCHEMAS and TABLES are associated, and PERM_SCHEMAS and PERM_TABLES are associated.

UV_VIEWS

The UV_VIEWS table describes all the views in the database. It has the following columns:

Column Name	Data Type	Description
VIEW_SCHEMA	CHAR(18)	Name of the schema where the view is located.
VIEW_NAME	CHAR(18)	Name of the view.
VIEW_TEXT	CHAR(254)	Query specification (SELECT statement) that creates the view.
TABLES	CHAR(254)	Names of the view's underlying tables and views. TABLES is a multivalued column.
COLUMN_MAP	CHAR(254)	If the view is updatable, maps the view's columns to columns in the underlying base tables and views.
IS_UPDATABLE	CHAR(3)	One of the following: NO: The view is read-only. yes: The view is updatable and does not contain all base table primary keys. YES: The view is updatable and contains all base table primary keys.
CHECK_OPTION	CHAR(8)	One of the following: NO: CHECK OPTION is not set. LOCAL: Only the WHERE clause of the view is checked. CASCADED: The WHERE clause of the view and all underlying views are checked.

UV_VIEWS Columns

VIEW_SCHEMA and VIEW_NAME form the primary key.

Using the SQL Catalog

You can retrieve data from the SQL catalog tables just as you can from any other table you have access to. For example:

```
>SELECT TABLE_NAME, TABLE_TYPE FROM UV_TABLES
SQL+WHERE TABLE_SCHEMA = 'MYSCHEMA'
SQL+ORDER BY TABLE_NAME;
```

All users can read all tables in the catalog. No user, not even a database administrator (DBA), can directly add, change, or delete anything in the catalog.

Finding SQL Catalog Inconsistencies

UniVerse SQL users can use the VERIFY.SQL command in UniVerse to examine the SQL catalog for inconsistencies.

VERIFY.SQL is a diagnostic tool that you should use if you suspect information in the SQL catalog is inconsistent with the schemas, tables, views, directories, and files on your system. Such inconsistencies should not occur during normal use, but they can happen when you delete, rename, or move files at the operating system level.

VERIFY.SQL compares data in the security and integrity constraints areas (SICAs) of tables and views, to data in the SQL catalog and displays any inconsistencies.

UniVerse SQL users can verify only tables and views they have operating system permissions and SQL privileges to access. DBAs can verify all tables and views on the system, provided they have proper file and directory permissions.

For example, here are two ways to verify the ORDERS table in the current schema. The first command specifies the table by name, the second specifies it by path.

```
>VERIFY.SQL TABLE ORDERS
>VERIFY.SQL TABLE /usr/accounts/ORDERS
```

The next example shows two ways to verify the Sales schema:

```
>VERIFY.SQL SCHEMA Sales
>VERIFY.SQL SCHEMA /usr/sales
```

The next example verifies the owners, paths, and schema names of all schemas on the system. This command can take a long time to execute, because it looks at all directories on the system.

```
>VERIFY.SQL SCHEMAS
```

The next example verifies the internal consistency of the SQL catalog:

```
>VERIFY.SQL CATALOG
```

The next example verifies all SQL objects on the system, including the internal consistency of the SQL catalog. This command can also take a long time to execute.

```
>VERIFY.SQL ALL
```

Fixing SQL Catalog Inconsistencies

DBAs can use the VERIFY.SQL command to fix inconsistencies between UniVerse SQL objects and the SQL catalog. VERIFY.SQL also fixes internal inconsistencies within the SQL catalog.

The FIX keyword changes the data in the SQL catalog to make it agree with data in the schemas' VOC files and in the SICAs of their tables and views. If data is found in the SQL catalog for a schema, table, or view that does not exist, VERIFY.SQL deletes the data in the SQL catalog. If SQL catalog data is internally inconsistent, VERIFY.SQL changes the data to make it agree with the data found in the SQL objects it is currently verifying. If there is no corresponding data in the SQL catalog, VERIFY.SQL fixes the inconsistencies as follows:

Command	Description
VERIFY.SQL SCHEMA *pathname* FIX VERIFY.SQL SCHEMA FIX	Creates the SQL catalog data using information in the schema's VOC file and in the SICAs of the schema's tables.
VERIFY.SQL TABLE *pathname* FIX	Creates the SQL catalog data using information in the SICA of the table.
VERIFY.SQL VIEW *pathname* FIX	Creates the SQL catalog data using information in the SICA of the view.

VERIFY.SQL Commands

Command	Description
VERIFY.SQL SCHEMA *schema* FIX	Changes no SQL catalog data and produces an error message. VERIFY.SQL cannot locate a schema by name if the name is not in the SQL catalog.
VERIFY.SQL TABLE *table* FIX	Changes no SQL catalog data and produces an error message. VERIFY.SQL cannot locate a table by name if the name is not in the SQL catalog.
VERIFY.SQL ALL FIX	Creates the SQL catalog data using information in all the schemas’ VOC files and in the SICAs of their tables. If a table is not in a valid schema, no SQL catalog data is changed and an error message appears.

VERIFY.SQL Commands (Continued)

Chapter

Data Types

3

This chapter describes the six UniVerse SQL data categories and the eleven SQL data types. Every column in a table has a data type that defines the kind of values the column contains. Data types fall into six general data categories.

UniVerse SQL Data Categories

UniVerse SQL recognizes seven data categories. All data, and literals in an SQL query, fall into one of these categories. The data category to which a value belongs determines how an SQL query processes the data. Some categories of data can be added, subtracted, compared, and so on, and some cannot. For example, you cannot add integers to character strings. The following table describes the seven data categories.

Category	Description
Integer	Positive or negative whole numbers such as 0, 5, +03, and 6758948398458.
Scaled number	Positive or negative numbers with fixed-length fractional parts, such as 2.00, 1999.95, and –0.75. These are also known as exact numbers.
Approximate number	Arbitrary real numbers that can include fractional parts of unknown length. These numbers may need to be rounded off to fit the computer's limits for storing significant digits. Examples are Avogadro's number (6.023E23) and pi (3.14159…).
Date	Dates are stored internally as the number of days since December 31, 1967. Dates are output in conventional date formats such as 2/28/92 or 31 Jan 1990. A conversion code converts the internal date to a conventional format.
Time	Times are stored internally as a number of seconds, which can represent either a time of day (number of seconds after midnight) or a time interval. They are output in conventional time formats such as 12:30 PM or 02:23:46. A conversion code converts the internal time to a conventional format.
Character string	Any mixture of numbers, letters, and special characters.
Bit string	Any arbitrary sequence of bits.

Data Categories

UniVerse SQL allows the following operations and comparisons among data categories, which the following sections cover:

- You can use any arithmetic operator (+, –, *, or /) on any pair of numbers (integer, scaled, or approximate). The result is an approximate number unless you add, subtract, or multiply two integers, in which case the result is also an integer.
- You can subtract a date from a date. The result is an integer representing days. You cannot add a date to a date.
- You can add and subtract times. The result is a time.
- You can add an integer to a date, and you can subtract an integer from a date. The result is also a date, *n* days later or earlier than the given date.
- You can add an integer to a time, and you can subtract an integer from a time. The result is also a time, *n* hours later or earlier than the given time.
- You can compare any integer, scaled number, and approximate number to any other integer, scaled number, or approximate number. You can compare dates to dates. You can compare times to times. And you can compare character strings to character strings.
- You cannot compare an integer to a time or a time to an integer.
- You can compare a bit string only to another bit string.
- You can use the MAX, MIN, and COUNT set functions with all data categories. You can use the SUM and AVG set functions with all data categories except character strings.

SQL Data Types

UniVerse SQL recognizes fifteen data types, as shown in the following table. The following sections describe each data type in detail.

Data Type	Description
BIT	Bit strings
CHAR CHARACTER	Character strings
DATE	Dates
DEC DECIMAL	Decimal fixed-point numbers
DOUBLE PRECISION	High-precision floating-point numbers
FLOAT	Floating-point numbers
INT INTEGER	Whole numbers
NCHAR NATIONAL CHAR NATIONAL CHARACTER	National character strings
NUMERIC	Decimal fixed-point numbers
NVARCHAR NCHAR VARYING NATIONAL CHAR VARYING NATIONAL CHARACTER VARYING	Variable-length national character strings
REAL	Floating-point numbers
SMALLINT	Small whole numbers

Data Types

Data Type	Description
TIME	Times
VARBIT BIT VARYING	Variable-length bit strings
VARCHAR CHAR VARYING CHARACTER VARYING	Variable-length character strings

Data Types (Continued)

BIT

The BIT data type stores bit strings of up to 2032 bits. When you specify a column's data type in a CREATE TABLE or ALTER TABLE statement, the syntax is as follows:

BIT [(*n*)]

n is an integer from 1 through 2032 that specifies the length of the column. If you do not specify *n*, the default length is 1 bit.

If you specify a conversion code when you define the column, it must be BB or BX; any other conversion code results in an error. By default, UniVerse SQL does not generate a conversion code. With a BB conversion, BIT data is displayed in binary format. With a BX conversion, BIT data is displayed in hexadecimal format. With no conversion code, BIT data is not displayed; instead, the phrase `<bit string>` is displayed.

If you do not specify a format code when you define the column, UniVerse SQL generates a FMT code of 1L if *n* is not specified; otherwise UniVerse SQL generates a FMT code of 10L.

CHAR

The CHAR data type stores character strings, which can be any combination of numbers, letters, and special characters. It is also called the CHARACTER data type. When you specify a column's data type in a CREATE TABLE or ALTER TABLE statement, the syntax is as follows:

```
CHAR [ (n) ]
CHARACTER [ (n) ]
```

n is an integer from 1 through 254 that specifies the length of the column. If you do not specify *n*, the default length is 1.

You use the CHAR data type to store names, addresses, phone numbers, zip codes, descriptions, and so on.

If you do not specify a format code when you define the column, UniVerse SQL generates a FMT code of *n*L, which specifies a column length of *n* and left justification.

DATE

The DATE data type stores dates. If you do not specify a conversion code when you define the column, UniVerse SQL generates a CONV code of D. If you specify a conversion code, it must begin with D; any other conversion code results in an error. If you do not specify a format code, UniVerse SQL generates a FMT code consistent with the conversion code.

DEC

The DEC data type stores decimal fixed-point numbers. It is also called the DECIMAL data type. When you specify a column's data type in a CREATE TABLE or ALTER TABLE statement, the syntax is as follows:

```
DEC [ (p [ ,s ] ) ]
DECIMAL [ (p [ ,s ] ) ]
```

p (precision) is the number of significant digits, which is the total number of digits to the left and the right of the decimal point. If you do not specify *p*, the default precision is 9.

s (scale) is the number of digits to the right of the decimal point. Scale can be from 0 through 9. If you do not specify *s*, the default scale is 0.

You use the DEC data type for numbers with fractional parts that must be calculated exactly, such as money or percentages.

If you do not specify a conversion code when you define the column, UniVerse SQL generates a CONV code of MD*ss*, where *s* is the scale. If you do not specify a format code, UniVerse SQL generates a FMT code of (*p*+1)R, where *p* is the precision and R specifies right justification.

Note*: When the scaling factor of an MD, ML, or MR conversion is not 0, the internal form of the data is numerically different from the external form. Therefore, if you specify one of these conversions for a column whose type is DEC, you should define the scaling factor of the conversion to be the same as scale (s), otherwise results will be unpredictable.*

In UniVerse SQL the DEC and NUMERIC data types are the same.

DOUBLE PRECISION

The DOUBLE PRECISION data type stores high-precision floating-point numbers. When you specify a column's data type in a CREATE TABLE or ALTER TABLE statement, the syntax is as follows:

```
DOUBLE PRECISION
```

You use the DOUBLE PRECISION data type for scientific numbers that can be calculated only approximately.

If you do not specify a conversion code when you define the column, UniVerse SQL generates a CONV code of QX. If you do not specify a format code, UniVerse SQL generates a FMT code of 30R, which specifies a column length of 30 and right justification.

FLOAT

The FLOAT data type stores floating-point numbers. When you specify a column's data type in a CREATE TABLE or ALTER TABLE statement, the syntax is as follows:

FLOAT [(*p*)]

p (precision) is the number of significant digits. If you do not specify *p*, the default precision is 15.

You use the FLOAT data type for scientific numbers that can be calculated only approximately.

If you do not specify a conversion code when you define the column, UniVerse SQL generates a CONV code of QX. If you do not specify a format code, UniVerse SQL generates a FMT code of (*p*+1)R, where *p* is the precision and R specifies right justification.

INT

The INT data type stores whole numbers. It is also called the INTEGER data type. When you specify a column's data type in a CREATE TABLE or ALTER TABLE statement, the syntax is as follows:

```
INT
INTEGER
```

You use the INT data type for counts, quantities, and so on. You can also use this data type for dates or times, provided you also specify an appropriate D conversion code (for dates) or MT conversion code (for times).

If you do not specify a conversion code when you define the column, UniVerse SQL generates a CONV code of MD0. If you do not specify a format code, UniVerse SQL generates a FMT code of 10R, which specifies a column length of 10 and right justification.

In UniVerse SQL the INT and SMALLINT data types are the same.

NCHAR

The NCHAR data type stores national character strings, which can be any combination of numbers, letters, and special characters. It is also called the NATIONAL CHARACTER data type. When you specify a column's data type in a CREATE TABLE or ALTER TABLE statement, the syntax is as follows:

```
NCHAR [(n)]
NATIONAL CHAR [(n)]
NATIONAL CHARACTER [(n)]
```

n is an integer from 1 through 254 that specifies the length of the column. If you do not specify *n*, the default length is 1.

You use the NCHAR data type to store names, addresses, phone numbers, zip codes, descriptions, and so on.

If you do not specify a format code when you define the column, UniVerse SQL generates a FMT code of *n*L, which specifies a column length of *n* and left justification.

To use NCHAR columns, NLS must be enabled.

NUMERIC

The NUMERIC data type stores decimal fixed-point numbers. When you specify a column's data type in a CREATE TABLE or ALTER TABLE statement, the syntax is as follows:

NUMERIC [(*p* [,*s*])]

p (precision) is the number of significant digits, which is the total number of digits to the left and the right of the decimal point. If you do not specify *p*, the default precision is 9.

s (scale) is the number of digits to the right of the decimal point. Scale can be from 0 through 9. If you do not specify *s*, the default scale is 0.

You use the NUMERIC data type for numbers with fractional parts that must be calculated exactly, such as money or percentages.

If you do not specify a conversion code when you define the column, UniVerse SQL generates a CONV code of MD*ss*, where *s* is the scale. If you do not specify a format code, UniVerse SQL generates a FMT code of (*p*+1)R, where *p* is the precision and R specifies right justification.

***Note**: When the scaling factor of an MD, ML, or MR conversion is not 0, the internal form of the data is numerically different from the external form. Therefore, if you specify one of these conversions for a column whose type is NUMERIC, you should define the scaling factor of the conversion to be the same as scale (s), otherwise results will be unpredictable.*

In UniVerse SQL the NUMERIC and DEC data types are the same.

NVARCHAR

The NVARCHAR data type stores variable-length character strings, which can be any combination of numbers, letters, and special characters. It is also called the NCHAR VARYING or NATIONAL CHARACTER VARYING data type. When you specify a column's data type in a CREATE TABLE or ALTER TABLE statement, the syntax is as follows:

```
NVARCHAR [(n)]
NCHAR VARYING [(n)]
NATIONAL CHAR VARYING [(n)]
NATIONAL CHARACTER VARYING [(n)]
```

n is an integer from 1 through 65535 that specifies the length of the column. If you do not specify *n*, the default length is 254.

You use the NVARCHAR data type to store names, addresses, phone numbers, zip codes, descriptions, and so on.

If you do not specify a format code when you define the column, UniVerse SQL generates a FMT code of 10T, which specifies a column length of 10 and text justification.

To use NVARCHAR columns, NLS must be enabled.

REAL

The REAL data type stores floating-point numbers. When you specify a column's data type in a CREATE TABLE or ALTER TABLE statement, the syntax is as follows:

```
REAL
```

You use the REAL data type for scientific numbers that can be calculated only approximately.

If you do not specify a conversion code when you define the column, UniVerse SQL generates a CONV code of QX. If you do not specify a format code, UniVerse SQL generates a FMT code of 10R, which specifies a column length of 10 and right justification.

SMALLINT

The SMALLINT data type stores small whole numbers. When you specify a column's data type in a CREATE TABLE or ALTER TABLE statement, the syntax is as follows:

```
SMALLINT
```

You use the SMALLINT data type for counts, quantities, and so on. You can also use this data type for dates or times, provided you also specify an appropriate D conversion code (for dates) or MT conversion code (for times).

If you do not specify a conversion code when you define the column, UniVerse SQL generates a CONV code of MD0. If you do not specify a format code, UniVerse SQL generates a FMT code of 10R, which specifies a column length of 10 and right justification.

In UniVerse SQL the SMALLINT and INT data types are the same.

TIME

The TIME data type stores times. If you do not specify a conversion code when you define the column, UniVerse SQL generates a CONV code of MTS. If you specify a conversion code, it must begin with MT; any other conversion code results in an error. If you do not specify a format code, UniVerse SQL generates a FMT code consistent with the conversion code.

VARBIT

The VARBIT data type stores bit strings of up to 524,280 bits. When you specify a column's data type in a CREATE TABLE or ALTER TABLE statement, the syntax is as follows:

```
VARBIT [(n)]
BIT VARYING [(n)]
```

n is an integer from 1 through 524,280 that specifies the length of the column. If you do not specify *n*, the default length is 2032 bits.

If you specify a conversion code when you define the column, it must be BB or BX; any other conversion code results in an error. By default, UniVerse SQL does not generate a conversion code. With a BB conversion, VARBIT data is displayed in binary format. With a BX conversion, VARBIT data is displayed in hexadecimal format. With no conversion code, VARBIT data is not displayed; instead, the phrase `<bit string>` is displayed.

If you do not specify a format code when you define the column, UniVerse SQL generates a FMT code of 10L.

VARCHAR

The VARCHAR data type stores variable-length character strings, which can be any combination of numbers, letters, and special characters. It is also called the CHAR VARYING or CHARACTER VARYING data type. When you specify a column's data type in a CREATE TABLE or ALTER TABLE statement, the syntax is as follows:

> VARCHAR [(*n*)]
> CHAR VARYING [(*n*)]
> CHARACTER VARYING [(*n*)]

n is an integer from 1 through 65535 that specifies the length of the column. If you do not specify *n*, the default length is 254.

You use the VARCHAR data type to store names, addresses, phone numbers, zip codes, descriptions, and so on.

If you do not specify a format code when you define the column, UniVerse SQL generates a FMT code of 10T, which specifies a column length of 10 and text justification.

Data Types and Data Categories

When you define a column in a table, you specify its data type. The data type determines what category of data the column stores. The following table summarizes UniVerse SQL data categories, data types, and the conversion and format codes that UniVerse SQL generates when it creates a table's dictionary entries.

Data Category	**Data Type**	**Generated Conversion Code**	**Generated Format Code**
Integer	INT	MD0	10R
	SMALLINT	MD0	10R
	DECIMAL (*p*,0)	MD00	(*p*+1)R
	NUMERIC (*p*,0)	MD00	(*p*+1)R
Scaled number	DECIMAL (*p,s*)	MD*ss*	(*p*+1)R
	NUMERIC (*p,s*)	MD*ss*	(*p*+1)R
Approximate number	REAL	QX	10R
	FLOAT (*p*)	QX	(*p*+1)R
	DOUBLE PRECISION	QX	30R
Date	DATE	D	11R
Time	TIME	MTS	8R
Character string	CHAR (*n*)		*n*L
	VARCHAR (*n*)		10T
	NCHAR (*n*)		*n*L
	NVARCHAR (*n*)		10T
Bit string	BIT (*n*)	BB or BX	1L or 10L
	VARBIT (*n*)	BX	10L

Data Types and Data Categories

Scale indicates the number of digits to the right of the decimal point. When you define a DECIMAL or NUMERIC column, a scale of *s* (an integer from 1 through *p*) generates an MD*ss* conversion. UniVerse uses precision (*p*) when generating the format field of the dictionary.

In the UniVerse environment, CHAR, VARCHAR, NCHAR, and NVARCHAR (character strings) are the same. DOUBLE PRECISION, FLOAT (floating-point numbers), and REAL are the same. INT and SMALLINT (integers) are the same. NUMERIC and DECIMAL (fixed-scale numbers) are the same. BIT and VARBIT (bit strings) are the same.

If you specify a conversion code for a DATE or TIME column in the CREATE TABLE statement, the generated format code will be consistent with the code you specify. For example, D2- gives 8R, and MTHS gives 10R. If you also specify a format code in the CREATE TABLE statement, it overrides the generated format code.

Dates and times can also be stored in columns defined as INT or SMALLINT. If you define date and time columns this way, you must specify a valid D conversion code for date columns and a valid MT conversion code for time columns.

Data Types and Conversion Codes

When you define a column with the CREATE TABLE or ALTER TABLE statement, you can specify a conversion code along with the data type. If you do not specify a conversion code, UniVerse SQL generates one, except for character strings. For example, if you specify the data type INT for a column, UniVerse SQL supplies the conversion code MD0.

The conversion code you specify must be compatible with the data type or results will be unpredictable. Specifically, you should do the following:

- Use MD, ML, and MR conversions having a nonzero scaling factor only for DECIMAL and NUMERIC columns. Be sure the scale specified in the conversion agrees with the scale implied by the data type.
- Use D conversions only for DATE, INT, or SMALLINT columns.
- Use MT conversions only for TIME, INT, or SMALLINT columns.
- Use BB and BX conversions only for BIT and VARBIT columns.

The following table definition has four columns defined to contain CHAR, REAL, DATE, and TIME data types. The DATE and TIME columns also include date and time conversion codes.

```
CREATE TABLE TEMPS
   (CITY CHAR(20),
   TEMPERATURE REAL,
   "DATE" DATE CONV 'D2-',
   "TIME" TIME CONV 'MTHS',
   ...);
```

The column names DATE and TIME must be enclosed in double quotation marks because they are UniVerse SQL reserved words. For information about reserved words, see Delimited Identifiers in Chapter 6, "UniVerse SQL Statements," and Appendix B, "Reserved Words."

The following table definition defines a monetary field, PRICE. The data type specification and the conversion code both specify the same scaling factor, 2.

```
CREATE TABLE INVENTORY
   (PRODNO INT NOT NULL PRIMARY KEY,
   .
   .
   .
   PRICE DEC(6,2) CONV 'MD2$'
   .
   .
   .
```

Chapter

UniVerse SQL in Client Programs

4

Other chapters of this book treat UniVerse SQL as an interactive language—that is, a language you use to enter SQL statements at the system prompt, the results from which are displayed on your screen. This chapter describes how to use UniVerse SQL in client programs that access a UniVerse server. The version of UniVerse SQL you use in client programs is called *programmatic SQL*.

Programmatic SQL and interactive SQL are mostly the same. This chapter describes the differences between interactive SQL and programmatic SQL. Programmatic SQL returns output to program variables instead of to a terminal screen.

UniVerse provides two application program interfaces (APIs) you can use to write client programs:

- UniVerse Call Interface (UCI)
- BASIC SQL Client Interface

These APIs are based on the Microsoft Open Database Connectivity (ODBC) interface.

UCI is a C-language API, and the BASIC SQL Client Interface is a UniVerse BASIC API. Both let application programmers write client programs that use SQL function calls to access data in UniVerse databases. For information about UCI, see *UCI Developer's Guide*. For information about the BASIC SQL Client Interface, see *UniVerse BASIC SQL Client Interface Guide*. For a general background on programming with ODBC, see *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*.

Programmatic SQL includes only UniVerse SQL statements; it does not include any of the following:

- UniVerse commands, such as LIST, SORT, and CREATE.INDEX
- UniVerse commands that relate directly to UniVerse SQL, such as CONNECT, CONVERT.SQL, LIST.SICA, and VERIFY.SQL
- UniVerse BASIC transaction statements, such as BEGIN TRANSACTION, COMMIT, and ROLLBACK

Programming with ODBC

Programs that use ODBC do the following:

- Set up an execution environment
- Connect to a database server (data source)
- Allocate variables to contain column contents or parameter values
- Execute SQL statements
- Execute stored procedures
- Fetch result rows
- Define data conversions
- Receive status and error information
- Control transactions

These functions are fully described in the *UCI Developer's Guide* and the *UniVerse BASIC SQL Client Interface Guide*.

Each client program using programmatic SQL has its own user environment on the UniVerse server, defined by a user name (login name) and a UniVerse account directory. The user name and account directory are established when the client program connects to the server. This environment affects programmatic SQL statements in the same way that an interactive user's environment affects interactive SQL statements. For example, the client's user name is used to verify SQL table privileges, and the account directory determines which VOC file to use as the source of file names.

Multivalued Columns and Associations

Because SQL Client Interface functions are part of UniVerse BASIC, the SQL statements they use treat multivalued data on UniVerse data sources in the same way UniVerse BASIC programs or UniVerse SQL statements do. SQL statements in UCI client programs, on the other hand, can use a special first-normal-form mode when they access tables and UniVerse files that contain multivalued columns. For detailed information about how UCI handles multivalued data in UniVerse tables and files, see the *UCI Developer's Guide*.

Using SQL Statements in Programs

SQL statements come in two categories: those that define data, and those that manipulate data. Programs can also use the CALL statement to invoke procedures stored on the server. You can use the following UniVerse SQL statements in UCI and BASIC SQL Client Interface programs:

Data Definition Language (DDL)	**Data Manipulation Language (DML)**	**Procedure Calls**
ALTER TABLE	DELETE	CALL
CREATE INDEX	INSERT	
CREATE SCHEMA	SELECT	
CREATE TABLE	UPDATE	
CREATE TRIGGER		
CREATE VIEW		
DROP INDEX		
DROP SCHEMA		
DROP TABLE		
DROP TRIGGER		
DROP VIEW		
GRANT		
REVOKE		

UniVerse SQL Statements for UCI and BCI

You can use DML statements to query and manipulate data in both of the following:

- SQL tables
- UniVerse files that are not SQL tables

You can use DML statements in transactions. Since DML statements are transactional (that is, each DML statement is automatically committed as a separate transaction), they become nested transactions when used in a transaction (for information about nested transactions, see *UniVerse BASIC*).

Note*: You cannot use DDL statements in transactions.*

SQL Syntax in Programs

You can use programmatic SQL statements in client programs or in programs that are stored on a UniVerse server. The latter are *procedures*, stored on the server, that can be called by client programs. Procedures can provide a significant performance improvement in a client/server environment. Applications often have many steps, where the result from one step becomes the input for the next. If you run such an application from a client, it can take a lot of network traffic to perform each step and get results from the server. If you run the same program as a procedure, all the intermediate work occurs on the server; the client simply calls the procedure and receives a result. For information about creating and executing procedures, see the *UniVerse BASIC SQL Client Interface Guide* or the *UCI Developer's Guide*.

A programmatic SQL statement must be only one statement, and it must be complete. It cannot be a partial SQL statement or a group of several SQL statements. You can include or omit the final ; (semicolon) in programmatic SQL. You cannot include comments in programmatic SQL statements.

Reserved words in programmatic SQL statements are not case-sensitive. You can code all SQL statement names (such as CREATE TABLE and INSERT), SQL keywords (such as INTEGER and GROUP BY), and UniVerse-specific keywords (such as ROWUNIQUE and UNNEST) in uppercase or lowercase letters. However, SQL identifiers are case-sensitive. You must code identifiers such as table and column names to match the format of the identifier as originally defined.

SELECT Statements in Programmatic SQL

In programmatic SQL the client program retrieves output from an SQL SELECT or CALL statement one row at a time by issuing **SQLFetch** calls. Each row is returned as a set of column values, and each column value is stored in its own program variable. Since data returned to a client program is not meant to be formatted for screen or printer output, certain display formatting codes are not allowed in programmatic SQL.

Programmatic SQL supports all standard SQL keywords such as WHERE, HAVING, LIKE, and so forth. The following table lists all UniVerse-specific SQL keywords and indicates which ones you can use in programmatic SQL.

Syntax Element	**Allowed**	**Not Allowed**
SELECT Clause	TO SLIST	
Table reference	Explicit primary keys	INQUIRING
	UNNEST clause	SLIST 0
	DICT	
	DATA	
	SLIST[a]	
	USING DICT	
	NO.INDEX	
Field modifiers		AVG
		BREAK ON
		BREAK SUPPRESS
		CALC
		PCT
		TOTAL

UniVerse Keywords in Programmatic SQL

Syntax Element	Allowed	Not Allowed
Field qualifiers	AS	ASSOC
	DISPLAYLIKE	ASSOCIATED
	DISPLAYNAME	MULTIVALUED
	CONV	SINGLEVALUED
	FMT	
Report qualifiers		COLUMN SPACES
		COUNT.SUP
		DOUBLE SPACE
		FOOTING
		GRAND TOTAL
		HEADING
		LPTR
		MARGIN
		NOPAGE
		SUPPRESS COLUMN HEADING
		SUPPRESS DETAIL
		VERTICALLY
Processing qualifiers	EXPLAIN	REPORTING
	NO.OPTIMIZE	
	NOWAIT	
	SAMPLE	
	SAMPLED	

UniVerse Keywords in Programmatic SQL (Continued)

a. Except SLIST 0.

CONV and FMT have no effect on data fetched to your variables.

When used in client programs, EXPLAIN only returns an explanation. It does not execute the DML statement.

Field Qualifiers in Programmatic SQL

When you use field qualifiers in programmatic SQL, the settings they specify for a column are available to the **SQLColAttributes** function, as follows:

Field Qualifier Value	Stores in This Column Attribute
CONV	SQL.COLUMN.CONVERSION SQL_COLUMN_CONVERSION
DISPLAYNAME	SQL.COLUMN.LABEL SQL_COLUMN_LABEL
FMT (column width)	SQL.COLUMN.DISPLAY.SIZE SQL_COLUMN_DISPLAY_SIZE
FMT	SQL.COLUMN.FORMAT SQL_COLUMN_FORMAT

SQLColAttributes Function

Column attribute names with periods are used with the BASIC SQL Client Interface, names with underscores are used with UCI.

The DISPLAYLIKE field qualifier causes **SQLColAttributes** to refer to another column for the values stored in the column attributes listed in the previous table.

Using Parameter Markers in DML Statements

You can use parameter markers in SQL statements to mark where to insert values to send to the data source. Programmatic SQL uses a ? (question mark) as a parameter marker.

Typically, you use parameter markers when an SQL statement must be issued repeatedly with different values. For example, you can use a single-row INSERT statement repeatedly to load rows with different values into a table. But you can also use parameter markers in nonrepeated DML statements.

***Note**: You cannot use parameter markers in DDL statements.*

You can use parameter markers almost anywhere you can use literal values in any DML statement (SELECT, INSERT, UPDATE, and DELETE) and in CALL statements. However, you cannot use parameter markers to mark the place of the following literals:

- A character string literal after the DISPLAYNAME, FMT, CONV, and EVAL keywords
- A numeric literal after the SAMPLE and SAMPLED keywords
- A column specification in a SELECT clause
- A column or select expression in a set function
- An explicit record ID in a table expression
- A column number in an ORDER BY clause
- Both expressions in a comparison predicate
- Expressions on both sides of an arithmetic operator (+, –, *, /)
- Both the first expression in a BETWEEN comparison and either of the expressions following the BETWEEN keyword
- Both the first expression in an IN comparison and the first value following the IN keyword
- The operand of a unary plus or minus

***Note**: You can supply only literal values as parameters. You cannot supply column names, expressions, set functions, or keywords such as USER, DEFAULT, or NULL.*

For more information about using parameter markers, see the *UniVerse BASIC SQL Client Interface Guide* and the *UCI Developer's Guide*.

Chapter

5 Triggers

This chapter describes triggers, which in UniVerse SQL are UniVerse BASIC programs associated with a table. They are executed (“fired”) when some action changes the table’s data.

Applying Business Rules

Developers use triggers to enforce certain business rules when users or programs make changes to a database. Traditional database systems require that applications enforce their own business rules. When many applications reference the same tables, they need duplicate code to enforce business rules for those tables, making it difficult to maintain consistency. Databases that use triggers, on the other hand, can enforce business rules directly. When the database enforces business rules, it enforces them consistently, and you need to maintain only one single code source.

Some business rules that application programs might enforce are:

- A customer number on an order must correspond to an existing customer.
- The program cannot delete customers if they have any orders.
- When a customer's number changes, the program updates all of that customer's orders.

In UniVerse SQL it is more efficient to use referential integrity instead of triggers to enforce such rules. On the other hand, triggers can enforce more precise rules such as the following:

- A customer number on an order must correspond to an *active* customer.
- The program cannot delete customers if they have any *open* orders.
- Certain changes are not allowed, depending on date, time, user, or specific data.
- The program inserts date, time, or user stamps.
- The program maintains audit copies of changes in another table.

Using Triggers

SQL statements, BASIC programs, ProVerb procs, and UniVerse paragraphs can all change a table's data and thus fire a trigger program. Events that change the database include:

- INSERT statement
- UPDATE statement
- DELETE statement
- BASIC WRITE statements
- BASIC DELETE statement
- ProVerb F-WRITE command

Making changes to the database using the UniVerse Editor and ReVise also fire triggers. Other UniVerse and operating system commands and operations, such as CLEAR.FILE, *rm*, or a roll-forward, do not.

When Does a Trigger Fire?

A trigger can fire either before or after a change is made to the database. When you create a trigger, you specify whether it should fire before the event that changes the database or after it. A trigger program fires (is executed) for each row that is inserted, updated, or deleted.

Note: *UniVerse SQL supports only triggers that fire once for each row. It does not support triggers that fire once for each statement.*

What Events Fire a Trigger?

All events that change the database are treated as one of the following:

- INSERT statement
- UPDATE statement
- DELETE statement

A BASIC WRITE statement is treated as an INSERT statement if the record ID does not exist. It is treated as an UPDATE statement if the record ID already exists. To change a primary key using BASIC, you must first DELETE the old record (firing any DELETE triggers), then WRITE a new record (firing any INSERT triggers).

If you use SQL to change a primary key, any BEFORE UPDATE triggers fire during the automatic DELETE, and any AFTER UPDATE triggers fire during the automatic WRITE.

If a change made to an updatable view affects the base table, the change fires the base table's triggers.

A change made to a dynamically normalized association fires any UPDATE trigger on the underlying table. If you change the primary key while the association is dynamically normalized, the UPDATE trigger is executed twice, first for the old key, then for the new key.

A delete operation on a table referenced by another table causes the action specified by the ON DELETE clause of the referencing table to occur. If that action is a delete operation, any DELETE triggers fire on the referencing table; if the action is an update operation (such as SET NULL or SET DEFAULT), any UPDATE triggers fire.

Creating Triggers

You create triggers for a table using the CREATE TRIGGER statement. You delete them using the DROP TRIGGER statement. You can also disable and reenable a table's triggers using the ALTER TABLE statement. Triggers are enabled by default when you first create them.

You can define triggers only for tables, not for associations, views, or UniVerse files that are not tables. You can include trigger definitions as part of the initial database definition, or you can add them to tables later.

You must be the table's owner, or have ALTER Privilege on the table, or be a DBA to create a trigger.

Modifying Triggers

If you want to modify a trigger program, do the following:

- Use the DROP TRIGGER statement to drop all triggers that use the program.
- Change, compile, and catalog the trigger program.
- Use the CREATE TRIGGER statement to recreate the trigger.

Listing Information About Triggers

Use the LIST.SICA command to list information about a table's triggers. LIST.SICA lists the following:

- Names of triggers
- Names of the BASIC trigger programs they invoke
- Whether the triggers are enabled or disabled

The following example assumes trigger programs TRIG1 and TRIG2 have been globally cataloged before executing the CREATE TRIGGER statements:

```
>CREATE TABLE TBL4003 (C1 INT NOT NULL PRIMARY KEY, C2 INT);
Creating Table "TBL4003"
Adding Column "C1"
Adding Column "C2"
>CREATE TRIGGER TRIG1
SQL+BEFORE DELETE ON TBL4003 FOR EACH ROW CALLING '*TRIG1';
Adding trigger "TRIG1"
>CREATE TRIGGER TRIG2
SQL+AFTER DELETE ON TBL4003 FOR EACH ROW CALLING '*TRIG1';
Adding trigger "TRIG2"
>CREATE TRIGGER TRIG3
SQL+AFTER INSERT ON TBL4003 FOR EACH ROW CALLING '*TRIG1';
Adding trigger "TRIG3"
>CREATE TRIGGER TRIG4
SQL+BEFORE INSERT OR UPDATE ON TBL4003
SQL+FOR EACH ROW CALLING '*TRIG2';
Adding trigger "TRIG4"
>LIST.SICA TBL4003

LIST.SICA TBL4003 11:37:24am  21 May 1997   Page    1
==========================================
Sica Region for Table "TBL4003"

  Schema:           TESTTRG
  Revision:         3
  Checksum is:      8429
    Should Be:      8429
  Size:             388
  Creator:          0
  Total Col Count:  2
    Key Columns:    1
    Data Columns:   1
  Check Count:      0
  Permission Count:0
  History Count:    0

  Data for Column "C1"
  Position:      0
  Key Position:  1
  Multivalued:   No
  Not Null:      constraint UVCON_0 Yes
  Not Empty:     No
  Unique:        No
  Row Unique:    No
  Primary Key:   Yes
  Default Type:  None
  Data Type:     INTEGER
  Conversion:    MD0
  Format:        10R
  No Default Value Defined
  No association defined

Data for Column "C2"

  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:     INTEGER
  Conversion:    MD0
  Format:        10R
  No Default Value Defined
  No association defined

Trigger "TRIG4" is enabled, creator is "VMARK\csm".
        calls "*TRIG2" for
        Row Before Insert Update
Trigger "TRIG3" is enabled, creator is "VMARK\csm".
        calls "*TRIG1" for
        Row After Insert
Trigger "TRIG2" is enabled, creator is "VMARK\csm".
        calls "*TRIG1" for
        Row After Delete
Trigger "TRIG1" is enabled, creator is "VMARK\csm".
        calls "*TRIG1" for
        Row Before Delete
```

Trigger Programs

Trigger programs are compiled and cataloged BASIC subroutines. You must catalog such programs either normally or globally. For information about cataloging BASIC programs, see *UniVerse BASIC*.

Each BASIC subroutine must define 14 arguments in the following order:

Argument	Contains...
trigger.name	Name of the trigger
schema	Name of the schema containing the trigger's table
table	Name of the trigger's table
event	INSERT, UPDATE, or DELETE
time	BEFORE or AFTER
new.recordID	If *event* is INSERT or UPDATE, new record ID, otherwise empty
new.record	If *event* is INSERT or UPDATE, new record, otherwise empty
old.recordID	If *event* is UPDATE or DELETE, old record ID, otherwise empty
old.record	If *event* is UPDATE or DELETE, old record, otherwise empty
association	Name of a dynamically normalized association
association.event	INSERT, UPDATE, or DELETE on a dynamically normalized association
count	Number of triggers that are currently firing
chain.cascade	Number of active triggers since the last cascade operation
cascade	UPDATE or DELETE if a cascaded update or delete fires the trigger, otherwise empty

Trigger Subroutine Arguments

The position of the argument determines what information it contains. For the syntax of the SUBROUTINE statement, see *UniVerse BASIC*.

BASIC programs can use the *new.recordID*, *new.record*, *old.recordID*, and *old.record* variables to access the current record. For INSERT and UPDATE events, changes made to *new.record* in BEFORE triggers are put into the record written to disk. Other changes to *new.record* are ignored. *new.recordID*, *old.recordID*, and *old.record* are read-only variables.

BASIC programs can access columns in the current record in the usual way, for example, by referring to the field number. For example, if the SALARY column were the eighth field in the EMPLOYEES table, the following line of code would test whether SALARY had changed:

```
IF OLD.RECORD<8> # NEW.RECORD<8>...
```

Input of any kind is not allowed in trigger programs. Print output is allowed. However, if the trigger fires as the result of an **SQLExecDirect** or **SQLExecute** function call, no print output is returned.

Transactions

Whenever a change is made to a table with a trigger, a transaction starts that includes:

- The change that fires the trigger
- All changes made by the trigger program

Use the BEGIN TRANSACTION, COMMIT, and ROLLBACK statements to make nested transactions in a trigger program. The trigger program must commit or roll back any transaction it starts, and it should not commit or roll back any transaction not started by the trigger program. Because a trigger program is always in its own transaction, it cannot contain any DDL statements (ALTER TABLE, CREATE INDEX, CREATE SCHEMA, CREATE TABLE, CREATE TRIGGER, CREATE VIEW, DROP INDEX, DROP SCHEMA, DROP TABLE, DROP TRIGGER, DROP VIEW, GRANT, and REVOKE). For the same reason it cannot use the **SQLConnect** function to connect to any data source.

Certain behavior of existing programs can change if you make them into trigger programs:

- Locks are held until the outermost transaction finishes, even if the program explicitly unlocks them.
- ISOMODE of 1 does not allow records to be written outside a transaction without first locking them.

- A BASIC WRITE on a file not activated for transaction logging fails even if transaction logging is disabled or suspended.

Opening Files

Trigger programs should open all tables and files to a common variable. This avoids having to open and close each table or file whenever the program references it. For example, when you open files to a common variable, a statement that updates many records, each of which fires a trigger that writes an audit record, opens the audit file only once.

Warning*: Do not add, drop, enable, or disable any triggers for a table while the table is in use.*

Using the @HSTMT, @OLD, and @NEW Variables

As of Release 9, each UniVerse process opens a local connection to itself. This is true for both user and server processes. BASIC programs running on a UniVerse server can use the @variable @HSTMT to refer to this local connection.

Programmatic SQL uses the @HSTMT variable to access the current record. @HSTMT works similarly to the way it works in called procedures, except that in triggers, the result in @HSTMT is not returned to the client program—it can be used only by the trigger program.

Successive invocations of the same trigger, even when fired by different events, all use the same @HSTMT. The program should clear the results in @HSTMT using the **SQLFreeStmt** function before using @HSTMT again.

Warning*: If a trigger uses a statement environment to do something that either directly or indirectly fires the trigger again, the outcome of the trigger program is unpredictable since the statement environment is already in use.*

For more information about @HSTMT and called procedures, see *UniVerse BASIC SQL Client Interface Guide*.

Programmatic SQL uses two read-only @variables to contain the old and new contents of the current record:

@variable	Contains...
@OLD	The original contents of the current record
@NEW	The new contents of the current record

Programmatic SQL @variables

@OLD and @NEW contain pseudo-tables comprising one row. You can use only the SELECT statement against these pseudo-tables. Use @OLD and @NEW as qualifiers in the SELECT clause as follows:

SELECT @OLD.*columnname*, **@NEW.***columnname* …

Use them as table names in the FROM clause as follows:

SELECT * FROM @OLD …

SELECT * FROM @NEW …

Use them as qualifiers in WHERE clauses as follows:

SELECT * FROM *tablename* **WHERE @OLD.***columnname* …

SELECT * FROM *tablename* **WHERE @NEW.***columnname* …

The following example is from a program that maintains an audit file of salary changes:

```
CMD="INSERT INTO SALARY_AUDIT
   SELECT @OLD.EMPID, CURRENT_DATE, CURRENT_TIME, USER,
   @OLD.SALARY, @NEW.SALARY
   FROM @OLD, @NEW"
STATUS=SQLExecDirect(@HSTMT, CMD)
```

Handling Errors

When a trigger encounters an error, it can reject database changes. After the trigger program finishes, UniVerse checks the SQL diagnostics area associated with @HSTMT. If it finds an error, the change is rejected and the transaction is rolled back.

The trigger program can use the **SetDiagnostics** function to load error message text into the SQL diagnostics area associated with @HSTMT.

Error messages are returned as follows, depending on how the attempted change was made:

- Errors triggered by SQL statements entered at the UniVerse prompt appear on the terminal screen.
- Errors make BASIC programs take the ELSE clause. Programs can get the error text using the **GetDiagnostics** function.
- Client programs can use the **SQLError** function to get the error text.

If a program uses @HSTMT for SQL operations and an operation fails, the failure causes the trigger to reject database changes. To prevent the rejection, the program must call **ClearDiagnostics** or **GetDiagnostics** for all pending diagnostics messages.

If a program uses an HSTMT other than @HSTMT for SQL operations, the program must call **SQLError** (to get the error message), then call **SetDiagnostics** in order to make a failure reject database changes.

Handling Record and File Locks

If a trigger program contains INSERT, UPDATE, or DELETE statements that use the NOWAIT keyword, and if a record or file lock blocks the execution of such a statement, the trigger program returns an SQLSTATE of 40001. The program is responsible for taking appropriate action in such cases. Probably the most appropriate action is for the trigger program to exit at that point, passing the 40001 error to the calling program in the SQL diagnostics area.

If an INSERT, UPDATE, or DELETE statement with NOWAIT fires a trigger, the NOWAIT condition applies to all SQL operations in the trigger program. That is, in the trigger program if any SELECT, INSERT, UPDATE, or DELETE statement, with or without NOWAIT, is blocked by a record or file lock, the statement fails and returns SQLSTATE 40001 to the trigger program. If the trigger program does BASIC read or write operations, it should always use the LOCKED clause to prevent being blocked by lock conflicts.

Order of Operations

A record is written to a table as follows:

1. If the table references another table or has triggers, a transaction begins.

2. A BEFORE trigger fires if there is one. If there is an error, the trigger rolls back the transaction and aborts the write.
3. Table constraints and noncascading referential integrity are checked.
4. The record is written to the transaction cache.
5. If there are cascading referential integrity constraints, the referencing tables are updated.
6. An AFTER trigger fires if there is one. If there is an error, the trigger rolls back the transaction and aborts the write.
7. If a transaction began in step 1, it is committed.

Nested Triggers and Trigger Recursion

A database change can trigger or cascade another change, which can trigger or cascade yet other changes, and so on. If a database change fires a trigger that does a second change, and the second change fires another trigger, the second trigger is called a *nested* trigger. Trigger *recursion* occurs when the second change is to the same table as the first, thereby firing the same trigger program and thus indirectly calling itself.

Triggers contain two variables containing information about such changes:

- *count* contains the number of triggers that are currently firing.
- *chain.cascade* contains the number of active triggers in a chain of triggers.

Here is an example of how these two variables function:

1. A user initiates a change to the database.
2. The change fires a trigger that makes a second change.
3. The second change cascades a third change (via a referential constraint).
4. The third change triggers a fourth change.
5. The fourth change triggers a fifth change.

The following table shows the values returned to the *cascade*, *count*, and *chain.cascade* variables defined in each trigger program:

Event	***cascade***	***count***	***chain.cascade***
User changes a record, which fires the first trigger.	empty	1	0
The trigger makes a second change that cascades a third change that triggers a fourth change.	UPDATE or DELETE	2	1
The fourth change triggers a fifth change.	empty	3	2

Values Returned to Variables

These variables let the trigger program determine the cause of the database change that fired the trigger, as the following table shows:

Cause of the Database Change	***cascade***	***count***	***chain.cascade***
User change	empty	1	0
Changes made by the chain of triggers	empty	>1	0
Changes made by cascade or chain of cascades	UPDATE or DELETE	>=1	1

Cause of Database Changes

In addition these variable can be used to recognize more complex situations, as the following table shows:

Cause of the Database Change	***cascade***	***count***	***chain.cascade***
Changes made by cascade or chain of cascades from a triggered change or chain of triggered and cascaded changes	UPDATE or DELETE	>1	>0
Changes made by trigger or chain of triggers after cascade or chain of cascades from user change	empty	>=1	>1

Note*: A trigger program can inadvertently get into a loop pattern in which it repeatedly causes itself to fire, or in which a nested trigger on another table makes a change to the first table, firing the trigger again. Use the count argument when you want to limit the number of nested triggers currently firing.*

Some Examples

The examples in the following sections illustrate how you can use triggers to apply some of the business rules mentioned at the beginning of this chapter.

Extending Referential Integrity

To enforce the rule that a customer number on an order must correspond to a customer in the CUSTOMER table, you would normally use a referential constraint on the customer number column of the ORDER table:

...CUSTNO INT REFERENCES CUSTOMER...

If you want to extend this rule to require that the customer number on an order must correspond to an *active* customer, either a BEFORE or an AFTER trigger can enforce the rule. A BEFORE trigger is better if the table has other constraints that need to be checked first, particularly if the trigger contains many added checks that are likely to cause the change to be rejected.

The trigger should fire only when an order is added or modified, not when it is deleted. Create the trigger as follows:

```
>CREATE TRIGGER CUSTOMERCHECK
SQL+BEFORE INSERT OR UPDATE
SQL+ON CUSTOMER
SQL+FOR EACH ROW CALLING "CUSTOMERCHECK";
```

Assume that field 8 of the ORDER table contains customer numbers and field 12 contains customers' status. The STATUS column contains A for active customers. Since the initial INSERT checks to see that the customer exists, there is no need to check the customer if it wasn't changed in the ORDER record. So the first step is to exit the program if field 8 is unchanged.

Next, the program must ensure that no one deletes, renames, or changes the status of the customer after checking that he or she exists and is active, but before committing the order. One way to do this is to lock the CUSTOMER record until the ORDER record is committed. Since RECORDLOCK locks nonexistent records and requires an additional test that the customer exists, it is simpler to lock and read the CUSTOMER record. To set a read lock, the program must open the file. For efficiency in processing frequent, single-record changes, keep the file variable in the common area. The code might look like this:

```
COMMON CUSTFILE
IF TRIG.REC<8> = TRIG.REC.OLD<8> THEN RETURN
IF FILEINFO(CUSTFILE,0) ELSE OPEN 'CUSTOMER' TO CUSTFILE
   ELSE SetDiagnostics("Unable to open CUSTOMER table; STATUS is "
      :STATUS())
   RETURN
READL CUSTREC FROM CUSTFILE TRIG.REC<8>
   ON ERROR SetDiagnostics("READL error; STATUS is ":STATUS())
   LOCKED SetDiagnostics("Customer '":@RECORD<8>:"' is locked by
another
      user '":STATUS():"'. Try again later.")
   THEN NULL * Do nothing: required customer exists and is locked
   ELSE SetDiagnostics("Customer '":TRIG.REC.OLD<8>:"' does not
exist.")
IF CUSTREC<12> # 'A'
   THEN SetDiagnostics("Customer '":TRIG.REC.OLD<8>: "' is not
active.")
```

Preventing Deletions

To enforce the rule that you cannot delete a customer if he or she has any orders, you would normally use a referential constraint on the customer number column of the ORDER table:

...CUSTNO INT REFERENCES CUSTOMER...

If you want to extend this rule to require that you cannot delete a customer with any *open* orders, you can use a trigger to enforce the rule as an AFTER DELETE trigger. Creating it as an AFTER trigger ensures that another user cannot add an open order after the trigger program checks that there are none. When a user tries to add such an order, the trigger program needs to check that the customer exists and should fail after the customer is deleted.

The trigger program can look for open orders with code such as the following:

```
SELECT COUNT(*) FROM ORDERS, @OLD
   WHERE ORDERS.CUSTNUM = @OLD.NUM AND ORDERS.STATUS = 'OPEN'
```

Validating Data

To validate data and prevent the making of specific changes based on date, time, user, or the data itself, a trigger might include code like the following:

```
IF... SetDiagnostics("You can't do that!")
```

Changing a Record Before Writing It

A trigger can change data in a record before writing it to disk. You might do this to provide calculated defaults or to insert date, time, or user stamps. For example, a table might include columns containing the date and time of the last modification. You can use a BEFORE UPDATE trigger to maintain these columns and replace any data a user tries to put in them. If a record contains a date stamp in field 4 and a time stamp in field 5, the trigger might contain code such as the following:

```
TRIG.REC<4> = DATE()
TRIG.REC<5> = TIME()
* check if midnight occurred between the above two calls
IF TRIG.REC<4> # DATE() AND TIME() > TRIG.REC<5>
THEN TRIG.REC<4> = DATE()
```

The BASIC functions DATE and TIME refer to system date and time and do not change. UniVerse SQL uses a different concept of date and time. While an SQL statement or a trigger is being executed, the SQL keywords CURRENT_DATE and CURRENT_TIME return the same date and time for each reference. BASIC can access these values using the @SQL.DATE and @SQL.TIME variables.

Auditing Changes

A trigger can create copies of changes made to a table and store them in an audit file, replicating the data. To write an audit record containing selected data plus time stamp information, a program might use code such as the following:

```
CMD="INSERT INTO SALARYAUDITFILE SELECT (@OLD.ID,
   CURRENT_DATE, CURRENT_TIME, USER, @OLD.SALARY, @NEW.SALARY)
   FROM @OLD, @NEW"
SQLExecDirect(@HSTMT, CMD)
```

For efficiency across transactions, a BASIC program needs to keep the audit file open using the common area. A trigger program could then materialize an audit record and record ID, then write the record, using code such as the following:

```
AUDREC<1>=TRIG.REC.OLD<8>
AUDREC<2>=TRIG.REC<8>
AUDID<1>=TRIG.ID
AUDID<2>=DATE()
AUDID<3>=TIME()
AUDID<4>=@ACCOUNT
WRITE AUDREC TO SALAUDFILE AUDID
   ON ERROR SetDiagnostics( 'Audit error; STATUS is ':STATUS())
```

To reference the SQL CURRENT_DATE and CURRENT_TIME, replace DATE( ) and TIME( ) with @SQL.DATE and @SQL.TIME, respectively.

Chapter

UniVerse SQL Statements

6

This chapter describes every UniVerse SQL statement. The first part of the chapter comprises reference pages that describe the following statements. The statements are arranged alphabetically.

- ALTER TABLE
- CALL
- CREATE INDEX
- CREATE SCHEMA
- CREATE TABLE
- CREATE TRIGGER
- CREATE VIEW
- DELETE
- DROP INDEX
- DROP SCHEMA
- DROP TABLE
- DROP TRIGGER
- DROP VIEW
- GRANT
- INSERT
- REVOKE
- SELECT
- UPDATE

The second part of the chapter describes the following elements of the syntax that are common to several statements. Reference pages of statements in the first part of "UniVerse SQL Statements" have cross-references to the syntax elements in the second part of the chapter.

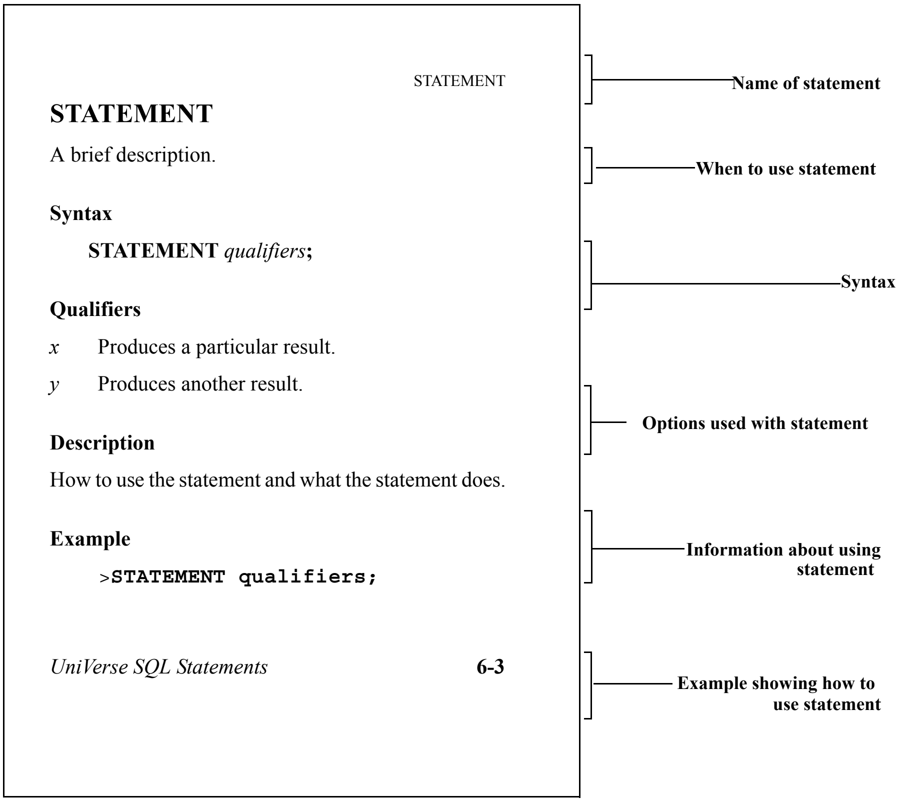
- Column
- Condition
- Data type
- Expression
- Identifier
- Literal

- Relational operator
- Set function
- Subquery
- Table

SQL reserved words (statement names and all keywords) are case-insensitive. You can type them in uppercase, lowercase, or mixed case letters. In this book they are always shown in uppercase letters.

Statement Page Layout

The following sample shows a typical statement reference page:


STATEMENT

STATEMENT

A brief description.

Syntax

STATEMENT *qualifiers***;**

Qualifiers

x Produces a particular result.

y Produces another result.

Description

How to use the statement and what the statement does.

Example

```
>STATEMENT qualifiers;
```

UniVerse SQL Statements **6-3**

Name of statement
When to use statement
Syntax
Options used with statement
Information about using statement
Example showing how to use statement

ALTER TABLE

Use the ALTER TABLE statement to modify the definition of an existing base table. ALTER TABLE can add columns, table constraints, or associations; remove table constraints, associations, or default values; change a column's default value; and enable or disable triggers. To use the ALTER TABLE statement, you must own the table or have ALTER Privilege on it.

Syntax

ALTER TABLE *tablename* { ADD clause | DROP clause | ALTER clause | TRIGGER clause } ;

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
tablename	An identifier specifying the name of a table to modify in the schema you are logged in to.
ADD clause	Adds a column, column synonym, association, or table constraint to the table.
DROP clause	Removes an association or a table constraint from the table.
ALTER clause	Adds, changes, or removes the default value of a column.
TRIGGER clause	Enables or disables the specified triggers.

ALTER TABLE Parameters

Description

You must use either the ADD, DROP, ALTER, or TRIGGER clause to modify a table. To use the ADD clause to add a table constraint, you must have write permissions on the VOC file in the current directory.

Altering a table on which a view depends does not invalidate the view.

***Note**: Since the ALTER TABLE statement modifies the structure of a table, use ALTER TABLE only when others are not using the table. This avoids lock conflicts.*

ADD Clause: Column

To add a column to a table, use the following syntax:

ADD [**COLUMN**] *columnname datatype* [DEFAULT clause] [*column_constraints*] [*output_format*

Parameter	Description
columnname	An identifier specifying the name of the column.
datatype	Specifies the kind of data in the column. You must define the data type for each column in a table. For detailed information about data types, see Chapter 3, "Data Types."
DEFAULT clause	Specifies the value to insert into a column when no value is specified.
column_constraints	One or more column constraints separated by spaces. Column constraints protect the integrity and consistency of the database. You cannot specify a PRIMARY KEY constraint using ALTER TABLE. You can use UNIQUE only if the table is empty (that is, has no rows).
output_format	One or more of the following, separated by spaces: DISPLAYNAME '*column_heading*' CONV '*conversion_code*' FMT '*format_code*'

ADD Column Parameters

You can add columns to an empty table or to a table that already has rows. Adding a column to a table does not affect any existing query specification in a view that is derived from the altered table.

When you add a column to a table that already contains rows, each row is rewritten. The new column's default value is added to each row.

This example adds the COLOR column to the INVENTORY table, specifying the VARCHAR data type:

```
>ALTER TABLE INVENTORY ADD COLOR VARCHAR;
```

The next example adds the ORD_DATE column to the ORDERS table, specifying a conversion code and a column heading:

```
>ALTER TABLE ORDERS ADD ORD_DATE DATE CONV 'D-' DIS-
PLAYNAME 'Date';
```

ADD Clause: Column Synonym

To add a column synonym to a table, use the following syntax:

ADD [**COLUMN**] *synonym* **SYNONYM FOR** *columnname* [*output_format*]

The following table describes each parameter of the syntax.

Parameter	Description
synonym	The name of a column synonym.
columnname	An identifier specifying the name of an existing column in this table.
output_format	For *output_format* syntax, see "Output Format."

ADD Clause Column Synonym Parameters

The column synonym is created as another entry in the dictionary of the table, but it is not added to the SICA.

The following example adds the ADDRESS synonym for the existing BILLTO column to the CUSTOMERS table:

```
>ALTER TABLE CUSTOMERS ADD ADDRESS SYNONYM FOR
BILLTO;
```

ADD Clause: Association

To create a new association of multivalued columns in a table, use the following syntax:

ADD ASSOC [**IATION**] *name* [*position*] **(***columnname* [**KEY**] [**,***columnname* [**KEY**]]… **)**

The following table describes each parameter of the syntax.

Parameter	Description
name	An identifier specifying the name of the association. *name* cannot be the same as another column name or association name in this table.
position	For the description of *position*, see "Positioning of Association Rows."
columnname	An identifier specifying the name of a multivalued column to include in the association. All columns specified in an ASSOC clause must be defined in a column definition and cannot belong to another association. *columnname* cannot be an alias.
KEY	Defines the preceding *columnname* as an association key column. Key columns must have a column constraint of NOT NULL. If you specify only one column as the key, it must have the column constraint ROWUNIQUE. If you specify two or more columns as the association key, they are treated as jointly rowunique at run time. You cannot specify any association keys if *position* is INSERT PRESERVING.

ADD Clause: Association Parameters

Note*: If any columns you include in the association have check constraints, you must drop the check constraint and recreate it.*

This example adds the BOUGHT association for the ITEM and QTY columns to the ORDERS table:

```
>ALTER TABLE ORDERS ADD ASSOC BOUGHT (ITEM KEY,
QTY);
```

ADD Clause: Table Constraint

To add a CHECK, UNIQUE, or FOREIGN KEY table constraint to a table, use the following syntax:

ADD [**CONSTRAINT** *name*] *table_constraint*

The following table describes each parameter of the syntax.

Parameter	Description
name	An identifier specifying the name of the table constraint. If you do not specify *name*, the system generates one in the format UVCON_*n*.
table_constraint	One of the following: UNIQUE CHECK FOREIGN KEY

ADD Clause: Table Constraint Parameters

Note*: Newly added table constraints must not violate existing data in the table. If a violation occurs, UniVerse SQL returns a message and does not execute the statement. You can delete the rows that caused the violation and retry the ALTER TABLE statement.*

This example adds a CHECK table constraint called CHK_DATE to the ORDERS table. The constraint specifies that only orders dated after January 1, 1994 are allowed in the table.

```
>ALTER TABLE ORDERS ADD CONSTRAINT CHK_DATE CHECK ("DATE" >
'1/1/94');
```

DROP Clause: Association

To remove an association from a table, use the following syntax:

DROP ASSOC[**IATION**] *name*

name is the name of the association to remove.

This example drops the association BOUGHT from the ORDERS table:

```
>ALTER TABLE ORDERS DROP ASSOCIATION BOUGHT;
```

DROP Clause: Integrity Constraint

To remove an integrity constraint from a table, use the following syntax:[1]

DROP CONSTRAINT *name* [**RESTRICT** | **CASCADE**

The following table describes each parameter of the syntax.

Parameter	Description
name	The name of the constraint to be dropped. You can drop UNIQUE, CHECK, and FOREIGN KEY table constraints. You can also drop CHECK and REFERENCES column constraints. You cannot drop a PRIMARY KEY constraint.
RESTRICT	Prevents removal of the UNIQUE constraint if the columns are referenced by foreign key columns. RESTRICT is the default keyword.
CASCADE	Removes the UNIQUE constraint from referenced columns, and also removes the referential constraints from the referencing foreign key columns. For example, you want to remove a foreign key constraint from the CUSTNO column in the ORDERS table. This column references the CUSTNO column in the CUSTOMERS table. If you use the CASCADE option, it removes the UNIQUE constraint from the CUSTNO column of the CUSTOMERS table, and it also removes the referential constraint from the CUSTNO column of the ORDERS table.

DROP Clause Integrity Constraint Parameters

This example drops the table constraint named UNIQ_DATE from the ORDERS table:

```
>ALTER TABLE ORDERS DROP CONSTRAINT UNIQ_DATE;
```

1. Tables created on a UniVerse Release 7 system may include unnamed constraints. To drop an unnamed constraint, use LIST.SICA to list any unnamed constraints, then use ALTER TABLE to drop the constraint. Use the following syntax: **ALTER TABLE** *tablename* **DROP CONSTRAINT "UNNAMED****n***"**, where *n* is the position number of the constraint as shown by LIST.SICA.

ALTER Clause: SET DEFAULT

To change the default value of a column, use the following syntax:

ALTER [**COLUMN**] *columnname* **SET DEFAULT** { *value* | **NULL** | **USER** }

The following table describes each parameter of the syntax.

Parameter	Description
columnname	An identifier specifying the name of the column whose default value is to be changed.
value	Either a character string enclosed in single quotation marks, a bit string literal, or a number. If *value* is the empty string, the column's data type must be CHAR, VARCHAR, NCHAR, or NVARCHAR, and it cannot have the column constraint NOT EMPTY.
NULL	If the default column value is NULL, the column cannot have the column constraint NOT NULL. You cannot specify NULL as the default value of a column that is part of a primary key.
USER	Specifies the effective user name of the current user. The column's data type must be CHAR or VARCHAR.

ALTER Clause: SET DEFAULT Parameters

This example changes the default value for the CUSTOMER column in the ORDERS table to AJAX. If a default value was previously defined, this redefines it for new default data, but it does not change existing data in the column.

```
>ALTER TABLE ORDERS ALTER COLUMN CUSTOMER SET DEFAULT 'AJAX';
```

ALTER Clause: DROP DEFAULT

To remove the default value of a column, use the following syntax:

ALTER [**COLUMN**] *columnname* **DROP DEFAULT**

The following table describes each parameter of the syntax.

Parameter	Description
columnname	An identifier specifying the name of the column whose default value is to be dropped.
DROP DEFAULT	Removes the default value of a column.

ALTER Clause: DROP DEFAULT Parameters

Removing the default value of a column effectively sets the default value to null. It does not affect any data in existing rows.

This example removes the current default value definition from the CUSTOMER column in the ORDERS table:

```
>ALTER TABLE ORDERS ALTER COLUMN CUSTOMER DROP DEFAULT;
```

TRIGGER Clause

To enable or disable one trigger or all triggers associated with a table, use the following syntax:

{ **ENABLE** | **DISABLE** } **TRIGGER** { *triggername* | **ALL** }

The following table describes each parameter of the syntax.

Parameter	Description
triggername	An identifier specifying the trigger you want to enable or disable.
ALL	Enables or disables all of a table’s triggers.

TRIGGER Clause Parameters

Do not enable or disable a trigger while a program using the table is running. Results may be unpredictable.

This example disables the AUDIT_EMPLOYEES trigger of the EMPLOYEES table:

```
>ALTER TABLE EMPLOYEES DISABLE TRIGGER AUDIT_EMPLOYEES;
```

CALL

Use the CALL statement in an ODBC client program to call a procedure stored on a server system. You can use CALL only in programmatic SQL.

Syntax

CALL *procedure* [**(**[*parameter* [**,***parameter*]…]**)**] [**;**]

CALL *procedure* [*argument* [*argument*]…] [**;**]

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
procedure	The name of the procedure to call and execute. If *procedure* contains characters other than alphabetic or numeric, enclose the name in double quotation marks. To embed a double quotation mark in *procedure*, use two consecutive double quotation marks.
parameter	Either a literal value or a parameter marker. Programmatic SQL uses a ? (question mark) as a parameter marker. Use parameters only if the procedure is a BASIC subroutine. The number and order of parameters must correspond to the number and order of the subroutine arguments.
argument	Any keyword, literal, or other token you can use in a UniVerse command line. Use arguments only if the procedure is a UniVerse paragraph, stored sentence, UniVerse command, ProVerb proc, or BASIC program that accepts command line arguments.

CALL Parameters

Description

The CALL statement has two syntaxes. The first follows the ODBC pattern, in which a comma-separated list of arguments is enclosed in parentheses. The second follows the UniVerse syntax pattern, in which a space-separated list of arguments not enclosed in parentheses follows the procedure name.

You can call any of the following as a procedure:

- BASIC subroutine
- BASIC program
- UniVerse paragraph
- ProVerb proc
- Stored sentence
- UniVerse command

BASIC subroutines must be globally, normally, or locally cataloged. If a BASIC program is not cataloged, CALL searches the BP file in the account your program is connected to.

You can call a UniVerse command as a procedure if the VOC entry defining it contains a G in field 4.

For complete information about writing and using called procedures, see the *UniVerse BASIC SQL Client Interface Guide* or the *UCI Developer's Guide*.

CREATE INDEX

Use the CREATE INDEX statement to create a secondary index on a table. You must be the owner of the table, have ALTER Privilege on it, or have DBA Privilege, to create an index on a table.

Syntax

CREATE [**UNIQUE**] **INDEX** *indexname* **ON** *tablename*
(*columnname* [**ASC** | **DESC**] [**,***columnname* [**ASC** | **DESC**]]...**);**

Parameters

The following table describes each parameter of the syntax.

Parameter	**Description**
UNIQUE	Requires that data in the secondary index be unique. When you create a unique secondary index, a UNIQUE Table Constraintis also created on the table.
indexname	An identifier specifying the name of the index. *indexname* cannot include the **Space** character. An index name concatenated to its corresponding table name with a period must be unique within a schema. *indexname* must not duplicate an entry in the table's dictionary.
tablename	An identifier specifying the name of a table in the schema you are logged in to whose columns you want to index. *tablename* cannot include any of the following characters: " (double quotation mark)\ (backslash) , (comma)**Space**

CREATE INDEX Parameters

Parameter	Description
columnname	An identifier specifying the name of the column you want to index. *columnname* cannot be a column synonym. In a unique index, columns must be defined as NOT NULL, cannot be part of the primary key or a unique constraint, and cannot be part of another unique index. All columns in a multicolumn index must be singlevalued.
ASC	Sorts values in the index in ascending order. ASC is the default sort order.
DESC	Sorts values in the index in descending order.

CREATE INDEX Parameters (Continued)

Description

Using secondary indexes can improve the performance of certain queries where the table lookup is not based on the primary key.

You can create indexes only on base tables, but not on views, associations, or UniVerse files. You can create indexes on up to 16 columns in a table. The total length of the indexed data including delimiters cannot exceed 255 characters (the maximum record ID length).

CREATE INDEX creates an I-descriptor in the table dictionary whose record ID is *indexname*. Field 1 contains a description of the index in the following format:

ISQL *column1* [ASC | DESC] *column2* [ASC | DESC] …

Example

This example creates secondary indexes on two columns of the VENDORS.T table:

```
>CREATE INDEX VENDORS_INDEX ON VENDORS.T (COMPANY, CONTACT);
Adding Index "VENDORS_INDEX".
```

The I-descriptor in the dictionary looks like this:

```
     VENDORS_INDEX
0001 ISQL COMPANY ASC CONTACT ASC
0002 IF ISNULL(COMPANY) THEN @NULL.STR ELSE COMPANY;IF
     ISNULL(CONTACT) THEN @NULL.STR ELSE CONTACT;@1:@TM:@2
0003
0004
0005 10R
0006
0007
```

CREATE SCHEMA

Use the CREATE SCHEMA statement to create a new UniVerse SQL schema. You can create the schema as a new UniVerse account directory, or you can convert an existing UniVerse account to a schema. Users must have RESOURCE Privilege to create their own schemas, andDBA Privilege to create schemas for other users.

Syntax

CREATE SCHEMA [*schema*] [**AUTHORIZATION** *owner*]
[**HOME** *pathname*]
[CREATE TABLE statements]
[CREATE VIEW statements]
[CREATE INDEX statements]
[CREATE TRIGGER statements]
[GRANT statements]**;**

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
schema	An identifier specifying the name of the schema. *schema* must not be the name of an existing schema. If you do not specify *schema*, you must include the AUTHORIZATION clause.
AUTHORIZATION *owner*	Specifies the owner of the schema. You must have DBA Privilege to specify another user as *owner*. If you do not have DBA privilege, you must specify yourself as *owner*. If you omit the AUTHORIZATION clause, the schema owner is set to your effective user name. If you do not specify *schema*, *owner* is the name of the schema. *owner* must be a UniVerse SQL user defined in the SQL catalog and must have write permissions on the account directory where the schema resides.
HOME *pathname*	Specifies the account directory where the schema is to reside. *pathname* is the full path of an existing directory. The owner of the schema must have write permissions on this directory. If you omit the HOME clause, the current UniVerse account directory is converted to a schema.
CREATE TABLE statements	For information about the CREATE TABLE statement, see CREATE TABLE.
CREATE VIEW statements	In a CREATE SCHEMA statement, any CREATE VIEW statement must follow the statements that create the view's underlying tables and views.
CREATE INDEX statements	In a CREATE SCHEMA statement, any CREATE INDEX statement must follow the statement that creates the table being indexed.
CREATE TRIGGER statements	In a CREATE SCHEMA statement, any CREATE TRIGGER statement must follow the statement that creates the table the trigger is associated with.
GRANT statements	In a CREATE SCHEMA statement, any GRANT statement must follow the statement that creates the table or view it refers to.

CREATE SCHEMA Parameters

Do not use a ; (semicolon) to terminate CREATE TABLE, CREATE VIEW, CREATE INDEX, CREATE TRIGGER, and GRANT statements in the CREATE SCHEMA statement. Use only one semicolon to terminate the entire CREATE SCHEMA statement.

Description

A schema is created as a UniVerse account. If the schema directory does not contain a UniVerse account, CREATE SCHEMA creates all the necessary UniVerse files. When you create a schema, its name is added to the UV_SCHEMA table in the SQL catalog.

Any files that are in the schema directory when you create the schema are unaffected by the CREATE SCHEMA statement.

Examples

This example creates a schema called Susan in the directory */usr/susan*:

```
>CREATE SCHEMA Susan HOME /usr/susan;
```

The next example creates the schema PERSONNEL, specifies its owner, *sally*, and creates the EMPLOYEES table in it:

```
>CREATE SCHEMA PERSONNEL HOME /usr/personnel
SQL+AUTHORIZATION sally
SQL+CREATE TABLE EMPLOYEES
SQL+(EMPNO INT PRIMARY KEY,
   .
   .
   .
```

CREATE TABLE

Use the CREATE TABLE statement to create a new table in the schema you are logged in to. CREATE TABLE creates the data file and the file dictionary in the current schema.

Syntax

CREATE TABLE *tablename* [**DATA** *pathname*] [**DICT** *pathname*] **(**[*file_format*,] *column_definitions* [,*associations*] [,*table_constraints*]**);**

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
tablename	An identifier specifying the name of a table in the schema you are logged in to. *tablename* must not exist in the schema or be the ID of a record in the VOC file.
pathname	The absolute path of the table's data file or its dictionary. The directory to contain the data file or dictionary must already exist, and the file name of the data file or dictionary must not exist. If you do not specify paths for the table's data file or its dictionary, the default paths are: Data file: *current_directory/tablename* Dictionary: *current_directory/*D*_tablename*
file_format	One or more clauses that define the file format. If you do not specify a file format, the table is created as a dynamic file.
column_definitions	One or more column definitions separated by commas.
associations	One or more association definitions separated by commas.
table_constraints	One or more table constraint definitions separated by commas.

CREATE TABLE Parameters

Description

When you create a table, you define one or more columns for it. You also define each column's data type. You can put constraints on columns that protect the integrity and consistency of the data. And you can associate multivalued columns so each value in one column is associated with its corresponding values in the associated columns.

If you want to add columns to an existing table, or if you want to add or remove table constraints or association definitions to or from an existing table, use the ALTER TABLE statement.

File Format

The UniVerse SQL CREATE TABLE statement lets you specify the UniVerse file format of a table. As with the UniVerse CREATE.FILE command, you can specify the file type of the data file. If you create the table as a static hashed file, you can specify its modulo and separation; if you create it as a dynamic file, you can specify dynamic file parameters. For more details about file formats, see *UniVerse System Description*.

The syntax of the file format options is as follows:

[{ **TYPE** *type#* | **DYNAMIC** } ,] [**MODULO** *mod#*,] [**SEPARATION** *sep#*,]
[*dynamic_parameters* ,]

The following table describes each parameter of the syntax.

Parameter	Description
TYPE *type#*	Specifies the UniVerse file type. *type#* is a number from 2 through 18, and 30. Types 2 through 18 are static hashed files. Type 30 files are dynamic files. If you specify a type from 2 through 18, you can specify modulo and separation as well. If you specify type 30, you can set the file parameters using the dynamic parameter keywords. An SQL table cannot be a type 1, type 19, or type 25 file, nor can it be a distributed file or a file with multiple data files.
DYNAMIC	Specifies a UniVerse dynamic file (type 30). If you specify a dynamic file type, you can set the file parameters using the dynamic parameter keywords.
MODULO *mod#*	Specifies the number of groups in the file. *mod#* is an integer from 1 through 8,388,608. Modulo is ignored if the file is dynamic.
SEPARATION *sep#*	Specifies the group buffer size in 512-byte blocks. *sep#* is an integer from 1 through 8,388,608. If you specify a file type from 2 through 18 and you do not specify separation, a default separation of 4 (2048 bytes) is used. Separation is ignored if the file is dynamic.
dynamic_parameters	One or more of the following, separated by commas: GENERAL | SEQ NUM GROUP SIZE { 1 | 2 } MINIMUM MODULUS *n* SPLIT LOAD *n* MERGE LOAD *n* LARGE RECORD *n* RECORD SIZE *n* MINIMIZE SPACE For complete details, see "Dynamic File Parameters" on page 27.

File Format Parameters

Dynamic File Parameters

Dynamic file parameters define a hashing algorithm, the group size, the minimum number of groups in the file, modulo parameters, row size, and the amount of space the file needs.

Parameter	**Description**
GENERAL	Specifies that the general hashing algorithm should be used for the dynamic file. GENERAL is the default.
SEQ NUM	Specifies that a hashing algorithm suitable for sequential numbers should be used for the dynamic file. You should use this hashing algorithm only for rows with primary keys that are mainly numeric, sequential, and consecutive.
GROUP SIZE { 1 | 2 }	Specifies the size of each group in the file. The argument 1 specifies a group size of 2048 bytes, which is equivalent to a separation of 4. The argument 2 specifies a group size of 4096 bytes, which is equivalent to a separation of 8. 1 is the default.
MINIMUM MODULUS *n*	Specifies the minimum modulo of the file. This keyword takes an integer argument greater than 1. This value is also the initial value of the modulo of the dynamic file. 1 is the default.
SPLIT LOAD *n*	Specifies the level at which the file's modulo is increased by 1. SPLIT LOAD takes a numeric argument indicating a percentage of the space allocated for the file. When the data in the file exceeds the space allocated for the file, the data in one of the groups divides equally between itself and a new group, to increase the modulo by 1. The default SPLIT LOAD is 80%.
MERGE LOAD *n*	Specifies the level at which the file's modulo is decreased by 1. MERGE LOAD takes a numeric argument indicating a percentage of the space allocated for the file. When the data in the file uses less than the space allocated for the file, the data in the last group of the file merges with another group, to decrease the modulo by 1. The default MERGE LOAD is 50%.

Dynamic File Parameters

Parameter	Description
LARGE RECORD *n*	Specifies the size of a row to be considered too large to be included in the primary group buffer. LARGE RECORD takes an integer as an argument indicating a percentage of the group size. When the size of a row exceeds the percentage specified, the data for the row is put in an overflow buffer, but the primary key (or the @ID value if the table has no primary key) is put in the primary buffer. This method of large row storage increases access speed. The default LARGE RECORD size is 80%.
RECORD SIZE *n*	Causes the values for group size and large row size to be calculated based on the value of the estimated average row size specified. RECORD SIZE takes an argument of your estimate of the average row size for the dynamic file, specified in bytes. RECORD SIZE does not limit the size of rows. If you specify a value for group size or large row size, the value you specify overrides the value calculated by RECORD SIZE.
MINIMIZE SPACE	Calculates the values for the split load, merge load, and the large row size to optimize the amount of space required by the file at the expense of access time. If you specify a value for split load, merge load, or large row size, the value you specify overrides the value that is calculated by MINIMIZE SPACE. If MINIMIZE SPACE and RECORD SIZE are both specified, the value for large row size calculated by MINIMIZE SPACE overrides the value calculated by RECORD SIZE.

Dynamic File Parameters (Continued)

This example creates the ORDERS table as a type 18 static hashed file, with a modulo of 59 and a separation of 4:

```
>CREATE TABLE ORDERS
SQL+(TYPE 18, MODULO 59, SEPARATION 4,
SQL+ORDERNO INT PRIMARY KEY,
SQL+"DATE" DATE NOT NULL CONV 'D2/',
    .
    .
    .
```

The next example creates the same table as a dynamic file. Dynamic file parameters specify a hashing algorithm suitable for sequential numbers, a group size of 2 (4096 bytes), a minimum modulo of 59, and space optimization.

```
>CREATE TABLE ORDERS
SQL+(DYNAMIC, SEQ NUM, GROUP SIZE 2,
SQL+MINIMUM MODULUS 59, MINIMIZE SPACE,
SQL+ORDERNO INT PRIMARY KEY,
SQL+"DATE" DATE NOT NULL CONV 'D2/',
    .
    .
    .
```

Column Definition

The CREATE TABLE statement requires at least one column definition.

A column definition names the column and defines its data type. It can also specify default column values and integrity constraints. A column definition can also define a synonym for an existing column.

The syntax of a column definition is as follows:

columnname datatype [DEFAULT clause] [*column_constraints*] [*output_format*]

The following table describes each parameter of the syntax.

Parameter	**Description**
columnname	An identifier specifying the name of the column. *columnname* must be unique within the table.
datatype	Specifies the kind of data in the column. You must define the data type for each column in a table. For detailed information about data types, see Chapter 3, "Data Types."

Column Definition Parameters

Parameter	Description
DEFAULT Clause	Specifies the value to insert into a column when no value is specified.
column_constraints	One or more column constraints separated by spaces. Column constraints protect the integrity and consistency of the database.
output_format	One or more of the following, separated by spaces: DISPLAYNAME '*column_heading*' CONV '*conversion_code*' FMT '*format_code*'

Column Definition Parameters (Continued)

This example defines four columns in the CUSTOMERS file. The first column is the primary key. The next three columns include output format specifications.

```
>CREATE TABLE CUSTOMERS
SQL+(CUSTNO INT PRIMARY KEY,
SQL+FNAME VARCHAR FMT '15T',
SQL+LNAME VARCHAR FMT '15T',
SQL+COMPANY VARCHAR FMT '20T',
    .
    .
    .
Creating Table "CUSTOMERS"
Adding Column CUSTNO
Adding Column FNAME
Adding Column LNAME
Adding Column COMPANY
    .
    .
    .
```

DEFAULT Clause

The DEFAULT clause specifies the value to put into a column if no value is specified for this column in an INSERT or UPDATE statement. If you do not specify a DEFAULT clause, the default column value is NULL. The syntax is as follows:

DEFAULT { *value* | **NEXT AVAILABLE** | **NULL** | **USER** | **CURRENT_DATE** | **CURRENT_TIME** }

The following table describes each parameter of the syntax.

Parameter	Description
value	A character string enclosed in quotation marks, a bit string literal, or a number. If *value* is the empty string, the column's data type must be CHAR, VARCHAR, NCHAR, or NVARCHAR, and it cannot have the column constraint NOT EMPTY.
NEXT AVAILABLE	Specifies that a primary key column automatically generates a numeric sequence of integers starting with 1. The column's data type must be INT, NUMERIC, or DECIMAL. Use this option only in a column definition for a primary key.
NULL	If the default column value is NULL, the column cannot have the column constraint NOT NULL. You cannot specify NULL as the default value of a column that is part of a primary key.
USER	Specifies the effective user name of the current user. The column's data type must be CHAR or VARCHAR.
CURRENT_DATE	Specifies the current system date. The column's data type must be DATE.
CURRENT_TIME	Specifies the current system time. The column's data type must be TIME.

DEFAULT Clause Parameters

This example specifies the empty string as the default column value of the FNAME and LNAME columns:

```
>CREATE TABLE CUSTOMERS
SQL+(CUSTNO INT PRIMARY KEY,
SQL+FNAME VARCHAR FMT '15T' DEFAULT '',
SQL+LNAME VARCHAR FMT '15T' DEFAULT '',
   .
   .
   .
```

Output Format

Use output format specifications to specify the following for a column:

- A column heading (DISPLAYNAME)
- A conversion code (CONV)

- Width and justification of the display column, and other format masking (FMT)

DISPLAYNAME

Use DISPLAYNAME (or one of its synonyms: DISPLAY.NAME or COL.HDG) to specify a column heading. The syntax is as follows:

> **DISPLAYNAME '***column_heading***'**

If you do not specify a column heading, *columnname* is the column heading. To specify a line break in the column heading, use the letter L enclosed in single quotation marks and enclose *column_heading* in double quotation marks.

CONV

Use CONV (or its synonym CONVERSION) to specify an output conversion for a column. The syntax is as follows:

> **CONV '***conversion_code***'**

If you do not specify a conversion code for a column, UniVerse SQL generates one if it is appropriate. For example, if you specify the data type DEC(9,2) for a column, UniVerse SQL supplies the conversion code MD22.

***Note**: If you specify a conversion code, it must be compatible with the column's data type, otherwise results will be unpredictable. See Chapter 3, "Data Types."*

For more information about conversion codes, see *UniVerse BASIC*.

FMT

Use FMT (or its synonym FORMAT) to specify the width and justification of the display column. The syntax is as follows:

> **FMT '***format_code***'**

You can specify a variety of format masks with FMT. For full details about the syntax of the format code, see the FMT function in *UniVerse BASIC*.

This example defines a column heading, the display-column width, and the justification for the CUSTNO and BILLTO columns:

```
>CREATE TABLE CUSTOMERS
SQL+(CUSTNO INT PRIMARY KEY DISPLAYNAME 'Customer'
SQL+FMT '4R',
SQL+BILLTO VARCHAR DISPLAYNAME 'Name & Address' FMT '30T',
   .
   .
   .
```

The next example uses the conversion code MD2$ to define a monetary output format for the COST column:

```
>CREATE TABLE INVENTORY
SQL+(PRODNO INT PRIMARY KEY,
SQL+DESCRIP VARCHAR FMT '32T',
SQL+QOH INT FMT '4R',
SQL+COST DEC(6,2) CONV 'MD2$',
   .
   .
   .
```

Column Constraints

Constraints protect the integrity and consistency of data. Column constraints are part of a column definition and affect only one column. To define a constraint for more than one column, use table constraints.

The syntax for defining a column constraint is as follows:

[**CONSTRAINT** *name*] *column_constraint*

The following table describes each parameter of the syntax.

Parameter	Description
name	An identifier specifying the name of the column constraint. You can name the following column constraints: NOT EMPTY, NOT NULL, CHECK, and REFERENCES. If you do not specify *name*, the system generates one in the format UVCON_*n*.
column_constraint	One of the following: NOT EMPTY NOT NULL [UNIQUE | ROWUNIQUE] [NOT NULL] PRIMARY KEY CHECK REFERENCES MULTIVALUED | SINGLEVALUED See the following sections for descriptions of each column constraint.

CONSTRAINT Parameters

NOT EMPTY

Use NOT EMPTY to specify that the column cannot contain empty string values.

NOT NULL

Use NOT NULL to specify that the column cannot contain null values. If you also specify UNIQUE or ROWUNIQUE, the NOT NULL keyword must immediately precede it.

PRIMARY KEY

Use PRIMARY KEY to define the column as the primary key of the table. NOT NULL is optional before PRIMARY KEY, but the column must be single-valued and cannot contain null values. A CREATE TABLE statement can have only one PRIMARY KEY column constraint. Use the PRIMARY KEY table constraint to define more than one column as the primary key.

This example defines the ORDERNO column as the primary key of the ORDERS table:

```
>CREATE TABLE ORDERS
SQL+(ORDERNO INT PRIMARY KEY,
    .
    .
    .
```

UNIQUE

Use UNIQUE to specify that the column cannot contain duplicate values. If the column is singlevalued, it can have no duplicate values in more than one row. If the column is multivalued, it can have no duplicate values in the same row or in any other row. The column must have the NOT NULL constraint.

This example defines the DESCRIP column in the INVENTORY table as UNIQUE. This ensures that all items in inventory will have unique descriptions.

```
>CREATE TABLE INVENTORY
SQL+(PRODNO SMALLINT PRIMARY KEY,
SQL+DESCRIP VARCHAR NOT NULL UNIQUE FMT '32T',
    .
    .
    .
```

ROWUNIQUE

Use ROWUNIQUE to specify that no row in a multivalued column can contain duplicate values. You can put the ROWUNIQUE constraint only on a column defined as MULTIVALUED. Each value must be unique in the column row, but the same values can appear in other rows in the column. The column must have the NOT NULL constraint.

If you define a single column as the key to an association, CREATE TABLE automatically puts the ROWUNIQUE constraint on it. If you define two or more columns as the key, they are treated as jointly rowunique at run time.

This example defines the multivalued column PRODNO in the ORDERS table as ROWUNIQUE. Each row in ORDERS represents one order. Each order can include the same product number only once—that is, within each row, product numbers must be unique. But different orders can include the same product number—that is, the same product number can appear in different rows.

```
>CREATE TABLE ORDERS
SQL+(ORDERNO SMALLINT PRIMARY KEY,
SQL+PRODNO SMALLINT MULTIVALUED NOT NULL ROWUNIQUE,
    .
    .
    .
```

CHECK

Use CHECK to specify criteria a value must meet to be included in the column. The syntax is as follows:

CHECK (*condition***)**

In a column constraint, *condition* can be checked only against the column itself. Use a table constraint to check against values in other columns in the table.

You cannot use the CURRENT_DATE and CURRENT_TIME keywords in a CHECK condition.

If *condition* includes columns in an association, you must put the CHECK clause after the ASSOC Clause that defines the association.

This example defines a CHECK constraint for the PRODNO column. Values in the PRODNO column must fall between 100 and 999.

```
>CREATE TABLE INVENTORY
SQL+(PRODNO INT PRIMARY KEY
SQL+CHECK (PRODNO BETWEEN 100 AND 999),
    .
    .
    .
```

CHECK constraints are enforced:

- When you add or change data values using the INSERT and UPDATE statements
- When you add or change data values using the UniVerse Editor or ReVise
- When a BASIC program writes to a table opened with the OPENCHECK statement

- When a BASIC program uses the ICHECK function to check if data values violate SQL integrity constraints
- For all BASIC programs, if the OPENCHK configurable parameter is set (this is the default setting)

REFERENCES

Use REFERENCES to put a referential constraint on the column. To put a referential constraint on a column, you must haveREFERENCES Privilege on the referenced table or column. The syntax is as follows:

REFERENCES [*schema***.**]*tablename* [**(***columnname***)**]
[**ON DELETE** *action*] [**ON UPDATE** *action*]

The following table describes each parameter of the syntax.

Parameter	Description
*schema***.**	An identifier specifying the name of an existing schema, followed by a . (period). If you do not specify a schema name, *tablename* is assumed to be in the current schema.
tablename	An identifier specifying the name of an existing table or the table you are creating.
columnname	An identifier specifying the name of an existing column in *tablename*. The column can be single-valued or multivalued. It must be either the primary key, @ID, or a column with a UNIQUE constraint. If you do not specify a column name, the primary key of *tablename* is the referenced column by default.

REFERENCES Parameters

Parameter	Description	
action	The referential action to take when executing a DELETE or UPDATE statement on a referenced table, or on an association or view based on a referenced table. *action* can be one of the following:	
	CASCADE	If a row in the referenced table is deleted, the corresponding row in the referencing table is also deleted. If the referenced value is updated, the referencing value is updated with the same value. You cannot specify CASCADE in an ON UPDATE clause if the referenced column is multivalued.
	SET NULL	If a value in the referenced table is deleted or updated, the value in the referencing table is set to the null value. You cannot specify SET NULL if the referencing column is defined as NOT NULL.
	SET DEFAULT	If a value in the referenced table is deleted or updated, the value in the referencing table is set to the default value for that column.
	NO ACTION	A DELETE or UPDATE on a referenced table has no effect on referencing tables.

REFERENCES Parameters (Continued)

A referential constraint defines a dependent relationship between one column (the referencing column) and another column (the referenced column). The referencing column becomes a foreign key. The referenced column is a primary key, @ID, or other column containing unique values. Only values contained in the referenced column, or null values, can be inserted into the foreign key column.

Every nonnull value written to the foreign key column must have a corresponding value in the referenced column. Note that the referenced column cannot contain null or duplicate values, but the referencing column can contain both.

The foreign key column can be singlevalued or multivalued. The referenced column can also be singlevalued or multivalued.

If you specify both the ON DELETE and ON UPDATE clauses, it does not matter which clause you specify first. For any pair of referenced and referencing tables, you can define only one ON DELETE clause and only one ON UPDATE clause that specify CASCADE, SET NULL, or SET DEFAULT.

When the referential actions CASCADE, SET NULL, and SET DEFAULT change referencing tables, the changes are verified as valid according to the referencing tables' integrity constraints. If a referencing table is also a referenced table, such changes may result in other referential actions occurring in other tables.

Note*: You cannot define referential constraints (unless they are self-referential) in CREATE TABLE statements that are part of a CREATE SCHEMA statement.*

For example, you might create a table using the following statement:

```
>CREATE TABLE DEPARTMENT
SQL+(NUMBER INT PRIMARY KEY,
SQL+DEPARTMENT VARCHAR);
```

You can then create another table that references it:

```
>CREATE TABLE EMPLOYEES
SQL+(EMPNO INT PRIMARY KEY,
SQL+EMPNAME VARCHAR,
SQL+DEPTNO INT REFERENCES DEPARTMENT (NUMBER));
```

The DEPTNO column in the EMPLOYEES table is a foreign key referencing the NUMBER column in the DEPARTMENT table. You can enter a value in the DEPTNO column only if the value also exists in the NUMBER column.

Note*: When you create a referential constraint that references a table in another schema, a VOC file entry is created whose record ID has the form schemaname.tablename. These are never deleted automatically; you must find them and delete them manually. These VOC entries are easy to find because field 1 contains one of the following:*

```
File pointer created by SQL for remote access
File pointer created by SQL for Referential Integrity
```

MULTIVALUED and SINGLEVALUED

Use MULTIVALUED to define the column as multivalued if the rows in the column will contain multiple values. Use SINGLEVALUED to define the column as singl-evalued. Columns are singlevalued by default. You cannot insert multiple values into a single-valued column.

You cannot use the CONSTRAINT keyword to name the MULTIVALUED and SINGLEVALUED column constraints.

Column Synonyms

A column definition can define a synonym for a column. You can use the synonym to assign alternate format, conversion, or column heading specifications for the column. The syntax is as follows:

synonym **SYNONYM FOR** *columnname* [*output_format*]

The following table describes each parameter of the syntax.

Parameter	Description
synonym	The name of a column synonym.
columnname	An identifier specifying the name of a column defined in another column definition in the CREATE TABLE statement.
output_format	For *output_format* syntax, see "Output Format" on page 31.

Column Synonym Parameters

The column synonym is created as another entry in the dictionary of the table, but it is not added to the SICA.

This example makes ADDRESS a column synonym for the BILLTO column in the CUSTOMERS table:

```
>CREATE TABLE CUSTOMERS
SQL+(CUSTNO INT PRIMARY KEY,
SQL+BILLTO VARCHAR,
SQL+ADDRESS SYNONYM FOR BILLTO,
    .
    .
    .
```

ASSOC Clause

The ASSOC clause defines an association of multivalued columns in a table. An association is 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.

You can define more than one association. The syntax is as follows:

> **ASSOC** [**IATION**] *name* [*position*] **(***columnname* [**KEY**]
> [**,***columnname* [**KEY**]] … **)**

The following table describes each parameter of the syntax.

Parameter	Description
name	An identifier specifying the name of the association. *name* cannot be the same as another column name or association name in this table.
position	For the description of *position*, see "Positioning of Association Rows" on page 42.
columnname	An identifier specifying the name of a multivalued column to include in the association. All columns specified in an ASSOC clause must be defined in a column definition and cannot belong to another association. *columnname* cannot be a synonym.
KEY	Defines the preceding *columnname* as an association key column. Key columns must have a column constraint of NOT NULL. If you specify only one key column, it automatically has the column constraint ROWUNIQUE. If you specify two or more key columns and you do not specify any of them as ROWUNIQUE, the columns are treated as jointly rowunique at run time. You cannot specify any association keys if *position* is INSERT PRESERVING.

ASSOC Clause Parameters

Positioning of Association Rows

position specifies where to position new association rows when they are inserted into this dynamically normalized association. *position* is one of the following:

Row	Description
INSERT LAST	New association rows are put after the last association row. This is the default if you do not specify *position*.
INSERT FIRST	New association rows are put before the first association row.

Positioning of Association Rows

Row	Description
INSERT IN *columnname* BY *seq*	
	New association rows are positioned among existing association rows according to the sequential position of the value in *columnname*. *seq* specifies the sequence and is one of the following: AL\Ascending, left-justified AR\Ascending, right-justified DL\Descending, left-justified DR\ Descending, right-justified
INSERT PRESERVING	New association rows are put in the position specified by the @ASSOC_ROW column specification in an INSERT statement. If @ASSOC_ROW is not set, new association rows are put after the last association row. INSERT PRESERVING defines this association as STABLE. For more information about STABLE associations, see the INSERT, UPDATE, and DELETE statements.

Positioning of Association Rows (Continued)

This example defines the association BOUGHT. It associates the multivalued columns PRODNO and QTY and defines PRODNO as the association key.

```
>CREATE TABLE ORDERS
SQL+(ORDERNO INT PRIMARY KEY,
SQL+PRODNO INT MULTIVALUED NOT NULL ROWUNIQUE,
SQL+QTY INT MULTIVALUED,
SQL+ASSOC BOUGHT (PRODNO KEY, QTY));
```

Table Constraints

You can define table constraints for one or more columns in a table. The syntax is as follows:

[**CONSTRAINT** *name*] *table_constraint*

The following table describes each parameter of the syntax.

Parameter	Description
name	An identifier specifying the name of the table constraint. You can name the following table constraints: UNIQUE, CHECK, and FOREIGN KEY. If you do not specify *name*, the system generates one in the format UVCON_*n*.
table_constraint	One of the following: PRIMARY KEY ['*separator*'] **(***columnnames***)** UNIQUE **(***columnnames***)** CHECK **(***condition***)** FOREIGN KEY **(***key_columns***)** REFERENCES [*schema***.**] *tablename* [**(***columnnames***)**] [ON DELETE *action*] [ON UPDATE *action*] See the following sections for descriptions of each table constraint.

Table Constraint Parameters

PRIMARY KEY Table Constraint

Use PRIMARY KEY to specify that the combination of values in a set of columns is the primary key of the table. The syntax is as follows:

PRIMARY KEY ['*separator*'] **(***columnnames***)**

separator is a single character used in the record ID field of the table to separate the values of a multicolumn primary key. The default *separator* is a text mark (CHAR(251)). Except for the text mark, *separator* must be a member of the 7-bit character set (except ASCII NUL (CHAR(0))). If you specify *separator*, you must also specify at least two *columnnames*.

columnnames is one or more names of columns defined in the CREATE TABLE statement, separated by commas.

A table can have only one primary key, and a CREATE TABLE statement can have only one PRIMARY KEY specification. All columns specified as the primary key must be single-valued and cannot contain null values. The combination of primary key values must be unique.

This example defines the PRODNO and BRAND columns as the primary key of the INVENTORY table:

```
>CREATE TABLE INVENTORY
SQL+(PRODNO INT,
SQL+BRAND CHAR(4),
SQL+PRIMARY KEY(PRODNO, BRAND),
    .
    .
    .
```

UNIQUE Table Constraint

Use UNIQUE to specify that the combination of values in a set of columns must be unique. The syntax is as follows:

UNIQUE (*columnnames***)**

columnnames is one or more names of columns defined in the CREATE TABLE statement, separated by commas.

If a UNIQUE constraint specifies only one column, the column can be either singl-evalued or multivalued. If a UNIQUE constraint specifies more than one column, all of the columns must be single-valued. All columns specified in a UNIQUE table constraint must also have the NOT NULL constraint.

You need write permissions on the directory to define the UNIQUE constraint.

This example defines the LNAME, FNAME, and MINIT columns as jointly unique:

```
>CREATE.TABLE PERSONNEL
SQL+(SSN INT PRIMARY KEY,
SQL+LNAME VARCHAR NOT NULL,
SQL+FNAME VARCHAR NOT NULL,
SQL+MINIT CHAR(1) NOT NULL,
SQL+UNIQUE(LNAME, FNAME, MINIT),
    .
    .
    .
```

CHECK Table Constraint

Use CHECK to specify criteria data must meet to be included in the column or set of columns. The syntax is as follows:

CHECK (*condition***)**

All columns included in *condition* must be in the same table.

If *condition* includes columns in an association, you must put the CHECK clause after the ASSOC Clause that defines the association.

This example defines a table constraint named DATECK. The constraint specifies that the order number must be greater than 10,000 and the order date must not be before January 1, 1994.

```
>CREATE TABLE ORDERS
SQL+(ORDERNO INT PRIMARY KEY,
SQL+"DATE" DATE CONV 'D2-',
SQL+CONSTRAINT DATECK CHECK (ORDERNO > 10000
SQL+AND "DATE" >= '1-1-94'),
    .
    .
    .
```

FOREIGN KEY Table Constraint

Use FOREIGN KEY to define a referential table constraint. To put a referential constraint on columns, you must have REFERENCES Privilege on the referenced table or columns.The syntax is as follows:

FOREIGN KEY (*key_columns***) REFERENCES** [*schema***.**]*tablename* [**(***columnnames***)**] [**ON DELETE** *action*] [**ON UPDATE** *action*]

The following table describes each parameter of the syntax.

Parameter	Description
key_columns	Identifiers specifying the names of one or more columns defined in the CREATE TABLE statement, separated by commas. If you specify more than one column, they must all be singlevalued. The number of column names in the FOREIGN KEY clause must be the same as the number of column names in the REFERENCES clause. The first column name in the FOREIGN KEY clause corresponds to the first column name in the REFERENCES clause, and so on. Data types of corresponding columns must be compatible.
*schema***.**	An identifier specifying the name of an existing schema, followed by a . (period). If you do not specify a schema name, *tablename* is assumed to be in the current schema.

FOREIGN KEY Table Constraint Parameters

Parameter	Description	
tablename	An identifier specifying the name of an existing table or the table you are creating.	
columnnames	An identifier specifying one or more names of existing columns in *tablename*, separated by commas. If you specify more than one column, they must all be singlevalued. The columns must be either the columns that make up the primary key or columns that make up a UNIQUE constraint. If you do not specify any column names, the primary key of *tablename* is the referenced column by default.	
	The number of column names must be the same as the number of column names in the FOREIGN KEY clause.	
action	The referential action to take when executing a DELETE or UPDATE statement on a referenced table, or on an association or view based on a referenced table. *action* can be one of the following:	
	CASCADE	If a row in the referenced table is deleted, the corresponding row in the referencing table is also deleted. If the referenced value is updated, the referencing value is updated with the same value.
		You cannot specify CASCADE in an ON UPDATE clause if the referenced column is multivalued.
	SET NULL	If a value in the referenced table is deleted or updated, the value in the referencing table is set to the null value.
		You cannot specify SET NULL if the referencing column is defined as NOT NULL.
	SET DEFAULT	If a value in the referenced table is deleted or updated, the value in the referencing table is set to the default value for that column.
	NO ACTION	A DELETE or UPDATE on a referenced table has no effect on referencing tables.

FOREIGN KEY Table Constraint Parameters (Continued)

This table constraint defines a dependent relationship between one column or set of columns (the referencing column) and another column or set of columns (the referenced column). The referencing column becomes a foreign key. The referenced column is a primary key, @ID, or some other column containing unique values. Only values contained in the referenced column, or null values, can be inserted into the foreign key column.

Every nonnull value written to the foreign key column must have a corresponding value in the referenced column. Note that the referenced columns cannot contain null or duplicate values, but the referencing columns can contain both.

If you define only one column as a foreign key, whether it is singlevalued or multivalued, the referenced column can also be singlevalued or multivalued. If you define several columns as a foreign key, they must be singlevalued and the corresponding referenced columns must also be singlevalued. You cannot include a multivalued column in a multicolumn foreign key.

If you specify both the ON DELETE and ON UPDATE clauses, it does not matter which clause you specify first. For any pair of referenced and referencing tables, you can define only one ON DELETE clause and only one ON UPDATE clause that specify CASCADE, SET NULL, or SET DEFAULT.

When the referential actions CASCADE, SET NULL, and SET DEFAULT change referencing tables, the changes are verified as valid according to the referencing tables' integrity constraints. If a referencing table is also a referenced table, such changes may result in other referential actions occurring in other tables.

***Note**: You cannot define referential constraints (unless they are self-referential) in CREATE TABLE statements that are part of a CREATE SCHEMA statement.*

This example puts referential constraints on the PRODNO and BRAND columns of the ORDERS table. These columns reference the PRODNO and BRAND columns of the INVENTORY table. The referenced columns are not explicitly named in the REFERENCES clause because they are the primary key of the INVENTORY table. A foreign key references the primary key of the referenced table by default.

```
>CREATE TABLE ORDERS
SQL+(ORDERNO INT PRIMARY KEY,
SQL+"DATE" DATE CONV 'D2/',
SQL+PRODNO INT NOT NULL,
SQL+BRAND CHAR(4) NOT NULL,
SQL+FOREIGN KEY (PRODNO, BRAND) REFERENCES INVENTORY,
    .
    .
    .
```

***Note**: When you create a referential constraint that references a table in another schema, a VOC file entry is created whose record ID has the form schemaname.tablename. These are never deleted automatically; you must find them and delete them manually. These VOC entries are easy to find because field 1 contains one of the following:*

```
    File pointer created by SQL for remote access
    File pointer created by SQL for Referential Integrity
```

CREATE TRIGGER

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.

Syntax

CREATE TRIGGER *triggername* { **BEFORE** | **AFTER** } *event* [**OR** *event*] …
ON *tablename* **FOR EACH ROW CALLING '***program***';**

Parameters

The following table describes each parameter of the syntax.

Parameter	**Description**
triggername	An identifier specifying the name of the trigger.
BEFORE	Specifies that the trigger program is fired before each row is written or deleted.
AFTER	Specifies that the trigger program is fired after each row is written or deleted.

event	Specifies the operation that fires the trigger program. *event* is one of the following:
	INSERT
	UPDATE
	DELETE

tablename	An identifier specifying the name of the table whose trigger you are creating.
program	Specifies the name of the trigger program.

CREATE TRIGGER Parameters

Description

A trigger specifies actions to perform before or after each row is changed by certain triggering events (SQL statements or BASIC I/O operations).

You can create triggers only for tables. You cannot create triggers for associations, views, or UniVerse files that are not tables. You can define up to six triggers for a table. The names of all triggers and their corresponding BASIC programs are stored in the table's SICA.

Trigger programs run with the SQL privileges of the trigger's creator, not with the privileges of the user whose action fires the trigger.

Triggers fired BEFORE an INSERT or UPDATE event are executed before integrity checks are performed. BEFORE triggers can change the data before it is written. They can also reject and roll back the current SQL statement or BASIC operation by using the **SetDiagnostics** function to set an error condition.

Triggers fired AFTER an INSERT, UPDATE, or DELETE event are executed after the changed data has been written to the transaction cache. AFTER triggers typically change related rows, audit database activity, and print or send messages, because at this point the full details of the row are known, the change has passed all tests, and the database can be updated.

The trigger program must be a subroutine with the following 14 arguments specified in the following order:

trigger.name
schema
table
event
time
new.recordID
new.record
old.recordID
old.record
association
association.event
count
chain.cascade
cascade

***Note**: Do not create a trigger while a program using the table is running. Results can be unpredictable.*

For more information about writing trigger programs, see Chapter 5, “Triggers.”

Example

This example creates the trigger AUDIT_EMPLOYEES on the EMPLOYEES table. It executes the AUDIT program whenever an INSERT, UPDATE, or DELETE event changes the table.

```
>CREATE TRIGGER AUDIT_EMPLOYEES
SQL+BEFORE INSERT OR UPDATE OR DELETE
SQL+ON EMPLOYEES FOR EACH ROW CALLING 'AUDIT';
```

CREATE VIEW

Use the CREATE VIEW statement to create a virtual table in the schema you are logged in to, derived from other tables and views. To create a view, you need SELECT Privilege on the underlying tables and views. To grant SELECT, INSERT, UPDATE, and DELETE privileges on a view you own to others, you must have the privileges WITH GRANT OPTION on all underlying tables and views.

Syntax

CREATE VIEW *viewname* [**(***columnnames***)**] **AS** *query_expression* [**WITH** [**LOCAL** | **CASCADED**] **CHECK OPTION**]**;**

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
viewname	An identifier specifying the name of the view. *viewname* must not exist in the schema or be the ID of a record in the VOC file. The current account directory must not contain a file named *viewname* or D_*viewname*.
columnnames	Identifiers specifying the names of the columns in the view, separated by commas. The number of columns in *column-names* must be the same as the number of columns in the SELECT clause of the query specification. That is, if you name *any* of the view's columns, you must name *all* columns specified in the SELECT clause.
query_expression	The SELECT statement that creates the view.

CREATE VIEW Parameters

Parameter	Description
WITH CHECK OPTION	Specifies that any change made to the view must fulfill the conditions specified in *query_expression*. If you do not specify LOCAL or CASCADED, the change must also fulfill the conditions specified in all views underlying this one. The view must be updatable if you specify WITH CHECK OPTION.
LOCAL	Specifies that only the WHERE clause of the view defined in *query_expression* is checked when an INSERT or UPDATE statement is executed.
CASCADED	Specifies that the WHERE clause of the view defined in *query_expression* as well as in all underlying views are checked when an INSERT or UPDATE statement is executed. This is the default action.

CREATE VIEW Parameters (Continued)

Description

CREATE VIEW defines a virtual table derived from one or more base tables or views. CREATE VIEW also creates a file dictionary for the view. A view behaves in many ways like a table created with the CREATE TABLE statement.

Updatable Views

A view is updatable if the following conditions in the query expression are met:

- At least one item in the SELECT clause must be a simple reference to a column in the base table, or at least one condition in the WHERE clause must state that a primary key column in the base table (or @ID if the base table has no primary key) equals some value.
- The SELECT Clause does not contain the keyword DISTINCT.
- The FROM clause:
  - Identifies only one table
  - Does not contain an UNNEST Clause

- The table expression:
  - Identifies a base table or an updatable view
  - Does not contain the keywords JOIN or UNION
  - Does not reference an association of multivalued columns or an unassociated multivalued column as a dynamically normalized table
- The WHERE Clause does not contain a subquery to the same table.
- There is no WHEN Clause, GROUP BY Clause, or HAVING Clause.

You can use the INSERT, UPDATE, DELETE, and SELECT statements against an updatable view. You can use only the SELECT statement against a read-only view.

***Note**: Read-only views created before Release 9.3.1 must be recreated to make them updatable. Nested views must be recreated from the bottom up.*

Inherited Associations

A view inherits an association from the table or view it is derived from. Any columns belonging to the association in the underlying table or view are associated in the view. The view definition determines the order of the associated columns in the view.

If the view is updatable, it inherits an association from the table or view it is derived from only if the following conditions are met:

- All parts of the base table's primary key (or @ID if the base table has no primary key) are referenced in the query expression.
- Each column in the association is referenced by its simple column name.
- If the association has keys, all association keys are referenced in the SELECT clause.
- If the association has no keys, all columns in the association are referenced in the SELECT Clause.

Query Expression

The query expression is a standard UniVerse SQL SELECT statement, except for the following:

- Field modifiers (AVG, BREAK ON, BREAK SUPPRESS, CALC, PCT, TOTAL, and their synonyms are not allowed.

- The field qualifiers SINGLEVALUED, MULTIVALUED, ASSOC, ASSOCIATED, and their synonyms are not allowed.
- Report qualifiers (HEADER, FOOTER, and so on) are not allowed.
- The TO SLIST clause is not allowed.
- The FROM clause cannot specify a UniVerse file or a table that is a part file of a distributed file.
- The ORDER BY clause is not allowed.

Specifying Rows Explicitly

When you create a view, you can use the FROM clause of the query specification to specify the rows you want the view to include. The FROM clause can include any of the following:

- A list of explicit primary keys
- An SLIST clause
- The INQUIRING keyword

Explicit Primary Keys

You can specify explicit primary keys in the CREATE VIEW statement even if the primary keys do not currently exist in the base table. Only the specified rows that exist when users access the view are included in the view.

SLIST Clause

You can specify a numbered select list in the CREATE VIEW statement even if the specified select list does not currently exist. When users access the view, the specified select list must exist, otherwise no view is generated.

INQUIRING Keyword

When you specify the INQUIRING keyword in the CREATE VIEW statement, no prompts appear asking you to specify primary keys. When users access the view, they are prompted to enter the primary keys they want to include.

Columns in a View

In a view, a column's name, data type, heading, format, and conversion are derived from the columns in the underlying tables and views. The width of a column is determined by either the width of the column heading or the width defined by a FMT clause, whichever is larger.

Column Names

Since the names of a view's columns are derived from the columns in the underlying tables and views, you do not need to name them explicitly in the CREATE VIEW statement, unless the SELECT clause in the query specification:

- Uses a select expression or EVAL expression to define a virtual column
- Specifies a set function to define a virtual column
- Specifies duplicate column names (even if they are qualified)

In these cases you must specify column names. Column names must be unique in a view. You can explicitly name a view's columns in two ways:

- Specify *columnnames* after *viewname* in the CREATE VIEW statement.
- Specify a column alias for the expression, set function, or duplicate column name in the query expression.

Column Heading

The column heading is derived from one of the following, in order of precedence:

- A DISPLAYNAME clause
- The column or column alias named in a DISPLAYLIKE clause
- A column alias
- A select expression or column name

Format

The column's format is derived from a FMT clause, if there is one, or from the column or column alias named in a DISPLAYLIKE clause. Default column formats are as follows:

Column	**Format**
Column name	As defined in the format field of the dictionary.
EVAL expression	10L, unless the dictionary entry for the leftmost column defined in the EVAL expression contains an MD conversion or specifies right justification, in which case the format is 10R.
Set function	10R for COUNT and AVG of a column whose data type is INT or SMALLINT. For MAX, MIN, SUM, and AVG of a column whose data type is not INT or SMALLINT: as defined in the format field of the dictionary.
Select expression	11R for an expression whose data type is DATE. 8R for an expression whose data type is TIME. 10R for numeric expression of any other data type.
Constant	10T for a character expression. 8L if the constant is USER. 1L if the constant is NULL.

Column Formats

Conversion

The column's conversion is derived from one of the following, in order of precedence:

- A CONV clause
- The conversion field in the column or alias named in a DISPLAYLIKE clause
- For columns, the conversion field of the dictionary
- For expressions, an appropriate conversion code for the data type

Examples

This example creates a view that includes all the columns from the INVENTORY table. The column names in INV_VIEW are the same as the names of the columns in the INVENTORY table.

```
>CREATE VIEW INV_VIEW AS SELECT * FROM INVENTORY;
Creating View "INV_VIEW"
Adding Column "PROD.NO"
Adding Column "DESC"
Adding Column "QOH"
Adding Column "COST"
Adding Column "SELL"
Adding Column "MARKUP"
```

If you subsequently add columns to the underlying table using ALTER TABLE, this view is unaffected by the additional columns. That is, the view in the preceding example still contains six columns.

The next example creates a view PRODQTY_VIEW that includes only two columns, PRODNO and QOH, from the INVENTORY table:

```
>CREATE VIEW PRODQTY_VIEW AS SELECT PRODNO, QOH FROM INVENTORY;
```

The next example creates a view DEPTINFO based on the DEPTS table in the OTHERSCHEMA schema:

```
>CREATE VIEW DEPTINFO (DEPTNAME, DEPTHEAD)
SQL+AS SELECT DEPTNAME, MANAGER FROM OTHERSCHEMA.DEPTS;
```

The query specification in the next example extracts the columns ORDERS.ORDERNO and ITEMS.ORDERNO from the ORDERS and ITEMS tables. You must specify unique, corresponding column names (CUSTNO and OCUSTNO) in the CREATE VIEW statement. Since this query specification specifies a virtual column resulting from a calculation (ITEMS.TOTPRICE * 1.25), you must also name the corresponding column in the view (NEWPRICE).

```
>CREATE VIEW SOLD_VIEW (CUSTNO, OCUSTNO, NEWPRICE)
SQL+AS SELECT ORDERS.ORDERNO, ITEMS.ORDERNO,
SQL+ITEMS.TOTPRICE * 1.25
SQL+FROM ORDERS, ITEMS
SQL+WHERE ORDERS.ORDERNO = ITEMS.ORDERNO
SQL+AND ITEMS.TOTPRICE > 150;
```

In the next example the CREATE VIEW statement uses SLIST 0 to create the ORD_VIEW view. When you use a SELECT statement to access the view, SLIST 0 must be active.

```
>CREATE VIEW ORD_VIEW AS SELECT * FROM ORDERS SLIST 0;
Creating View "ORD_VIEW"
Adding Column "ORDER.NO"
Adding Column "CUST.NO"
Adding Column "PROD.NO"
Adding Column "QTY"
Adding Column "ORDER.TOTAL"
Adding association "BOUGHT"
>SELECT TO SLIST 0 FROM ORDERS;

7 record(s) selected to SELECT list #0.
>>SELECT * FROM ORD_VIEW;
Order No    Customer No    Product No    Qty.    Total.......

10002              6518           605       1         $55.00
                                  501       1
                                  502       1
                                  504       1
10006              6518           112       3         $18.00
10004              4450           704       1        $205.00
                                  301       9
10005              9874           502       9         $45.00
10003              9825           202      10        $100.00
                                  204      10
10001              3456           112       7        $265.00
                                  418       4
                                  704       1
10007              9874           301       3         $30.00

7 records listed.
```

The next example uses the INQUIRING keyword to create the ORD_VIEW view. When you use a SELECT statement to access the view, you are prompted to enter primary keys.

```
>CREATE VIEW ORD_VIEW AS SELECT * FROM ORDERS INQUIRING;
Creating View "ORD_VIEW"
Adding Column "ORDER.NO"
Adding Column "CUST.NO"
Adding Column "PROD.NO"
Adding Column "QTY"
Adding Column "ORDER.TOTAL"
Adding association "BOUGHT"
>SELECT * FROM ORD_VIEW;
Primary Key for table ORD_VIEW = 10002

Order No    Customer No    Product No    Qty.    Total.......

10002              6518           605       1         $55.00
                                  501       1
                                  502       1
                                  504       1

Primary Key for table ORD_VIEW =
```

DELETE

Use the DELETE statement to remove rows from a table, view, or UniVerse file. You must own the table or view, or have DELETE Privilege on it, to use the DELETE statement. To delete rows from a view, you must also have DELETE privilege on the underlying base table and any underlying views. You cannot use DELETE on a type 1, type 19, or type 25 file.

Syntax

DELETE FROM *table_expression* [WHERE clause] [*qualifiers*];

Parameters

The following table describes each parameter of the syntax.

Parameter	Description	
table_expression	Specifies the table or view whose rows you want to delete. For the syntax of *table_expression*, see "Table." If *table_expression* references an association of multivalued columns or an unassociated multivalued column as a dynamically normalized table, only data in the selected association rows is deleted.	
WHERE clause	Specifies the criteria that data in the rows must meet for the rows to be deleted. If you omit the WHERE clause, all rows in the table are deleted (except when *table_expression* specifies a subset of rows to delete).	
qualifiers	One or more of the following processing qualifiers separated by spaces:	
	EXPLAIN	Lists the tables referenced by the DELETE statement and explains how the query optimizer will handle execution of the statement.

DELETE Parameters

Parameter	Description	
	NO.OPTIMIZE	Suppresses the optimizer when processing the WHERE clause.
	NOWAIT	If the DELETE statement encounters a lock set by another user, it terminates immediately. It does not wait for the lock to be released.
	REPORTING	Displays the primary key of each deleted row. If the table has no primary key, displays the value in @ID.

DELETE Parameters (Continued)

Description

If you try to delete more than one row and the DELETE statement fails (due to a constraint violation, for example), no rows are deleted.

If you delete an association row from a dynamically normalized association that is defined as STABLE, the @ASSOC_ROW values of the remaining association rows stay the same.

If you use the DELETE statement on a table that is referenced by a foreign key, the deleted data must meet the constraint criteria. You also need write permissions on the directory and the VOC file in the account.

If you delete rows in a referenced table, rows in the referencing table may also be deleted or changed, depending on the referential constraint action prescribed by the referencing table's REFERENCES clause. The following table shows what happens when the referencing table's REFERENCES clause contains the ON DELETE CASCADE clause:

If the referenced column is...	**And if the referencing column is...**	**The following referential integrity action occurs:**
Singlevalued	Singlevalued	The row in the referencing table corresponding to the deleted row in the referenced table is also deleted. If the corresponding columns are parts of a multipart column set, all corresponding part-columns must match for the corresponding rows in the referencing table to be deleted.
Multivalued	Singlevalued	The row in the referencing table corresponding to the deleted association row in the referenced table is also deleted.
Singlevalued	Multivalued	The association row in the referencing table corresponding to the deleted row in the referenced table is also deleted.
Multivalued	Multivalued	The association row in the referencing table corresponding to the deleted association row in the referenced table is also deleted.

ON DELETE CASCADE Clause Results

The following table shows what happens when the referencing table's REFER-ENCES clause contains the ON DELETE SET NULL or ON DELETE SET DEFAULT clause:

If the referenced column is...	**And if the referencing column is...**	**If the referencing table, either a null value or the column's default value replaces...**
Singlevalued	Singlevalued	The value in the referencing column corresponding to the deleted row in the referenced table.
Multivalued	Singlevalued	The value in the referencing column corresponding to the deleted association row in the referenced table.
Singlevalued	Multivalued	The multivalue in the referencing column corresponding to the deleted row in the referenced table.
Multivalued	Multivalued	The multivalue in the referencing column corresponding to the deleted association row in the referenced table.

ON DELETE SET NULL or ON DELETE SET DEFAULT Clause Results

Using EXPLAIN

The EXPLAIN keyword lists all tables referenced by the DELETE statement, including tables referenced by subqueries in the WHERE clause, and explains how the query optimizer will use indexes, process joins, and so forth, when the statement is executed. If the WHERE Clause includes subqueries, information is given about each query block.

If you use EXPLAIN in an interactive DELETE statement, after viewing the EXPLAIN message, press **Q** to quit, or press any other key to continue processing.

If you use EXPLAIN in a DELETE statement executed by a client program, the statement is not processed. Instead, an SQLSTATE value of IA000 is returned, along with the EXPLAIN message as the message text.

Using NOWAIT

The NOWAIT condition applies to:

- All locks encountered by the DELETE statement
- All cascaded updates and deletes that result from the DELETE statement
- All SQL operations in trigger programs fired by the DELETE statement

In these cases, the DELETE statement and all its dependent operations are terminated and rolled back, and an SQLSTATE of 40001 is returned to client programs.

Examples

This example deletes all rows from the CUST table:

```
>DELETE FROM CUST;
```

The next example deletes two rows from the EMPS table:

```
>DELETE FROM EMPS '2727' '2728';
```

The next example uses the keyword INQUIRING to prompt you for the primary key of the row you want to delete from CUSTOMERS:

```
>DELETE FROM CUSTOMERS INQUIRING;


Record =
```

The next example uses the REPORTING keyword to list the primary keys of the deleted rows:

```
>DELETE FROM CUSTOMERS REPORTING;
UniVerse/SQL: Record "4450" deleted.
UniVerse/SQL: Record "7230" deleted.
UniVerse/SQL: Record "9480" deleted.
UniVerse/SQL: Record "3456" deleted.
UniVerse/SQL: Record "6518" deleted.
UniVerse/SQL: Record "9874" deleted.
UniVerse/SQL: Record "9825" deleted.
UniVerse/SQL: Record "1001" deleted.
UniVerse/SQL: Record "1043" deleted.
UniVerse/SQL: Record "2309" deleted.
UniVerse/SQL: 10 records deleted.
```

This example deletes rows where any value in the multivalued PROD.NO column is 102:

```
>DELETE FROM ORDERS
SQL+WHERE PROD.NO = 102;
```

The next example deletes all the rows of the EMPS table for sales personnel where the sales are below average:

```
>DELETE FROM EMPS
SQL+WHERE DEPTCODE = 'Sales'
SQL+AND SALES < (SELECT AVG(SALES) FROM EMPS);
```

The next example removes rows whose department number is 209 from the DEPTS table in the OTHERSCHEMA schema:

```
>DELETE FROM OTHERSCHEMA.DEPTS
SQL+WHERE OTHERSCHEMA.DEPTS.DEPTNO = 209;
```

The next example removes a line item from an order in the ORDERS table. Line items in the ORDERS table are association rows.

```
>DELETE FROM ORDERS_BOUGHT
SQL+WHERE ORDER.NO = '10001'
SQL+AND PROD.NO = 888;

UniVerse/SQL: 1 record deleted.
```

ORDER.NO is the primary key and PROD.NO is the key of the association BOUGHT.

DROP INDEX

Use the DROP INDEX statement to delete a secondary index created by the CREATE INDEX statement. You must be the owner of the table, have ALTER Privilege on it, or have DBA Privilege to drop a table's indexes.

Syntax

DROP INDEX *tablename*.*indexname*;

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
tablename	An identifier specifying the name of a table in the schema you are logged in to whose indexes you want to drop.
indexname	An identifier specifying the name of the index you want to drop.

DROP INDEX Parameters

Description

DROP INDEX removes the specified secondary index from the table.

Note*: You cannot use DROP INDEX on secondary indexes created by the UniVerse command CREATE.INDEX.*

Example

This example drops the secondary indexes from the VENDORS.T table:

```
>DROP INDEX VENDORS.T.VENDORS_INDEX;
Dropping Index "VENDORS_INDEX".
```

DROP SCHEMA

Use the DROP SCHEMA statement to delete a schema. You must be the owner of the schema or have DBA Privilege to drop it. You must issue the DROP SCHEMA statement from a schema other than the one you want to drop.

Syntax

DROP SCHEMA *schema* [**CASCADE**]**;**

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
schema	An identifier specifying the name of the schema you want to drop.
CASCADE	Drops all SQL tables and views in *schema*. You must specify CASCADE if the schema has any SQL tables.

DROP SCHEMA Parameters

Description

DROP SCHEMA removes the specified schema and all its tables and views from the SQL catalog. You cannot drop a schema if any of its tables are referenced by tables in other schemas.

***Note**: You must first drop any referential constraints defined in other schemas that refer to tables in schema before using DROP SCHEMA CASCADE to drop schema.*

Depending on whether the schema was created in a non-UniVerse directory or was converted from an existing UniVerse account, DROP SCHEMA has different results:

Schema was created as...	**Schema has SQL tables**	**Using DROP SCHEMA without CASCADE...**	**Using DROP SCHEMA with CASCADE...**
A new schema in a non-UniVerse directory	No	Deletes all UniVerse files created by CREATE SCHEMA. Does not delete UniVerse data files.	Deletes all UniVerse files created by CREATE SCHEMA. Does not delete UniVerse data files.
	Yes	Displays a message telling you to use CASCADE.	Deletes all SQL tables and UniVerse files created by CREATE SCHEMA. Does not delete UniVerse data files.
A converted UniVerse account	No	Turns schema back into a normal UniVerse account, deleting nothing.	Turns schema back into a normal UniVerse account, deleting nothing.
	Yes	Displays a message telling you to use CASCADE.	Deletes all SQL tables and turns schema back into a normal UniVerse account.

DROP SCHEMA Results

Example

This example drops the schema Susan, including all its tables and views:

```
>DROP SCHEMA Susan CASCADE;
```

DROP TABLE

Use the DROP TABLE statement to delete a table. You must be the owner of the table or have DBA Privilege to drop it.

Syntax

DROP TABLE *tablename* [**CASCADE**]**;**

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
tablename	An identifier specifying the name of the table you want to drop. The table must be in the schema you are logged in to.
CASCADE	Drops all views derived from *tablename*. You must specify CASCADE if the table has views depending on it.

DROP TABLE Parameters

Description

DROP TABLE removes the table and all views depending on it from the SQL catalog. It also deletes the UniVerse data file, its associated dictionary, and any secondary indexes.

Table has dependent views	Using DROP TABLE without CASCADE...	Using DROP TABLE with CASCADE...
No	Deletes the table.	Deletes the table.
Yes	Displays a message telling you to use CASCADE.	Deletes the table and all its dependent views.

DROP TABLE Results

DROP TABLE automatically revokes all privileges on the table.

You cannot drop a table that is a part file of a distributed file or that has a partitioning algorithm attached to it. To drop such a table, use the DEFINE.DF command with the CANCEL option to remove the partitioning algorithm, then drop the table.

You cannot drop a table that is referenced by a foreign key constraint in another table. To drop such a table, first drop any referential constraints in the referencing table, then drop the referenced table.

Example

This example drops the table CUSTOMERS, including all views depending on it:

```
>DROP TABLE CUSTOMERS CASCADE;
```

DROP TRIGGER

Use the DROP TRIGGER statement to drop a trigger created by the CREATE TRIGGER statement. You must be the table's owner or have ALTER Privilege on it, or you must be a DBA to drop a table's triggers.

Syntax

DROP TRIGGER *tablename* { *triggername* | **ALL** } **;**

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
tablename	An identifier specifying the name of the table whose trigger you want to drop.
triggername	An identifier specifying the trigger you want to drop.
ALL	Drops all of a table's triggers.

DROP TRIGGER Parameters

Description

When you drop a trigger, its name is removed from the table's SICA, but the corresponding BASIC program is not deleted.

Do not drop a trigger while a program using the table is running. Results may be unpredictable.

Example

This example drops the AUDIT_EMPLOYEES trigger from the EMPLOYEES table:

```
>DROP TRIGGER EMPLOYEES AUDIT_EMPLOYEES;
```

DROP VIEW

Use the DROP VIEW statement to delete a view. You must be the owner of the view or have DBA Privilege to drop it.

Syntax

DROP VIEW *viewname* [**CASCADE**]**;**

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
viewname	An identifier specifying the name of the view you want to drop. The view must be in the schema you are logged in to.
CASCADE	Drops all views derived from *viewname*. You must specify CASCADE if the view has views depending on it.

DROP VIEW Parameters

Description

DROP VIEW removes the view from the SQL catalog. It also deletes its associated dictionary.

View has dependent views	Using DROP VIEW without CASCADE...	Using DROP VIEW with CASCADE...
No	Deletes the view.	Deletes the view.
Yes	Displays a message telling you to use CASCADE.	Deletes the view and all its dependent views.

DROP VIEW Results

DROP VIEW automatically revokes all privileges on the view.

You can query the UV_TABLES table for information about names of views derived from tables or from other views.

Example

This example drops the INV_VIEW view:

```
>DROP VIEW INV_VIEW;
```

GRANT

Use the GRANT statement to assign user privileges. UniVerse SQL users can assign table privileges on tables and views they own or on which they have the right to grant specific privileges. You must be a database administrator (DBA) to assign database privileges.

Syntax

GRANT *database_privilege* **TO** *users***;**

GRANT *table_privileges* **ON** *tablename* **TO** { *users* | **PUBLIC** } [**WITH GRANT OPTION**]**;**

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
database_privilege	One of the following: CONNECT RESOURCE DBA
table_privileges	One or more of the following, separated by commas: SELECT INSERT UPDATE [(*columnnames*)] DELETE REFERENCES [(*columnnames*)] ALTER ALL PRIVILEGES
tablename	An identifier specifying the name of an existing table or view.

GRANT Parameters

Parameter	Description
users	One or more user names separated by commas.
PUBLIC	Specifies all UniVerse users, whether or not they are defined in the SQL catalog.
WITH GRANT OPTION	Specifies that users can grant the specified table privileges to others.

GRANT Parameters (Continued)

Description

On UNIX Systems

When UniVerse is first installed, only one user is registered as an SQL user. If there is a user named *uvsql* in the */etc/passwd* file, the owner of the CATALOG schema is set to *uvsql*, and *uvsql* is put in the UV_USERS table as the first SQL user with DBA Privilege. If no *uvsql* user is defined in */etc/passwd*, the installer is prompted to do one of the following:

- Suspend the installation process in order to register a user named *uvsql*
- Let *root* or *uvadm* be the owner of the SQL catalog

Either *uvsql*, *root*, or *uvadm*, as the database administrator, has DBA privilege. The DBA registers other UniVerse SQL users by granting them the appropriate database privileges.

On Windows Platforms

When UniVerse is first installed, only the user NT AUTHORITY\SYSTEM is registered as an SQL user. This user is put in the UV_USERS table as the first SQL user with DBA Privilege. The DBA registers other UniVerse SQL users by granting them the appropriate database privileges.

Database Privileges

There are three levels of database privilege. From lowest to highest they are:

- CONNECT

- RESOURCE
- DBA

Only a user with DBA privilege can grant database privileges to other users.

CONNECT Privilege

CONNECT privilege registers a user as a UniVerse SQL user. The user must be defined in the */etc/passwd* file. When you grant CONNECT privilege to users, they are registered in the UV_USERS table of the SQL catalog.

Users with CONNECT privilege can do the following:

- Create tables
- Alter and drop tables they own
- Grant and revoke privileges on tables they own
- Use the SELECT, INSERT, UPDATE, and DELETE statements on tables they have access to
- Create and drop views on tables they have access to

This example grants CONNECT privilege to users *maria* and *mark*, who are defined in the */etc/passwd* file. The GRANT statement registers them in the SQL catalog.

```
>GRANT CONNECT
SQL+TO maria, mark;
```

RESOURCE Privilege

RESOURCE privilege lets UniVerse SQL users create schemas. RESOURCE privilege includes all capabilities of the CONNECT Privilege. You can grant RESOURCE privilege only to users with CONNECT privilege.

DBA Privilege

DBA privilege lets UniVerse SQL users execute all SQL statements on all tables and files as if they owned them. It is the highest database privilege. You can grant DBA privilege only to users with CONNECT privilege. DBA privilege includes all capabilities of the RESOURCE privilege and in addition lets the database administrator do the following:

- Grant and revoke database privileges

- Create schemas and tables for other UniVerse SQL users
- Grant privileges on any table to any user
- Revoke privileges on any table from any user

Table Privileges

When you create a table, you are the only user with privileges on it, except for users with DBA privilege. The owner of a table can grant any of the following table privileges on it to other users:

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

You can use the ALL PRIVILEGES keyword to grant all six table privileges at once.

To create a view, you must have SELECT Privilege on the underlying tables or views. You can grant privileges on a view only if you are the owner of the view or if you have been granted SELECT privilege on them WITH GRANT OPTION.

SELECT Privilege

SELECT privilege lets users retrieve data from a table.

INSERT Privilege

INSERT privilege lets users add new rows to a table with the INSERT statement. UniVerse users who have INSERT and UPDATE privileges can add or change data in a table using the Editor, ReVise, or a BASIC program.

UPDATE Privilege

UPDATE privilege lets users modify existing data in a table using the UPDATE statement. UniVerse users who have the INSERT Privilege and the UPDATE Privilege can add or change data in a table using the Editor, ReVise, or a BASIC program. You can grant UPDATE privilege on all columns in a table or on specific columns. The syntax is as follows:

UPDATE [**(***columnnames***)**]

columnnames is one or more column names separated by commas. If you do not specify column names, users can update all columns in the table.

DELETE Privilege

DELETE privilege lets users delete rows from a table.

REFERENCES Privilege

REFERENCES privilege lets users define referenced columns in a referential constraint. You can specify REFERENCES privilege on all columns in a table or on specific columns. The syntax is as follows:

REFERENCES [**(***columnnames***)**]

columnnames is one or more column names separated by commas. If you do not specify column names, users can specify all columns in the table as referenced columns.

ALTER Privilege

ALTER privilege lets users change the structure of a table. Users with ALTER privilege can do the following:

- Add columns to a table
- Add and remove table constraints
- Add and remove association definitions
- Set and remove default specifications

Examples

This example grants SELECT, UPDATE, and DELETE privileges on the CUSTOMERS table to users *maria* and *mark*:

```
>GRANT SELECT, UPDATE, DELETE
SQL+ON CUSTOMERS
SQL+TO maria, mark;
```

The next example gives *maria* the ability to grant SELECT privilege on the CUSTOMERS table to others:

```
>GRANT SELECT
SQL+ON CUSTOMERS
SQL+TO maria
SQL+WITH GRANT OPTION;
```

The next example gives *mark* the ability to update the CUSTNO, FNAME, and LNAME columns of the CUSTOMERS table:

```
>GRANT UPDATE (CUSTNO, FNAME, LNAME)
SQL+ON CUSTOMERS
SQL+TO mark;
```

INSERT

Use the INSERT statement to insert new rows into a table, view, or UniVerse file. To insert data into a table or view, you must own it or have INSERT Privilege on it. To insert data into a view, you must also have INSERT privilege on the underlying base table and any underlying views. You cannot use INSERT to insert data into a type 1, type 19, or type 25 file.

Syntax

INSERT INTO *table_expression* [(*columnnames*)] *values* [*qualifiers*];

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
table_expression	Specifies the table or view into which to insert new data. For the syntax of *table_expression*, see “Table.” If *table_expression* references an association of multivalued columns or an unassociated multivalued column as a dynamically normalized table, INSERT inserts a new association row into specified base table rows.
columnnames	Identifiers specifying the names of one or more columns in the table or view, separated by commas. If you do not specify *columnnames*, all columns are assumed. If *table_expression* references an association of multivalued columns or an unassociated multivalued column as a dynamically normalized table, you can use the @ASSOC_ROW keyword to specify unique association row keys if the association has no keys.
values	Values to insert in the columns. *values* can be a VALUES Clause or a query specification.
qualifiers	One or more of the following processing qualifiers separated by spaces:

INSERT INTO Parameters

Parameter	Description	
	EXPLAIN	Lists the tables referenced by a query specification in an INSERT statement and explains how the query optimizer will handle execution of the query specification.
	NO.OPTIMIZE	Suppresses the optimizer when processing a query specification.
	NOWAIT	If the INSERT statement encounters a lock set by another user, it terminates immediately. It does not wait for the lock to be released.
	REPORTING	Displays the primary key of each inserted row. If the table has no primary key, displays the value in @ID.

INSERT INTO Parameters (Continued)

Description

The INSERT statement creates one or more new rows in a table or file. You can specify the values you want to insert in one row, or you can use a SELECT statement to insert a set of rows from another table.

If you use the INSERT statement on a table that has column or table constraints, the inserted data must meet the constraint criteria.

If you try to insert more than one row and the INSERT statement fails (due to a constraint violation, for example), no new rows are added.

You cannot insert data into the same table from which you select data.

Using EXPLAIN

The EXPLAIN keyword lists all tables referenced by a query specification in INSERT statement and explains how the query optimizer will use indexes, process joins, etc., when the statement is executed. Information is given about each query block.

If you use EXPLAIN in an interactive INSERT statement, after viewing the EXPLAIN message, press **Q** to quit, or press any other key to continue processing.

If you use EXPLAIN in an INSERT statement executed by a client program, the statement is not processed. Instead, an SQLSTATE value of IA000 is returned, along with the EXPLAIN message as the message text.

Using NOWAIT

The NOWAIT condition applies to:

- All locks encountered by the INSERT statement
- All cascaded updates and deletes that result from the INSERT statement
- All SQL operations in trigger programs fired by the INSERT statement

In these cases the INSERT statement and all its dependent operations are terminated and rolled back, and an SQLSTATE of 40001 is returned to client programs.

Specifying Columns

Tables

You can specify the columns into which you want to insert values. The order and number of the values to insert must be the same as the order and number of the column names you specify. If you specify fewer columns than the table contains, the unnamed single-valued columns are filled with default values, and unnamed multi-valued columns are left empty. If you specify no columns, you must specify values for all columns in the table.

If several column names map to the same column in a view's underlying base table or view, the last column name in the list determines the value of the column.

Values are inserted into columns in the order in which the columns were defined or last altered.

This example inserts a row into the ORDERS table. The primary key is 10009. All other columns are filled with default values.

```
>INSERT INTO ORDERS (ORDER.NO) VALUES (10009);
UniVerse/SQL: 1 record inserted.
```

UniVerse Files

When you insert data into a UniVerse file, the order and number of the values to insert must be the same as the order and number of the column names you specify. Unnamed columns are left empty.

If you specify no columns, the file dictionary must contain an @INSERT phrase. (An @INSERT phrase in a table dictionary is ignored.) The @INSERT phrase defines the fields into which INSERT can insert values and the order in which to insert them. The order and number of values to insert must be the same as the order and number of fields in the @INSERT phrase. If the @INSERT phrase specifies fewer fields than are defined in the file dictionary, the unnamed fields are left empty.

Dynamic Normalization

If *table_expression* is of the form *tablename_association*, the specified association of multivalued columns is treated as a virtual first-normal-form table. Using both the primary key of the base table and the key of the association, you can insert new association rows into specified base table rows. If the base table has no primary key, the values in @ID are used. If the association does not have association keys, use the @ASSOC_ROW keyword as a column name. Dynamic normalization combines the base table's primary keys with the value mark count generated by @ASSOC_ROW to create a set of jointly unique association row keys. These keys let you specify the particular association rows to insert.

If you insert an association row into a dynamically normalized association that is defined as STABLE, the @ASSOC_ROW value for the new row cannot already exist. If the @ASSOC_ROW value of the new row is higher than the highest existing @ASSOC_ROW value, empty association rows are inserted between the last existing row and the new one.

If you dynamically normalize a UniVerse file and you specify no columns, values are inserted only in the associated multivalued columns included in the @INSERT phrase.

Specifying Values

Specify *values* to insert using one of the following:

- A VALUES Clause
- A query specification

The number of values supplied must equal the number expected, as determined by the columns you specify (see Specifying Columns).

VALUES Clause

Use the VALUES clause to insert a single row into a table. The VALUES clause has two syntaxes:

> **VALUES (***valuelist***)**
>
> **DEFAULT VALUES**

valuelist is one or more values separated by commas. You can specify each value as one of the following:

Value	Description
expression	Specifies a literal; the keywords USER, CURRENT_DATE, or CURRENT_TIME; or numbers, combined using arithmetic operators or the concatenation operator. Character value expressions can include any of the function expressions. The primary key of the new row must be unique.
NULL	If the value is NULL, the column cannot have the column constraint NOT NULL. You cannot specify NULL as the value of a column that is part of a primary key.
multivalues	A set of comma-separated values enclosed in angle brackets, to insert into a multivalued column. The syntax is as follows: <*value*[,*value*…]> The angle brackets are part of the syntax and must be typed. A value can be NULL or an expression. If the column belongs to an association, the number and order of values for each row must be the same as the number and order of the corresponding values in the association key. If *table_expression* references an association of multivalued columns or an unassociated multivalued column as a dynamically normalized table, you cannot specify *multivalues* in the VALUES clause.

valuelist **Values**

You can insert a row of default values into a table by specifying DEFAULT VALUES. Default values are defined by the CREATE TABLE and ALTER TABLE statements. If no default values are defined for columns in a table, the default value is null. For columns in a UniVerse file, the empty string is the default value.

Query Specification

Use a query specification to insert multiple rows into a table. The query specification selects rows from a table, a view, or a UniVerse file. The query specification is a standard UniVerse SQL SELECT statement, except for the following:

- Field modifiers (AVG, BREAK ON, BREAK SUPPRESS, CALC, PCT, TOTAL, and their synonyms) are not allowed.
- Field qualifiers (CONV, FMT, and so on) are not allowed.
- Report qualifiers are not allowed.
- The ORDER BY Clause is not allowed.

Examples

The following example writes a new row with data in four columns into the ORDERS table. The columns ITEM and QTY are multivalued.

```
>INSERT INTO ORDERS (ORDNO, CUSTOMER, ITEM, QTY)
SQL+VALUES (2, 'STAR', <'BOLT', 'SCREW', 'HINGE'>, <33, 34, 35>);
```

The next example extracts from the EMPS table all rows with information about employees in the Sales department and inserts them into the SALESMEN table:

```
>INSERT INTO SALESMEN
SQL+SELECT * FROM EMPS
SQL+WHERE DEPTCODE = 'SALES';
```

The next example inserts a row into the DEPTS table in the remote schema OTHERSCHEMA:

```
>INSERT INTO OTHERSCHEMA.DEPTS (DEPTNO) VALUES (316);
```

The next example inserts data from the PHONE and NAME columns of the CUSTOMERS file into the PHONES table. The UNNEST clause explodes the multivalues in the PHONE field, generating a separate row for each multivalue.

```
>INSERT INTO PHONES
SQL+SELECT PHONE, NAME
SQL+FROM UNNEST CUSTOMERS ON PHONE;
```

The next example inserts a line item into an order in the ORDERS table. Line items in the ORDERS table are association rows.

```
>INSERT INTO ORDERS_BOUGHT
SQL+VALUES ('10001', 888, 5);
UniVerse/SQL: 1 record inserted.
```

10001 is the primary key, and 888 is the key of the association BOUGHT. The base table row must already exist when you are inserting an association row.

REVOKE

Use the REVOKE statement to remove privileges from users. UniVerse SQL users can revoke specific table privileges they granted on tables and views. You must be a database administrator (DBA) to revoke database privileges.

Syntax

REVOKE *database_privilege* **FROM** *users***;**

REVOKE [**GRANT OPTION FOR**] *table_privileges* **ON** *tablename* **FROM** { *users* | **PUBLIC** } **;**

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
database_privilege	One of the following: CONNECT RESOURCE DBA For more information see "Database Privileges" on page 76.
table_privileges	One or more of the following, separated by commas: SELECT INSERT UPDATE [(*columnnames*)] DELETE REFERENCES [(*columnnames*)] ALTER ALL PRIVILEGES
tablename	An identifier specifying the name of an existing table or view.

REVOKE Parameters

Parameter	Description
users	One or more user names separated by commas.
PUBLIC	Specifies all UniVerse users, whether or not they are defined in the SQL catalog.
GRANT OPTION FOR	Specifies that users can no longer grant the specified table privileges to others.

REVOKE Parameters (Continued)

Description

When you revoke a user's ability to grant a privilege to others (privileges granted using WITH GRANT OPTION), you do not break the chain of privileges granted by that user. Only that user's grant option is revoked; all privileges granted by that user remain in effect.

You cannot revoke privileges from yourself.

Database Privileges

There are three levels of database privilege. From lowest to highest they are:

- CONNECT
- RESOURCE
- DBA

Only a user with DBA privilege can revoke database privileges.

CONNECT Privilege

CONNECT privilege registers a user as a UniVerse SQL user. When you revoke a user's CONNECT privilege, the user is removed from the UV_USERS table of the SQL catalog. All of the user's other database and table privileges are also revoked. Schemas and tables formerly owned by users whose CONNECT privilege is revoked become the property of the owner of the SQL catalog (on UNIX systems, either *uvsql*, *root*, or *uvadm*; on Windows platforms, NT AUTHORITY\SYSTEM).

To revoke CONNECT privilege from users who also have RESOURCE or DBA privilege, you must first revoke their RESOURCE Privilege. Revoking RESOURCE privilege also revokes DBA privilege from users who have it.

Users with CONNECT privilege can do the following:

- Create tables
- Alter and drop tables they own
- Grant and revoke table privileges on tables they own
- Use the SELECT, INSERT, UPDATE, and DELETE statements on tables they have access to
- Create and drop views on tables they have access to

This example revokes *maria*'s and *mark*'s CONNECT privileges. The GRANT statement removes their names from the SQL catalog. All their tables become the property of the owner of the SQL catalog.

```
>REVOKE CONNECT
SQL+FROM maria, mark;
```

RESOURCE Privilege

RESOURCE privilege lets UniVerse SQL users create schemas. RESOURCE privilege includes all capabilities of the CONNECT Privilege. When you revoke a user's RESOURCE privilege, the user still has CONNECT privilege. When you revoke RESOURCE privilege from a user who also has DBA privilege, DBA privilege is also revoked.

DBA Privilege

DBA privilege lets UniVerse SQL users execute all SQL statements on all tables and files as if they owned them. It is the highest database privilege. DBA privilege includes all capabilities of the CONNECT Privilege and the RESOURCE Privilege. Users with DBA privilege can also do the following:

- Grant and revoke database privileges
- Create schemas and tables for other UniVerse SQL users
- Grant privileges on any table to any user
- Revoke privileges on any table from any user

When you revoke a user's DBA privilege, the user still has RESOURCE and CONNECT privileges.

Table Privileges

When you create a table, you are the only user with privileges on it, except for users with DBA privilege. The owner of a table can grant any of the following table privileges on it to other users:

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

You can use the ALL PRIVILEGES keyword to revoke all six table privileges at once.

SELECT Privilege

SELECT privilege lets users retrieve data from a table.

INSERT Privilege

INSERT privilege lets users add new rows to a table with the INSERT statement. UniVerse users must have INSERT and UPDATE privileges to add or change data in a table using the Editor, ReVise, or a BASIC program.

UPDATE Privilege

UPDATE privilege lets users modify existing data in a table with the UPDATE statement. UniVerse users must have INSERT and UPDATE privileges to add or change data in a table using the Editor, ReVise, or a BASIC program. You can revoke UPDATE privilege on all columns in a table or on specific columns. The syntax is as follows:

UPDATE [**(***columnnames***)**]

columnnames is one or more column names separated by commas. If you do not specify column names, the privilege is revoked on all columns.

DELETE Privilege

DELETE privilege lets users delete rows from a table.

REFERENCES Privilege

REFERENCES privilege lets users define referenced columns in a referential constraint. You can revoke REFERENCES privilege on all columns in a table or on specific columns. The syntax is as follows:

> **REFERENCES** [**(***columnnames***)**]

columnnames is one or more column names separated by commas. If you do not specify column names, the privilege is revoked on all columns.

ALTER Privilege

ALTER privilege lets users change the structure of a table. Users with ALTER privilege can do the following:

- Add columns to a table
- Add and remove table constraints
- Add and remove association definitions
- Set and remove default specifications

Examples

This example revokes SELECT, UPDATE, and DELETE privileges on three tables from users *maria* and *mark*:

```
>REVOKE SELECT, UPDATE, DELETE
SQL+ON CUSTOMERS, ORDERS, INVENTORY
SQL+FROM maria, mark;
```

The next example revokes *maria*'s ability to grant SELECT privilege on the CUSTOMERS table:

```
>REVOKE GRANT OPTION FOR SELECT
SQL+ON CUSTOMERS
SQL+FROM maria;
```

The next example revokes *mark*'s UPDATE privilege on the CUSTNO column of the CUSTOMERS table:

```
>REVOKE UPDATE (CUSTNO)
SQL+ON CUSTOMERS
SQL+FROM mark;
```

SELECT

Use the SELECT statement to retrieve data from SQL tables and UniVerse files. You must have SELECT Privilege on a table or view in order to retrieve data from it. If you have SELECT privilege only on certain columns of a table, you can retrieve data only from those columns.

Syntax

```
SELECT clause FROM clause
    [WHERE clause]
    [WHEN clause [WHEN clause]…]
    [GROUP BY clause]
    [HAVING clause]
    [ORDER BY clause]
    [FOR UPDATE clause]
    [report_qualifiers]
    [processing_qualifiers]
    [UNION SELECT statement];
```

Description

The SELECT statement can comprise up to nine fundamental clauses. The SELECT and FROM clauses are required.

Parameter	Description
SELECT clause	Specifies the columns to select from the database.
FROM clause	Specifies the tables containing the selected columns.
WHERE clause	Specifies the criteria that rows must meet to be selected.
WHEN clause	Specifies the criteria that values in a multivalued column must meet for an association row to be output.
GROUP BY clause	Groups rows to summarize results.

SELECT Parameters

Parameter	Description
HAVING clause	Specifies the criteria that grouped rows must meet to be selected.
ORDER BY clause	Sorts selected rows.
FOR UPDATE clause	Locks selected rows with exclusive record or file locks.
report_qualifiers	Formats a report generated by the SELECT statement.
processing_qualifiers	Modifies or reports on the processing of the SELECT statement.
UNION	Combines two SELECT statements into a single query that produces one result table.

SELECT Parameters (Continued)

You must specify clauses in the SELECT statement in the order shown. You can use the SELECT statement with type 1, type 19, and type 25 files only if the current isolation level is 0 or 1. The following sections describe each clause in detail.

SELECT Clause

The SELECT clause specifies the columns you want to select from the database. The syntax is as follows:

SELECT { [**ALL** | **DISTINCT**] *column_specifications*
| [*schema*.]*tablename*[*_association*]**.***
| ***** } [**TO SLIST** *list*]

The following table describes each parameter of the syntax.

Parameter	Description
ALL	Selects all specified values, including duplicate values. ALL is the default.
DISTINCT	Eliminates duplicate rows. You can use DISTINCT once per query block.
column_specifications	One or more column expressions, select expressions, or both, separated by commas. In programmatic SQL you cannot use a parameter marker in place of a column specification.
[*schema*.] *tablename* [*_association*].*	Selects all columns in *tablename*, *tablename_association*, *schema.tablename*, or *schema.tablename_association*. *tablename* is an identifier specifying the name of a table, view, or UniVerse data file. *association* is an identifier specifying either the name of an association of multivalued columns in *tablename* or the name of an unassociated multivalued column. The _ (underscore) is part of the syntax and must be typed. If you specify *schema*, *tablename* cannot be the name of a UniVerse file. If the FROM clause has defined an alias for *tablename*, you must use *alias*.* instead of *tablename*.* . If you use *schema* or *association*, they also must be specified in the FROM clause. If the table dictionary contains an @SELECT phrase, *tablename*.* means all columns, real and virtual, listed in the phrase. The list of columns in @SELECT takes precedence over the table's SICA. If *tablename* is a UniVerse file, *tablename*.* means all fields in the @SELECT phrase for the file. If there is no @SELECT phrase, "all columns" means all fields in the @ phrase, plus the record ID (unless the @ phrase contains the ID.SUP keyword). If neither @SELECT nor the @ phrase exist, *tablename*.* means just the record ID field.

SELECT Clause Parameters

Parameter	Description
*	Selects all columns. Columns from a table are selected in the order in which they were defined in the CREATE TABLE statement. If the table dictionary contains an @SELECT phrase, "all columns" means all columns, real and virtual, listed in the phrase. The list of columns in @SELECT takes precedence over the table's SICA. When used to query a UniVerse file, "all columns" means all fields in the @SELECT phrase for the file. If there is no @SELECT phrase, "all columns" means all fields in the @ phrase, plus the record ID (unless the @ phrase contains the ID.SUP keyword). If neither @SELECT nor the @ phrase exist, "all columns" means just the record ID field.
list	Specifies a UniVerse select list to be created. *list* is one of the following: A number from 0 through 10. The select list must be inactive. The name of a select list to be saved in the &SAVEDLISTS& file. The name must contain no spaces and be enclosed in single quotation marks. If you use the TO SLIST clause, you can omit *column_specifications* when you are selecting from a single table. Such a query selects values from the @ID column to the select list. If you use the TO SLIST clause, *column_specifications* cannot specify columns containing multivalued data (unless you are selecting from an association), and the SELECT statement cannot include the WHEN, GROUP BY, or HAVING clauses, or the UNION keyword.

SELECT Clause Parameters (Continued)

Column Expression

A column expression has the following syntax:

[*field_modifier*] *column* [*field_qualifiers*]

The following table describes each parameter of the syntax.

Parameter	Description
field_modifier	One of the following: TOTAL AVG PCT '*n*' BREAK ON "*text* '*options*' …" BREAK SUPPRESS "*text* '*options*' …" CALC Field modifiers do calculations or insert breakpoints before listing the specified column. In programmatic SQL you cannot use field modifiers.
column	You can specify a column by its name, by an EVAL expression, or by an alias. If the table expression references an association of multivalued columns or an unassociated multivalued column as a dynamically normalized table, you can use the @ASSOC_ROW keyword to specify a column containing unique association row numbers when the association has no association keys.
field_qualifiers	One or more of the following, separated by spaces: [AS] *alias* DISPLAYLIKE { [*tablename*.]*columnname* | *alias* } DISPLAYNAME *name* CONV *code* FMT *format* MULTIVALUED | SINGLEVALUED ASSOC *association* ASSOCIATED { [*tablename*.]*columnname* | *alias* } Field qualifiers are valid only for the duration of the current SELECT statement. They define column aliases and override column definitions in the table's dictionary. With the @ASSOC_ROW keyword, you can use only the AS, DISPLAYLIKE, DISPLAYNAME, and FMT field qualifiers.

Column Expressions

Select Expression

A select expression is one or more column expressions, set functions, CAST functions, literals, the keyword NULL, or the keyword USER, combined using arithmetic or string operators and parentheses. A select expression can be followed by one or more field qualifiers.

In programmatic SQL you cannot use a parameter marker in place of a select expression.

Column Aliases

You can use the AS field qualifier to define an alias for any column, whether specified by a column expression or a select expression. You can define only one column alias for each column or select expression.

You can reference the alias later in the same SELECT statement in the following places:

- Select expression
- Set function
- After the DISPLAYLIKE field qualifier
- WHERE Clause[1]
- WHEN Clause
- GROUP BY Clause[2]
- HAVING Clause
- ORDER BY Clause

The alias name is used as the column heading, unless it is overridden by the DISPLAYNAME or DISPLAYLIKE keyword.

1. If the alias is defined for a set function or for a select expression that includes a set function, you cannot reference the alias later in a WHERE or WHEN clause.

2. If the alias is defined for a select expression or a set function, you cannot reference the alias later in a GROUP BY clause.

Field Modifiers

You can use the following field modifiers in a column expression. Field modifiers always precede the column specification.

Field Modifier	Description
TOTAL	Calculates totals for numeric columns. This modifier is often used with breakpoints to produce subtotals. When used in a breakpoint query, the subtotal appears under a row of dashes (----) indicating the breakpoint. If a breakpoint comprises only one line, the subtotal is the same as the detail value. The grand total appears at the end of the report under a row of equal signs (====).
AVG	Calculates the average for numeric columns. Nonnumeric values are treated as zero values. When used in a breakpoint query, breakpoint averages are listed in addition to the overall average at the end of the report.
PCT *n*	Calculates percentages for numeric columns. PCT calculates the total value of the specified column for all rows, then calculates the percent of the total value of the specified column for each row. *n* is an integer from 0 through 5 that specifies how many digits to display after the decimal point. If you omit *n*, two digits are displayed.
BREAK ON "*text* '*options*' …"	Specifies which column to use to create breaks in a report. A break occurs when the column values change. Asterisks or user-specified *text* indicates the breakpoint. This modifier is often used with AVG, CALC, PCT, and TOTAL to perform the specified action and display results when the values change. To make the report more effective, sort the breakpointed column to process and display the same values together. If the column is multivalued, use the UNNEST clause to list association rows separately. *text* is the text you want to appear under the column value in the breakpoint line. If you do not specify *text*, a row of asterisks is used. *text* is not displayed when the report is in vertical format.

Field Modifiers

Field Modifier	Description
	options can be any of the following formatting options. All options suppress the breakpoint row of asterisks. B — Used with the B option of the HEADING or FOOTING report qualifier, includes the current breakpoint value in the heading or footing. Every time the breakpoint value changes, a new page is generated. Only the first B option in a query is used. D — Suppresses printing of the breakpoint line if there is only one line of detail for a value, but leaves a blank line between rows. L — Suppresses printing of the breakpoint line, but still skips a line when the value changes. If *text* is specified, it is ignored. N — Resets the page number to 1 for each new breakpoint value. O — Outputs each breakpoint value only once. P — Begins a new page for every new breakpoint value. V — Inserts the breakpoint field value instead of asterisks.
BREAK SUPPRESS "*text* '*options*' …"	Same as BREAK ON, except BREAK SUPPRESS does not display a row of asterisks or the values in the specified column. *options* can be B, D, and P, as described under BREAK ON.
CALC	Calculates totals in I-descriptors that contain the TOTAL function. Use CALC with breakpointing to produce subtotals. When used with breakpointing, CALC displays intermediate values for the expression on the breakpoint lines. Subtotals calculate an intermediate value at the breakpoint. A grand total is printed at the bottom of the report.

Field Modifiers (Continued)

Field Qualifiers

You can use the following field qualifiers in a column expression. They always follow the column specification. Field qualifiers temporarily override the table dictionary for the duration of the query.

Field Qualifier	Description
AS *alias*	Specifies a new name for *column*. If you omit AS, *alias* must be the first field qualifier in the list. *alias* cannot duplicate the name of an entry in the table's or view's dictionary.
DISPLAYLIKE { [*tablename*.] *columnname* | *alias* }	Sets a column's display characteristics to be the same as those of another column. When used in the same column expression with other field qualifiers, DISPLAYLIKE is processed before CONV, DISPLAYNAME, FMT, SINGLEVALUED, MULTI-VALUED, ASSOC, and ASSOCIATED. You can use any of these field qualifiers to override display characteristics set by DISPLAYLIKE.
DISPLAYNAME *name*	Defines a column heading for *column*. To specify a line break in the column heading, use the letter L enclosed in single quotation marks and enclose *name* in double quotation marks. In programmatic SQL, you cannot use a parameter marker in place of *name*.
CONV *code*	Defines a conversion for *column*. *code* is any BASIC conversion code available to the ICONV and OCONV functions. If there is a conversion in the dictionary entry and you want no conversion applied, specify an empty string in the CONV clause. For more information about conversion codes, see *UniVerse BASIC*. In programmatic SQL you cannot use a parameter marker in place of *code*.
FMT *format*	Defines a format for *column*. *format* specifies the width of the display column, the character used to pad the display field, the type of justification, the format of numeric data, and a format mask. For full details about the syntax of the format expression see the FMT function in *UniVerse BASIC*. If you specify *format* as an empty string, a default format of 10L is used. Invalid format expressions can give unpredictable results. In programmatic SQL you cannot use a parameter marker in place of *format*.

Field Qualifiers

Field Qualifier	Description
MULTIVALUED	Specifies that *column* be treated as multivalued, overriding any existing specification in field 6 of the table dictionary. You cannot use MULTIVALUED in the same column expression with SINGLEVALUED. You cannot use MULTIVALUED in programmatic SQL.
SINGLEVALUED	Specifies that *column* be treated as single-valued, overriding any existing specification in field 6 of the table dictionary. You cannot use SINGLEVALUED in the same column expression with MULTIVALUED, ASSOC, or ASSOCIATED. You cannot use SINGLEVALUED in programmatic SQL.
ASSOC *association*	Associates *column* with an existing association of multivalued columns in the same table. *association* is the record ID of the entry in the table dictionary that defines an association. You cannot use ASSOCIATED or SINGLEVALUED in the same column expression with ASSOC. You cannot associate a column with an association in another table. You cannot use ASSOC in programmatic SQL.
ASSOCIATED { [*tablename*.]*columnname* | *alias* }	Associates *columnname* or *alias* with another multivalued column expression in the same table. You cannot use ASSOC or SINGLEVALUED in the same column expression with ASSOCIATED. You cannot associate a column with a column in another table. You cannot use ASSOCIATED in programmatic SQL.

Field Qualifiers (Continued)

UniVerse Select List

The TO SLIST clause creates either an active or a saved select list. The select list can comprise multicolumn primary keys or association row keys. The resulting select list can be used to restrict the action of any SQL statement or UniVerse command to the rows specified in the select list. See "Specifying Rows with a Select List" on page 110.

In addition to the syntax of the SELECT clause shown earlier in this section, you can use the following simpler syntax to create a UniVerse select list of a table's primary keys:

SELECT TO SLIST *list* [FROM clause] [WHERE clause]
[ORDER BY clause]**;**

If a query expression or SELECT statement contains the TO SLIST clause, it cannot contain any of the following:

- UNION operator
- WHEN clause
- GROUP BY clause
- HAVING clause

Examples

The following examples show different ways to refer to columns in the SELECT clause.

Using Column Names

This example selects two columns by name from the INVENTORY table:

```
>SELECT PROD.NO, "DESC"
SQL+FROM INVENTORY;
Part No    Description..................

210        Red/Blue/Yellow Juggling Bag
102        Red Silicon Ball
112        Red Vinyl Stage Ball
202        Red Juggling Bag
502        Red Classic Ring
318        Gold Deluxe Stage Club
.
.
.
```

Selecting All Columns

The next example uses the * (asterisk) to select all columns from the INVENTORY table:

```
>SELECT * FROM INVENTORY;
Part No  Description.............. Available  Cost...  Price..
Reorder At

210     Red/Blue/Yellow Juggling Bag     77   $3.40   $5.00
21
```

```
102      Red Silicon Ball                45   $14.00   $25.00
9
112      Red Vinyl Stage Ball             20    $3.50    $6.00
12
202      Red Juggling Bag                 94    $3.40    $5.00
21
502      Red Classic Ring                42    $2.80    $5.00
9
318     Gold Deluxe Stage Club           45   $14.00   $23.00
9
```

Eliminating Duplicate Rows

The next example uses DISTINCT to eliminate duplicate rows from the results:

```
>SELECT DISTINCT CUST.NO
SQL+FROM ORDERS;
Customer No

       6518
       4450
       9874
       9825
       3456

5 records listed.
```

Using Constants

The next example uses a text string to clarify output. The text `Amount in stock is:` is printed for each row, in its own column.

```
>SELECT 'Amount in stock is:',QOH,PROD.NO
SQL+FROM INVENTORY;
Amount in stock is:    Available    Part No

Amount in stock is:           77    210
Amount in stock is:           45    102
Amount in stock is:           20    112
Amount in stock is:           94    202
Amount in stock is:           42    502
Amount in stock is:           45    318
      .
      .
      .
```

Selecting Set Functions

The next example uses a set function and two field qualifiers to calculate and list the average profit on items in inventory:

```
>SELECT AVG(SELL-COST) - 1.2 DISPLAYNAME 'Average Profit' CONV
'MD2$'
SQL+FROM INVENTORY;
Average Profit

         $6.30

1 records listed.
```

Using Field Modifiers and Field Qualifiers

The next example uses the field modifier TOTAL and the field qualifiers CONV, FMT, and AS:

```
>SELECT PROD.NO,TOTAL QOH,EVAL 'SELL-COST' CONV 'MD2$' FMT '6R' AS
PROFIT
SQL+FROM INVENTORY
SQL+WHERE PROFIT > 10;
SELL - COST
Part No    Available    PROFIT

102               45    $11.00
418               12    $12.00
704               12    $33.00
605                4    $18.00
101               25    $11.00
103               30    $11.00
419               12    $12.00
           =========
                 140


7 records listed.
```

Using CAST Functions

The next example joins two tables on a character column and an integer column:

```
>SELECT CAST(CHARCOL AS INT) FROM TABLEA UNION
SQL+SELECT INTCOL FROM TABLEB;
```

The next example joins two tables using a place holder:

```
>SELECT COLNAME, CAST(NULL AS DATE) FROM TABLEA UNION
SQL+SELECT COLNAME, DATECOL FROM TABLEB;
```

The next example adds a number to a character column **without** modifying the file dictionary:

```
>SELECT CAST(@ID AS INT) + 1 FROM FILE;
```

The next example adds 5 to a character column, then concatenates it with a % (percent sign):

```
>SELECT CAST(CAST(COLNAME AS INT)+5 AS VARCHAR)||'%'
SQL+FROM TABLEA;
```

The next example finds the internal value of a date:

```
>SELECT CAST('1996-12-25' AS INT) FROM TABLE;
```

The next example finds the date representation of an integer:

```
>SELECT CAST(10587 AS DATE) FROM TABLE;
```

FROM Clause

The FROM clause specifies one or more tables or UniVerse files from which to select data. Its syntax is as follows:

FROM { *table_specification* [[**AS**] *alias*] | *joined_table_specification* }
[, { *table_specification* [[**AS**] *alias*] | *joined_table_specification* } …]

The following table describes each parameter of the syntax.

Parameter	Description	
table_specification	Either a table expression or an UNNEST clause.	
	table_expression	For the syntax of *table_expression*, see “Table.”

FROM Parameters

Parameter	Description	
	UNNEST clause	For information about the UNNEST clause, see "UNNEST Clause."
alias	An identifier specifying another name for *table_specification*. You can use a table's alias to refer to it in other clauses of the SELECT statement. If *table_specification* references an association of multivalued columns or an unassociated multivalued column as a dynamically normalized table, *alias* cannot be the same as the name of a column in the base table.	
joined_table_specification	Specifies a temporary table made up of two or more tables. For the syntax of *joined_table_specification*, see "Joined Tables" on page 112.	

FROM Parameters (Continued)

Examples

The following examples show different ways to refer to tables in the FROM clause.

Using Table Names

This example uses a simple table name to identify the source of the data:

```
>SELECT BILL.TO FROM CUSTOMERS;
Bill to......................

Mr. B. Clown
1 Center Ct.
New York, NY 10020
Mr. D. Juggler
10 Pin Lane
Smalltown, MI 09876
Ms. H. Rider
RFD 3
Hayfield, VT 12345
   .
   .
   .
```

Specifying Rows

The next example specifies a subset of rows in the CUSTOMERS file as the source of the data:

```
>SELECT BILL.TO FROM CUSTOMERS '6518''2309''1043';
Bill to......................

Parade Supply Store
6100 Ohio Ave.
Washington, D.C.  14567
The Great Bandini
45 Cornnut Way
Anytown, OR  34257
Circus Performers Society
P.O. Box 3030
Chicago, IL 71945

3 records listed.
```

Using Aliases

The next example defines two aliases, A and B, for the same table name, and uses the aliases to qualify the PROD.NO and QOH columns so that values in each column can be compared to other values in the same column. This query returns products where two or more have the same quantity on hand (QOH).

```
>SELECT DISTINCT A.PROD.NO, A."DESC", A.QOH
SQL+FROM INVENTORY A, INVENTORY B
SQL+WHERE A.QOH = B.QOH AND A.PROD.NO <> B.PROD.NO
SQL+ORDER BY A.QOH;
Part No    Description..................    Available

418        Gold Deluxe Stage Torch                 12
419        Silver Deluxe Stage Torch               12
704        Sure Balance Unicycle                   12
204        Blue Juggling Bag                       45
111        White Vinyl Stage Ball                  45
102        Red Silicon Ball                        45
318        Gold Deluxe Stage Club                  45

7 records listed.
```

Specifying Rows with a Select List

The next example uses the TO SLIST clause of the SELECT statement to create a select list of all orders dated later than June 1, 1992, then uses the select list in the FROM clause of another SQL SELECT statement. (In programmatic SQL you can use a named select list, but you cannot use a numbered select list.)

```
>SELECT TO SLIST 0 FROM ORDERS WHERE "DATE" > '06-01-92';

4 record(s) selected to SELECT list #0.
>>SELECT ORDER.NO, "DATE", CUST.NO, PROD.NO, QTY FROM ORDERS SLIST
0;
Order No    Order Date    Customer No    Product No    Qty.

10002        14 JUL 92           6518           605       1
                                                501       1
                                                502       1
                                                504       1
10004        22 AUG 92           4450           704       1
                                                301       9
10005        25 NOV 92           9874           502       9
10007        06 JUL 92           9874           301       3

4 records listed.
```

Specifying Rows Interactively

The next example uses INQUIRING to prompt the user to enter a primary key. If the table has no primary key, enter a value from the @ID column. When the user enters a valid value at the prompt, the requested row is returned and the user is prompted to enter another primary key. (In programmatic SQL you cannot use INQUIRING.)

```
>SELECT * FROM ORDERS INQUIRING;
Order No    Customer No    Product No    Qty.    Total.......


Primary Key for table ORDERS =  10001
Order No    Customer No    Product No    Qty.    Total.......

10001              3456           112       7         $265.00
                                  418       4
                                  704       1


Primary Key for table ORDERS =
```

UNNEST Clause

The UNNEST clause unnests the multivalued data of an association. Its syntax is as follows:

UNNEST *table* **ON** { *columnname* | *association* }

The following table describes each parameter of the syntax.

Parameter	Description	
table	The name of a table or UniVerse file. You can specify *table* in two ways:	
	tablename	The name of a table or UniVerse data file.
	DATA *filename*,*datafile*	Specifies a data file that is part of a file with multiple data files.
columnname	The name of a column. *columnname* cannot be a column alias.	
association	The name of an association.	

UNNEST Parameters

UNNEST explodes associated table rows containing multivalued data, generating a separate row for each multivalue. Unnested columns are treated as singlevalued columns for the rest of the query processing. UNNEST acts before all other states of query processing.

The number of unnested rows created per base table row is determined as follows:

- For SQL tables, by the maximum number of multivalues in the unnested association key columns, as defined in the SICA
- For UniVerse files, by the maximum number of multivalues in the unnested association phrase columns or in the controlling attribute column

You cannot use UNNEST if an exploded select list is active. An exploded select list is a select list created using the BY.EXP or BY.EXP.DSND keyword.

Example

This example unnests the ORDERS table on the association called BOUGHT and sorts the data by product number. BOUGHT associates the columns PROD.NO and QTY.

```
>SELECT *
SQL+FROM UNNEST ORDERS ON BOUGHT
SQL+ORDER BY PROD.NO;
Order No    Customer No    Product No    Qty.    Total.......

10006              6518           112       3          $18.00
10001              3456           112       7         $265.00
10003              9825           202      10         $100.00
10003              9825           204      10         $100.00
10004              4450           301       9         $205.00
10007              9874           301       3          $30.00
10001              3456           418       4         $265.00
10002              6518           501       1          $55.00
10005              9874           502       9          $45.00
10002              6518           502       1          $55.00
10002              6518           504       1          $55.00
10002              6518           605       1          $55.00
10004              4450           704       1         $205.00
10001              3456           704       1         $265.00

14 records listed.
```

Joined Tables

The *joined_table_specification* can specify an inner join or a left outer join. The syntax is as follows:

{ *table_specification* | *joined_table_specification* } [**INNER** | **LEFT** [**OUTER**]] **JOIN** *table_specification* { **ON** *condition* | **USING** (*columns*) }

The following table describes each parameter of the syntax.

Parameter	Description	
table_specification	Either a table expression or an UNNEST clause:	
	table_expression	For the syntax of *table_expression*, see "Table."
	UNNEST clause	For information about the UNNEST clause, see "UNNEST Clause."
joined_table_specification	Specifies a temporary table made up of two or more tables.	
INNER	Specifies an inner join.	
LEFT [OUTER]	Specifies a left outer join.	
condition	The columns specified in *condition* must refer to columns in the joined tables that precede the ON keyword. If the joined table specification is part of a subquery, the columns specified in *condition* can refer to columns outside the joined table specification if they are valid outer references. Column names in *condition* cannot be aliases defined in the SELECT clause.	
columns	Specifies the columns on which to join the tables.	

***joined_table_specification* Parameters**

Multiple joins are processed from left to right. You cannot use parentheses in the syntax of a joined table specification to change the order in which multiple joins are processed.

You cannot use the INQUIRING keyword in the table specifications of a joined table specification.

If you omit the INNER or LEFT keywords, or if you specify INNER, the join condition applies to the Cartesian product of the two table specifications. The result table is similar to the one created by the following syntax:

FROM *table_specification1*, *table_specification2*

The result table of an outer join can contain more rows than the result table of the corresponding inner join, because a row is generated for each row in the first table specification (the outer table) that does not meet the join condition against any row in the second table specification. In an inner join, if a row in the first table specification does not meet the join condition against any row in the second table specification, that row is left out of the result table.

In the result of an outer join, the rows from the outer table that do not match any rows of the second table contain null values for all selected columns of the second table. If such a column is multivalued, the row is padded with a single null value.

WHERE Clause

The WHERE clause specifies the criteria that data in a row must meet for the row to be selected. The syntax is as follows:

WHERE [**NOT**] { *condition* | *subquery_condition* }
[{ **AND** | **OR** } [**NOT**] { *condition* | *subquery_condition* } …]

The following table describes each parameter of the syntax.

Parameter	**Description**
condition	For the syntax of *condition*, see "Condition."
subquery_condition	For the syntax of *subquery_condition*, see "Subquery."
AND	Both conditions connected by AND must be true for the row to be selected.
OR	Either condition connected by OR can be true for the row to be selected.

WHERE Parameters

A condition can do the following:

- Compare values to other values, including values returned by a subquery
- Specify a range of values
- Match a phonetic or string pattern
- Test for the null value
- Test whether a subquery produces results

Precede a condition with the keyword NOT to reverse the condition. When you reverse a condition, the test is true only if the condition is false.

A WHERE condition can specify either a condition or a join. The condition can be simple, or it can include another SELECT statement, called a subquery.

You cannot use a set function in a WHERE condition, unless it is part of a subquery.

The following sections provide examples of the kinds of condition you can specify in a WHERE clause.

Comparing Values

You can compare values using relational operators or the IN keyword.

Using Relational Operators

You can compare one expression to another using relational operators.

This example uses the relational operator < (less than) to select only those orders dated before April 1, 1994:

```
>SELECT CUST.NO, "DATE"
SQL+FROM ORDERS
SQL+WHERE "DATE" < '4/1/92';
Customer No    Order Date

       9825     07 MAR 92
       3456     11 FEB 92

2 records listed.
```

The next example evaluates the expression QOH – REORDER and uses the < (less than) operator to select only those items in inventory where the difference between the quantity on hand and the number to reorder is less than 10:

```
>SELECT "DESC", QOH, REORDER.QTY
SQL+FROM INVENTORY
SQL+WHERE QOH - REORDER.QTY < 10;
Description...................    Available    Reorder At

Red Vinyl Stage Ball                     20            12
Gold Deluxe Stage Torch                  12            15
Sure Balance Unicycle                    12             3
Collapsible Felt Top Hat                  4             5
Silver Deluxe Stage Torch                12            15

5 records listed.
```

Using the IN Keyword

You can compare an expression to a list of values using the IN keyword.

This example selects items from inventory where the markup (selling price minus cost) is either $33.00 or $18.00:

```
>SELECT * FROM INVENTORY
SQL+WHERE SELL - COST IN (33.00, 18.00);
Part No  Description.......... Available  Cost...  Price..
Reorder At

704      Sure Balance Unicycle        12   $82.00  $115.00
3
605      Collapsible Felt Top Hat      4   $22.00   $40.00
5

2 records listed.
```

Specifying a Range: BETWEEN

You can specify a range within which the selected data should be found using the BETWEEN keyword.

This example selects order dates and customer numbers of orders dated between March 1 and May 1, 1994:

```
>SELECT "DATE", CUST.NO
SQL+FROM ORDERS
SQL+WHERE "DATE" BETWEEN '3/1/92' AND '5/1/92';
Order Date    Customer No

 22 APR 92           6518
 07 MAR 92           9825

2 records listed.
```

The next example selects items in inventory whose cost is outside the range $3.00 through $22.00:

```
>SELECT PROD.NO, COST FROM INVENTORY
SQL+WHERE COST NOT BETWEEN 3 AND 22;
Part No    Cost...

502          $2.80
504          $2.80
704         $82.00
501          $2.80

4 records listed.
```

Phonetic Matching: SAID

You can test whether a character string is phonetically like another string using the SAID or SPOKEN keyword.

This example selects all customers whose last name sounds like Smith:

```
>SELECT FNAME, LNAME, CITY, STATE
SQL+FROM CUSTOMERS
SQL+WHERE LNAME SAID 'SMITH';
FNAME..........    LNAME..........    CITY...........    STATE

Susan              Smith              Somerville         MA
Brad               Smythe             New York           NY

2 records listed.
```

Pattern Matching: LIKE

You can test whether data matches a pattern using the LIKE, MATCHING, or MATCHES keyword.

This example selects all customers with the word *Clown* in their name:

```
>SELECT CUST.NO, BILL.TO, PHONE
SQL+FROM CUSTOMERS
SQL+WHERE BILL.TO LIKE '%Clown%';
Cust No    Bill to......................    Phone Number...

4450       Mr. B. Clown                      (918) 737-2118
           1 Center Ct.
           New York, NY 10020
9874       The Clown Convention              (617) 665-9890
           18 Porter St.
           Somerville, MA  02143

2 records listed.
```

Testing for the Null Value: IS NULL

You can test whether a value is the null value using the IS NULL keyword.

This example selects engagement dates where gate revenue is null (because the dates are all for future engagements):

```
>SELECT DISTINCT "DATE"
SQL+FROM ENGAGEMENTS.T
SQL+WHERE GATE_REVENUE IS NULL
SQL+ORDER BY "DATE";
DATE......

12/28/94
12/29/94
12/31/94
01/01/95
01/03/95
01/04/95
01/18/95
01/19/95
01/22/95
01/23/95
01/24/95
02/16/95
   .
   .
   .
```

Using Subqueries

You can introduce a subquery in a WHERE clause in three ways. Use one of the following:

- A relational operator
- The IN keyword
- The EXISTS keyword

This example uses the = (equal to) operator with a subquery to select the least expensive products, with their cost and selling price. The set function in the subquery returns one row.

```
>SELECT "DESC", COST, SELL FROM INVENTORY
SQL+WHERE SELL = (SELECT MIN(SELL) FROM INVENTORY);
Description...................    Cost...    Price..

Red/Blue/Yellow Juggling Bag        $3.40      $5.00
Red Juggling Bag                    $3.40      $5.00
Red Classic Ring                    $2.80      $5.00
```

```
Blue Juggling Bag                    $3.40      $5.00
Blue Classic Ring                    $2.80      $5.00
White Classic Ring                   $2.80      $5.00
Yellow Juggling Bag                  $3.40      $5.00

7 records listed.
```

The next example uses the ANY keyword to select any orders which include a product whose selling price is greater than $25.00:

```
>SELECT * FROM ORDERS
SQL+WHERE PROD.NO = ANY (SELECT CAST(PROD.NO AS INT)
SQL+FROM INVENTORY
SQL+WHERE SELL > 25);
Order No    Customer No    Product No    Qty.    Total.......

10002              6518           605       1          $55.00
                                  501       1
                                  502       1
                                  504       1
10004              4450           704       1         $205.00
                                  301       9
10001              3456           112       7         $265.00
                                  418       4
                                  704       1

3 records listed.
```

In the subquery, the CAST function changes the data type of the PROD.NO column (in the INVENTORY table) to INT so it can be compared to the corresponding PROD.NO column in the ORDERS table.

The next example uses the IN keyword with a subquery to extract the names and addresses of customers who have ordered item number 502 from June 30 through August 1, 1992:

```
>SELECT BILL.TO FROM CUSTOMERS
SQL+WHERE CAST(CUST.NO AS INT) IN (SELECT CUST.NO FROM ORDERS
SQL+WHERE PROD.NO = 502
SQL+AND "DATE" BETWEEN '30 JUN 92' AND '1 AUG 92');
Bill to......................

Parade Supply Store
6100 Ohio Ave.
Washington, D.C.  14567

1 records listed.
```

The next example uses the EXISTS keyword with a subquery to request billing addresses for all customers who have an outstanding order:

```
>SELECT BILL.TO FROM CUSTOMERS
SQL+WHERE EXISTS (SELECT * FROM ORDERS
SQL+WHERE CUST.NO = CAST(CUSTOMERS.CUST.NO AS INT))
SQL+DOUBLE SPACE;
BILLTO.......................

Mr. B. Clown
1 Center Ct.
New York, NY 10020

Ms. F. Trapeze
1 High Street
Swingville, ME 98765

Parade Supply Store
6100 Ohio Ave.
Washington, D.C.  14567

The Clown Convention
18 Porter St.
Somerville, MA  02143
North American Juggling
Association
123 Milky Way
Dallas, TX  53485


5 records listed.
```

Joining Tables

A join creates a temporary table made up of two or more tables. There are two types of join:

- Inner join
- Outer join

This section describes inner joins.

You use the WHERE clause to relate at least one column of one table to at least one column of another table. If the data in the related columns meets the criteria, the query returns a pair of related rows, one row from each table.

To join two or more tables in a query, you must do the following:

- List the tables to be joined in the FROM clause.

- Specify a join condition in the WHERE clause.

A join condition defines the relationships among the tables. The syntax is as follows:

> SELECT Clause
> **FROM** *table1*,*table2*[,*table*n…]
> **WHERE** *table1*.*columnname operator table2*.*columnname*
> [**AND** *table*n.*columnname operator table*m.*columnname*…]

This example joins the ORDERS and INVENTORY tables on the PROD.NO columns in each table. Since PROD.NO is multivalued in the ORDERS table, the data is unnested to produce one row for each multivalue. The rows are sorted by order number. Because both tables contain a PROD.NO column, each column name is qualified by prefixing it with its table name.

```
>SELECT ORDER.NO, QTY, "DESC"
SQL+FROM UNNEST ORDERS ON PROD.NO, INVENTORY
SQL+WHERE ORDERS.PROD.NO = CAST(INVENTORY.PROD.NO AS INT)
SQL+ORDER BY ORDER.NO;
Order No    Qty.    Description...................

10001          7    Red Vinyl Stage Ball
10001          4    Gold Deluxe Stage Torch
10001          1    Sure Balance Unicycle
10002          1    Collapsible Felt Top Hat
10002          1    Blue Classic Ring
10002          1    White Classic Ring
10002          1    Red Classic Ring
10003         10    Red Juggling Bag
10003         10    Blue Juggling Bag
10004          9    Classic Polyethylene Club
10004          1    Sure Balance Unicycle
10005          9    Red Classic Ring
10006          3    Red Vinyl Stage Ball
10007          3    Classic Polyethylene Club

14 records listed.
```

The next example shows a reflexive join on the ORDERS table: that is, it joins the ORDERS table to itself. The query finds all orders of customers who have submitted more than one order. The FROM clause specifies the ORDERS table twice, assigning different aliases to each table specification. The WHERE clause uses the aliases to distinguish two references to the same column in the same table. The SELECT clause also uses the aliases to avoid ambiguity.

```
>SELECT A.CUST.NO, A."DATE"
SQL+FROM ORDERS A, ORDERS B
SQL+WHERE A.CUST.NO = B.CUST.NO
SQL+AND A."DATE" <> B."DATE";
Customer No    Order Date

       6518     14 JUL 92
       6518     22 APR 92
       9874     25 NOV 92
       9874     06 JUL 92

4 records listed.
```

The next example shows a three-table join on the ORDERS, CUSTOMERS, and INVENTORY tables. The ORDERS table is unnested on the multivalued field QTY, and the report is double-spaced to make it easier to read.

```
>SELECT ORDER.NO, BILL.TO, QTY, "DESC"
SQL+FROM UNNEST ORDERS ON QTY, CUSTOMERS, INVENTORY
SQL+WHERE ORDERS.CUST.NO = CUSTOMERS.CUST.NO
SQL+AND ORDERS.PROD.NO = INVENTORY.PROD.NO DBL.SPC;
Order No  Bill to......................  Qty.
Description................

   10002  Parade Supply Store               1  Collapsible Felt
Top Hat
          6100 Ohio Ave.
          Washington, D.C.  14567

   10002  Parade Supply Store               1  White Classic Ring
          6100 Ohio Ave.
          Washington, D.C.  14567

   10002  Parade Supply Store               1  Red Classic Ring
          6100 Ohio Ave.
          Washington, D.C.  14567

   10002  Parade Supply Store               1  Blue Classic Ring
          6100 Ohio Ave.
          Washington, D.C.  14567

   10006  Parade Supply Store               3  Red Vinyl Stage
Ball
          6100 Ohio Ave.
          Washington, D.C.  14567
   10004  Mr. B. Clown                      9  Classic
Polyethylene Club
          1 Center Ct.
          New York, NY 10020

   10005  The Clown Convention              9  Red Classic Ring
          18 Porter St.
          Somerville, MA  02143

   10003  North American Juggling          10  Red Juggling Bag
          Association
          123 Milky Way
          Dallas, TX  53485

   10003  North American Juggling          10  Blue Juggling Bag
          Association
          123 Milky Way
          Dallas, TX  53485

   10001  Ms. F. Trapeze                    7  Red Vinyl Stage
Ball
          1 High Street
          Swingville, ME 98765

   10001  Ms. F. Trapeze                    4  Gold Deluxe Stage
Torch
          1 High Street
          Swingville, ME 98765

   10001  Ms. F. Trapeze                    1  Sure Balance
Unicycle
          1 High Street
          Swingville, ME 98765

   10007  The Clown Convention              3  Classic
Polyethylene Club
          18 Porter St.
          Somerville, MA  02143


14 records listed.
```

WHEN Clause

The WHEN clause limits output from multivalued columns to rows in an association that meet the specified criteria. WHEN lists selected multivalues in associated columns without having to unnest the association first. The syntax is as follows:

WHEN [**NOT**] { *condition* | *subquery_condition* }
[{ **AND** | **OR** } [**NOT**] { *condition* | *subquery_condition* } …]

The following table describes each parameter of the syntax.

Parameter	Description
condition	For the syntax of *condition*, see "Condition."
subquery_condition	For the syntax of *subquery_condition*, see "Subquery."
AND	Both conditions connected by AND must be true for the row to be selected.
OR	Either condition connected by OR can be true for the row to be selected.

WHEN Parameters

A condition can do the following:

- Compare values to other values, including values returned by a subquery
- Specify a range of values
- Match a phonetic or string pattern
- Test for the null value
- Test whether a subquery produces results

Precede a condition with the keyword NOT to reverse the condition. When you reverse a condition, the test is true only if the condition is false.

A WHEN condition can be simple, or it can include another SELECT statement, called a subquery.

You cannot use a set function in a WHEN condition unless it is part of a subquery.

The following sections provide examples of the kinds of condition you can specify in a WHEN clause.

Comparing Values

You can compare values using relational operators or the IN keyword.

Using Relational Operators

You can compare one expression to another using relational operators.

This example selects all rows from the ORDERS file but lists only the quantity ordered of product number 704:

```
>SELECT CUST.NO, "DATE", QTY, PROD.NO
SQL+FROM ORDERS
SQL+WHEN PROD.NO = 704;
Customer No    Order Date    Qty.    Product No

       6518     14 JUL 92
       6518     22 APR 92
       4450     22 AUG 92       1           704
       9874     25 NOV 92
       9825     07 MAR 92
       3456     11 FEB 92       1           704
       9874     06 JUL 92

7 records selected.  2 values listed.
```

Using the IN Keyword

You can compare an expression to a list of values using the IN keyword.

This example selects all rows in the ORDERS table but lists the quantity ordered only of product numbers 502 and 112:

```
>SELECT * FROM ORDERS WHEN PROD.NO IN (502,112);
Order No    Customer No    Product No    Qty.    Total.......

10002              6518           502       1         $55.00
10006              6518           112       3         $18.00
10004              4450                              $205.00
10005              9874           502       9         $45.00
10003              9825                              $100.00
10001              3456           112       7        $265.00
10007              9874                               $30.00

7 records selected.  4 values listed.
```

Specifying a Range: BETWEEN

You can specify a range within which the selected data should be found using the BETWEEN keyword.

This example select all rows in the ORDERS file but lists the product numbers and quantity ordered only if more than 5 or fewer than 20 were ordered:

```
>SELECT * FROM ORDERS WHEN QTY BETWEEN 5 AND 20;
Order No    Customer No    Product No    Qty.    Total.......

10002              6518                                $55.00
10006              6518                                $18.00
10004              4450           301       9         $205.00
10005              9874           502       9          $45.00
10003              9825           202      10         $100.00
                                  204      10
10001              3456           112       7         $265.00
10007              9874                                $30.00

7 records selected.  5 values listed.
```

Phonetic Matching: SAID

You can test whether a character string is phonetically like another string using the SAID or SPOKEN keyword.

Pattern Matching: LIKE

You can test whether data matches a pattern using the LIKE, MATCHING, or MATCHES keyword.

This example selects all rows in the LOCATIONS table but lists media information only for radio stations:

```
>SELECT DESCRIPTION FMT '30T', MEDIA_NAME, MEDIA_CONTACT,
MEDIA_PHONE
SQL+FROM LOCATIONS.T
SQL+WHEN MEDIA_NAME LIKE '%Radio%';


DESCRIPTION. Houston State Fair Ground
MEDIA_NAME..... MEDIA_CONTACT........... MEDIA_PHONE.
KKSR Radio      Irwin, Sandra            713/293-6175
KSHG Radio      Unruh, Sheryl            713/572-8998

DESCRIPTION. Washington State Fair Ground
MEDIA_NAME..... MEDIA_CONTACT........... MEDIA_PHONE.
WGBU Radio      Yamaguchi, James         202/522-5618

DESCRIPTION. Springfield State Fair Ground
MEDIA_NAME..... MEDIA_CONTACT........... MEDIA_PHONE.
KDVT Radio      Kroll, Vanessa           217/428-3921
KBXZ Radio      Martinez, Darlene        217/390-9427
```

```
    .
    .
    .
DESCRIPTION. Kansas City State Fair Ground
MEDIA_NAME..... MEDIA_CONTACT........... MEDIA_PHONE.
KHMU Radio      Kozlowski, Nancy          816/978-1378
KTEA Radio      Macbride, Gary            816/782-3572


32 records selected.  44 values listed.
```

Testing for the Null Value: IS NULL

You can test whether a value is the null value using the IS NULL keyword.

Using Subqueries

You can introduce a subquery in a WHEN clause in two ways. Use one of the following:

- A relational operator
- The IN keyword

You cannot use the EXISTS keyword in a WHEN subquery.

This example selects all rows in the ORDERS table but lists only those items ordered whose selling price is the lowest of all products in the INVENTORY table:

```
>SELECT ORDER.NO, PROD.NO, SELL FROM ORDERS
SQL+WHEN SELL = (SELECT MIN(SELL) FROM INVENTORY);
Order No    Product No    Sell.....

10002              605        $5.00
                   501        $5.00
                   502        $5.00
                   504
10006              112
10004              704
                   301
10005              502        $5.00
10003              202        $5.00
                   204        $5.00
10001              112
                   418
                   704
10007              301

7 records selected.  6 values listed.
```

The next example uses the ANY keyword to select any orders for products whose selling price is greater than $25.00:

```
>SELECT * FROM ORDERS
SQL+WHEN PROD.NO = ANY (SELECT CAST(PROD.NO AS INT)
SQL+FROM INVENTORY WHERE SELL > 25);
Order No    Customer No    Product No    Qty.    Total.......

10002              6518           605       1         $55.00
10006              6518                                $18.00
10004              4450           704       1        $205.00
10005              9874                                $45.00
10003              9825                               $100.00
10001              3456           418       4        $265.00
                                  704       1
10007              9874                                $30.00

7 records selected.  4 values listed.
```

The next example uses the IN keyword with a subquery to select all rows in the ORDERS table but list only those items ordered whose selling price is $5.00:

```
>SELECT ORDER.NO, "DATE", PROD.NO, QTY, SELL FROM ORDERS
SQL+WHEN PROD.NO IN (SELECT CAST(PROD.NO AS INT) FROM INVENTORY
SQL+WHERE SELL = 5.00);
Order No    Order Date    Product No    Qty.    Sell.....

10002        14 JUL 92           501       1       $40.00
                                 502       1        $5.00
                                 504       1        $5.00
                                                    $5.00
10006        22 APR 92                              $6.00
10004        22 AUG 92                            $115.00
                                                   $10.00
10005        25 NOV 92           502       9        $5.00
10003        07 MAR 92           202      10        $5.00
                                 204      10        $5.00
10001        11 FEB 92                              $6.00
                                                   $27.00
                                                  $115.00
10007        06 JUL 92                             $10.00

7 records selected.  6 values listed.
```

GROUP BY Clause

The GROUP BY clause groups rows that have identical values in all grouping columns and returns a single row of results for each group. The syntax is as follows:

GROUP BY { *columnname* | *alias* } [{ ,*columnname* | *alias* } …]

Each column in a GROUP BY clause must be singlevalued.

alias can refer only to a named column or an EVAL expression. It cannot refer to a select expression or a set function.

Each column selected in the SELECT Clause must be included in the GROUP BY clause. If the SELECT clause includes an EVAL expression, you must assign it an alias with the AS keyword, and include the alias in the GROUP BY clause. If the SELECT clause includes a set function, the set function is applied to each group.

Examples

This example groups all items from inventory that have the same selling price and shows how many items are in each selling price group. Each selling price produces a different row.

```
>SELECT SELL, COUNT(*)
SQL+FROM INVENTORY
SQL+GROUP BY SELL;
SELL...    COUNT ( * )

  $5.00              7
  $6.00              3
 $10.00              1
 $19.00              1
 $23.00              2
 $25.00              3
 $27.00              2
 $40.00              1
$115.00              1

9 records listed.
```

The next example groups items ordered and lists the total number and price for each item ordered:

```
>SELECT ORDERS.PROD.NO, SUM(QTY), SUM(QTY * ORDERS.SELL) CONV
'MR2$'
SQL+FROM UNNEST ORDERS ON PROD.NO, INVENTORY
SQL+WHERE ORDERS.PROD.NO = CAST(INVENTORY.PROD.NO AS INT)
SQL+GROUP BY ORDERS.PROD.NO;
Product No    SUM ( QTY )    SUM ( QTY * ORDERS.SELL )

       112             10                     $1054.00
       202             10                      $100.00
       204             10                      $100.00
       301             12                     $1155.00
       418              4                      $592.00
       501              1                       $55.00
```

```
        502             10                            $100.00
        504              1                             $55.00
        605              1                             $55.00
        704              2                            $273.00

  10 records listed.
```

HAVING Clause

The HAVING clause specifies the criteria that data in a group must meet for the group to be selected. The syntax is as follows:

HAVING [**NOT**] { *condition* | *subquery_condition* }
[{ **AND** | **OR** } [**NOT**] { *condition* | *subquery_condition* } …]

The following table describes each parameter of the syntax.

Parameter	**Description**
condition	For the syntax of *condition*, see "Condition."
subquery_condition	For the syntax of *subquery_condition*, see "Subquery."
AND	Both conditions connected by AND must be true for the group to be selected.
OR	Either condition connected by OR can be true for the group to be selected.

HAVING Parameters

A condition can do the following:

- Compare values to other values, including values returned by a subquery
- Specify a range of values
- Match a phonetic or string pattern
- Test for the null value
- Test whether a subquery produces results

Precede a condition with the keyword NOT to reverse the condition. When you reverse a condition, the test is true only if the condition is false.

A HAVING condition can be simple, or it can include another SELECT statement, called a subquery.

You most often use the HAVING clause with a GROUP BY Clause. If you do not include a GROUP BY clause, the whole table is considered to be a group.

Example

This example groups orders by customer number and selects only those customers whose order quantity is greater than 11:

```
>SELECT CUST.NO, SUM(QTY)
SQL+FROM ORDERS
SQL+GROUP BY CUST.NO
SQL+HAVING SUM(QTY) > 11
SQL+ORDER BY CUST.NO;
CUST.NO....    SUM ( QTY )

      3456             12
      9825             20
      9874             12

3 records listed.
```

ORDER BY Clause

The ORDER BY clause sorts the results of a query. The syntax is as follows:

ORDER BY { *column* | *col#* } [**ASC** | **DESC**] [, { *column* | *col#* } [**ASC** | **DESC**] …]

The following table describes each parameter of the syntax.

Parameter	Description
column	For the syntax of *column*, see "Column."
col#	A number representing the position of the column specification in the SELECT Clause. If you use an * (asterisk) in the SELECT clause to specify all columns, *col#* represents the position of the column name in the column list contained in the @SELECT phrase of the table dictionary. If there is no @SELECT phrase, *col#* represents the position of the column definition in the CREATE TABLE statement that created the table. In programmatic SQL you cannot use a parameter marker in place of a column number. If the SELECT clause includes the TO SLIST clause but does not include column specifications, you cannot use *col#* to specify the ordering column.
ASC	Sorts values in ascending order. ASC is the default sort order.
DESC	Sorts values in descending order.

ORDER BY Parameters

If *column* or *col#* is multivalued, value marks are ignored, and the values are treated as a single field and sorted as a unit. To sort by values in a multivalued column, use the UNNEST clause of the FROM clause, or dynamically normalize the multivalued column or association.

If the SELECT statement includes a GROUP BY Clause, you can sort only on columns included in the SELECT clause.

If the SELECT clause includes both the DISTINCT keyword and the TO SLIST clause, you cannot use an ORDER BY clause in the SELECT statement.

Examples

This example shows the column DESCRIP listed first in the SELECT clause, followed by the expression SELL – COST. The column is referred to as position 1, and the expression is referred to as position 2 in the ORDER BY clause.

```
>SELECT "DESC", SELL - COST
SQL+FROM INVENTORY
SQL+ORDER BY 2 DESC, 1;
Description..................    SELL - COST

Sure Balance Unicycle                     33
Collapsible Felt Top Hat                  18
Gold Deluxe Stage Torch                   12
Silver Deluxe Stage Torch                 12
Red Silicon Ball                          11
White Silicon Ball                        11
Yellow Silicon Ball                       11
Gold Deluxe Stage Club                     9
Silver Deluxe Stage Club                   9
   .
   .
   .
```

FOR UPDATE Clause

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 is as follows:

FOR UPDATE [**OF** { [[*schema*.]*tablename*.]*columnname* | *alias* }
[,{ [[*schema*.]*tablename*.]*columnname* | *alias* }] …]

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.

columnname cannot be the keyword @ASSOC_ROW. *alias* must refer to a named column, not to a select expression, an EVAL expression, or @ASSOC_ROW.

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. The following table shows which locks are set:

Isolation Level	Locks Set
0 or 1	Update record locks (READU) are set for all selected rows, even though an ordinary SELECT statement does not set locks at these isolation levels.
2 or 3	Update record locks (READU) are set for all selected rows, instead of the shared record locks (READL) set by an ordinary SELECT statement.
4	Exclusive file locks (FX) are set for all tables and files referenced by the SELECT statement, instead of the shared file locks (SH) set by an ordinary SELECT statement.

Isolation Levels

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;
```

Report Qualifiers

Report qualifiers format the output of interactive SELECT statements. For more information about Report Qualifier Keywords, see the *UniVerse User Reference*.

Specifying Headings and Footings

Use the following report qualifiers to define a heading or footing for the report:

Report Qualifier	Description	
HEADING "*text* [**'***options***'**]*…***"**	*options* (in single quotation marks) can be included anywhere inside the double quotation marks. *options* can be any of the following:	
	B[*n*]	Inserts the current breakpoint field value in a field of *n* spaces when used with the B option of BREAK ON. Each new value generates a new page.
	C[*n*]	Centers the heading in a field of *n* spaces.
	D	Inserts the current date.
	F[*n*]	Inserts the filename left-justified in a field of *n* spaces.
	G	Inserts gaps in the heading format.
	I[*n*]	Inserts the record ID left-justified in a field of *n* spaces. Same as R.
	L	Inserts a carriage return and linefeed to make a multiple-line heading.
	N	Suppresses page pause during a terminal display.
	P[*n*]	Inserts the page number left-justified in a field of *n* spaces. The keyword begins with page 1 and adds 1 for each successive page.
	Q	Lets you use the characters] (right bracket), ^ (caret), and \ (backslash) in heading text.
	R[*n*]	Inserts the record ID left-justified in a field of *n* spaces. Same as I.

Report Qualifiers

Report Qualifier	Description	
	S	Inserts the page number left-justified. One character space is reserved for the number. If the number of digits exceeds 1, text to the right of the number is shifted right by the number of extra digits.
	T	Inserts the current time and date.
HEADING DEFAULT	Generates the standard RetrieVe heading line.	
FOOTING "*text* [**'***options***'**]… **"**	*options* (in single quotation marks) can be included anywhere inside the double quotation marks. *options* are the same as those listed for the HEADING report qualifier.	

Report Qualifiers (Continued)

If you do not use the HEADING keyword, the output report has no heading line and starts on the next line of the screen. If you specify HEADING, output starts on a new page or at the top of the screen.

HEADER is a synonym for HEADING. FOOTER is a synonym for FOOTING.

This example defines a centered heading for a report:

```
>SELECT * FROM INVENTORY HEADING "CURRENT INVENTORY LIST: 'TC'";
                CURRENT INVENTORY LIST: 12:31:41PM  23 Jul 1997
Part No  Description.................  Available  Cost...
Price..  Markup

210      Red/Blue/Yellow Juggling Bag         77    $3.40
$5.00     47%
102      Red Silicon Ball                     45   $14.00
$25.00     79%
112      Red Vinyl Stage Ball                 20    $3.50
$6.00     71%
```

The T option prints the current time and date, and the C option centers the heading.

Specifying Text for a Grand Total Line

Use GRAND TOTAL to specify text to print on the grand total line of a report. The syntax is as follows:

GRAND TOTAL "*text* **'***options***'**… **"**

Options are L and P. L suppresses the display of the double bar line above the grand total line. P prints the double bar line and grand total line on a separate page.

GRAND.TOTAL is a synonym for GRAND TOTAL.

Suppressing Column Headings

Use SUPPRESS COLUMN HEADING to suppress default column headings. COL.SUP and SUPPRESS COLUMN HEADER are synonyms for SUPPRESS COLUMN HEADING.

This example suppresses the default column headings of the ORDERS table:

```
>SELECT * FROM ORDERS SUPPRESS COLUMN HEADING;


10002    6518    605       1         $55.00
                 501       1
                 502       1
                 504       1
10006    6518    112       3         $18.00
10004    4450    704       1        $205.00
                 301       9
10005    9874    502       9         $45.00
10003    9825    202      10        $100.00
                 204      10
10001    3456    112       7        $265.00
                 418       4
                 704       1
10007    9874    301       3         $30.00

7 records listed.
```

Suppressing Row Count

Use COUNT.SUP to suppress the message that lists the number of rows selected.

This example suppresses the message `7 records listed.`, which is normally printed after the last row:

```
>SELECT * FROM ORDERS COUNT.SUP;
Order No    Customer No    Product No    Qty.    Total.......

10002              6518           605       1         $55.00
                                  501       1
                                  502       1
                                  504       1
10006              6518           112       3         $18.00
10004              4450           704       1        $205.00
                                  301       9
10005              9874           502       9         $45.00
```

```
10003             9825          202      10          $100.00
                                204      10
10001             3456          112       7          $265.00
                                418       4
                                704       1
10007             9874          301       3           $30.00
```

Suppressing Breakpoint Detail Lines

Use SUPPRESS DETAIL in a SELECT statement that includes the BREAK ON field modifier to display breakpoint lines only. DET.SUP is a synonym for SUPPRESS DETAIL.

This example unnests the multivalued column PROD.NO and sorts the rows by PROD.NO. The quantity of each product ordered is totalled for each product. Only the breakpoint lines showing the totals are shown.

```
>SELECT ORDER.NO, BREAK ON PROD.NO, TOTAL QTY
SQL+FROM UNNEST ORDERS ON PROD.NO
SQL+ORDER BY PROD.NO
SQL+SUPPRESS DETAIL;
Order No    Product No    Qty.

10001              112      10
10003              202      10
10003              204      10
10007              301      12
10001              418       4
10002              501       1
10002              502      10
10002              504       1
10002              605       1
10001              704       2
                          ====
                            61

14 records listed.
```

Adjusting Spacing

Use the following report qualifiers to adjust spacing between columns, rows, and at the left margin:

- DOUBLE SPACE
- COLUMN SPACES[*n*]
- MARGIN *n*

DBL.SPC is a synonym for DOUBLE SPACE. COL.SPCS and COL.SPACES are synonyms for COLUMN SPACES.

This example double-spaces each row for ease in reading. Multivalues within rows are single-spaced.

```
>SELECT * FROM ORDERS DOUBLE SPACE;
Order No    Customer No    Product No    Qty.    Total.......

10002              6518           605       1          $55.00
                                  501       1
                                  502       1
                                  504       1

10006              6518           112       3          $18.00

10004              4450           704       1         $205.00
                                  301       9

10005              9874           502       9          $45.00

10003              9825           202      10         $100.00
                                  204      10

10001              3456           112       7         $265.00
                                  418       4
                                  704       1

10007              9874           301       3          $30.00
   .
   .
   .
```

The next example reduces the column spacing to two spaces:

```
>SELECT * FROM ORDERS COLUMN SPACES 2;
Order No  Customer No  Product No  Qty.  Total.......

10002            6518         605     1        $55.00
                              501     1
                              502     1
                              504     1
10006            6518         112     3        $18.00
10004            4450         704     1       $205.00
                              301     9
10005            9874         502     9        $45.00
10003            9825         202    10       $100.00
                              204    10
```

```
10001            3456         112     7       $265.00
                              418     4
                              704     1
10007            9874         301     3        $30.00

7 records listed.
```

Listing Data in Vertical Format

Use VERTICALLY to force output to be listed in vertical format. Each column of each row is listed on a separate line. VERT is a synonym for VERTICALLY.

This example displays each column value on its own line. Multivalues within rows are listed in columns.

```
>SELECT * FROM ORDERS VERTICALLY;


Order No.... 10002
Customer No. 6518
Product No Qty.
       605    1
       501    1
       502    1
       504    1
Total.......       $55.00

Order No.... 10006
Customer No. 6518
Product No Qty.
       112    3
Total.......       $18.00

Order No.... 10004
Customer No. 4450
Product No Qty.
       704    1
       301    9
Total.......      $205.00
   .
   .
   .
```

Suppressing Paging

Use NOPAGE to override automatic paging. The report is scrolled continuously on the screen or printed without formatted page breaks on the printer. NO.PAGE is a synonym for NOPAGE.

Sending Output to the Printer

Use LPTR [*n*] to send output to the system printer. *n* is an integer from 0 through 255 specifying a logical print channel number. Use AUX.PORT to send output to a printer attached to the terminal's auxiliary port (output also goes to the screen).

Processing Qualifiers

Processing qualifiers affect or report on the processing of SQL queries.

Showing How a Query Will Be Processed (EXPLAIN)

Use EXPLAIN in a SELECT statement to display information about how the statement will be processed. This lets you decide if you want to rewrite the query more efficiently.

EXPLAIN lists the tables included in the query, 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 you use EXPLAIN in a SELECT statement executed by a client program, the statement is not processed. Instead, an SQLSTATE value of IA000 is returned, along with the EXPLAIN message as the message text.

This example shows what the EXPLAIN display looks like:

```
>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. To avoid using the optimizer, use the NO.OPTIMIZE keyword.

This example runs the preceding example without using the query optimizer:

```
>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)

At isolation level 2 or higher, 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 instead of waiting when the SELECT statement encounters a record or file lock. If the SELECT 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 the client program.

If a SELECT statement with NOWAIT selects an I-descriptor or an EVAL expression that executes a 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.*

This example runs the preceding example. If the query 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;
```

Limiting Number of Rows Selected (SAMPLE and SAMPLED)

Use SAMPLE and SAMPLED to limit the number of rows selected:

SAMPLE [*n*]	Selects the first *n* rows.
SAMPLED [*n*]	Selects every *n*th row.

SAMPLE and SAMPLED select a limited number of rows before the ORDER BY clause processes the data.

In programmatic SQL you cannot use a parameter marker in place of *n* to specify rows.

This example selects a sample of three rows from the ORDERS table:

```
>SELECT * FROM ORDERS SAMPLE 3;
Order No    Customer No    Product No    Qty.    Total.......

10002              6518           605       1         $55.00
                                  501       1
                                  502       1
                                  504       1
10006              6518           112       3         $18.00
10004              4450           704       1        $205.00
                                  301       9

Sample of 3 records listed.
```

UNION Operator

The UNION operator combines the results of two SELECT statements into a single result table. A set of SELECT statements joined by UNION operators is called a *query expression*. You can use query expressions as interactive SQL queries, programmatic SQL queries, and in the CREATE VIEW statement. However, you cannot use a query expression as a subquery or in the INSERT statement.

The syntax of a query expression is as follows:

SELECT statement [**UNION** [**ALL**] SELECT statement] …

The following table describes each parameter of the syntax.

Parameter	**Description**
UNION	Combines two SELECT statements.
ALL	Specifies that duplicate rows not be removed from the result table. If you do not specify ALL, duplicate rows are removed from the result table.

UNION Parameters

By default, SELECT statements joined by the UNION operator are processed from left to right. Use parentheses to specify a different processing order.

Note*: You cannot enclose the entire query expression in parentheses.*

The column names, column headings, formats, and conversions used when generating the result table are all taken from the first SELECT statement.

If the query expression is used in a CREATE VIEW statement, the view's SICA and table dictionary will contain only one set of column definitions, based on the first SELECT statement. Such a view is not updatable.

The following restrictions apply to SELECT statements joined by the UNION operator:

- All of the SELECT statements must specify the same number of result columns.
- Corresponding columns among the SELECT statements must all belong to the same data category (character, number, date, or time). For information about data categories, see Chapter 3, "Data Types."
- You cannot use the INQUIRING keyword in FROM clauses.
- You cannot use field modifiers in column expressions.
- You can use only the field qualifiers AS, DISPLAYLIKE, DISPLAYNAME, CONV, and FMT. Except for AS, you must use these field qualifiers only in the first SELECT statement.
- In an interactive or programmatic SQL query, you can specify an ORDER BY Clause only after the last SELECT statement in the query expression. You must use integers, not column specifications, to specify the columns by which you want to order the result set. Ordering applies to the entire result table.
- You cannot use the following report qualifiers in a query expression:

  GRAND TOTAL
  SUPPRESS DETAIL

  You can specify all other report qualifiers only after the last SELECT statement in the query expression, and only after the ORDER BY clause if there is one.
- You cannot use the following processing qualifiers in a query expression:

  EXPLAIN
  SAMPLE
  SAMPLED

  You can specify NO.OPTIMIZE and NOWAIT only after the last SELECT statement in the query expression, and only after the ORDER BY clause if there is one.

Example

Consider the following three SELECT statements:

```
SELECT ITEM FROM INVENTORY;
SELECT PRODUCT FROM SHELF;
SELECT REQUEST FROM ORDER;
```

The first query returns two rows: NUT and BOLT. The second query also returns two rows: HINGE and BOLT. The third query returns one row: NUT. If you combine these queries this way:

```
>SELECT ITEM FROM INVENTORY UNION
SQL+SELECT PRODUCT FROM SHELF UNION ALL
SQL+SELECT REQUEST FROM ORDER;
```

the resulting rows are NUT, BOLT, HINGE, and NUT. If you use parentheses to change the order of processing, like this:

```
>SELECT ITEM FROM INVENTORY UNION
SQL+(SELECT PRODUCT FROM SHELF UNION ALL
SQL+SELECT REQUEST FROM ORDER);
```

the resulting rows are NUT, BOLT, and HINGE.

UPDATE

Use the UPDATE statement to modify values in a table, view, or UniVerse file. To update data in a table or view, you must own it or have UPDATE Privilege on it. To update a view you must also have UPDATE privilege on the underlying base table and any underlying views. You also need write permissions on the directory and the VOC file in the account. You cannot update rows in a type 1, type 19, or type 25 file.

Syntax

```
UPDATE table_expression
       SET set_expressions
       [WHERE clause]
       [WHEN clause [WHEN clause]... ]
       [qualifiers];
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
table_expression	Specifies the table or view to update with new data. For the syntax of *table_expression*, see "Table." If *table_expression* references an association of multivalued columns or an unassociated multivalued column as a dynamically normalized table, UPDATE changes data in specified association rows.
SET	Introduces the SET clause.
set_expressions	One or more set expressions separated by commas. A set expression specifies the columns in the table to be updated and the new values for these columns.
WHERE Clause	Specifies which rows are to be updated.

UPDATE Parameters

Parameter	Description	
WHEN Clause	Specifies criteria for updating rows in an association. You can use one or more WHEN clauses. Multivalued columns named in a WHEN clause must all belong to the same association. You cannot use a WHEN clause if *table_expression* references an association of multivalued columns or an unassociated multi-valued column as a dynamically normalized table.	
qualifiers	One or more of the following processing qualifiers separated by spaces:	
	EXPLAIN	Lists the tables referenced by the UPDATE statement and explains how the query optimizer will handle execution of the statement.
	NO.OPTIMIZE	Suppresses the optimizer when processing WHERE and WHEN clauses.
	NOWAIT	If the UPDATE statement encounters a lock set by another user, it terminates immediately. It does not wait for the lock to be released.
	REPORTING	Displays the status of the update and the primary keys of the rows updated. If the table has no primary key, displays the values in @ID.

UPDATE Parameters (Continued)

Description

You must specify clauses in the UPDATE statement in the order shown. If you do not specify a WHERE Clause, all rows are updated. If you specify a WHEN Clause, only values in the specified association rows are updated. If you are updating multivalued columns or an unassociated multivalued column and you do not specify a WHEN clause, all multivalues in the selected rows are updated.

When you use the UPDATE statement to modify rows in a table, the modified rows are written back to the original table. If you use the UPDATE statement on a table that has column or table constraints, the new data must meet the constraint criteria. If the UPDATE statement fails while it is updating more than one row, no rows are updated.

Using EXPLAIN

The EXPLAIN keyword lists all tables referenced by the UPDATE statement, including tables referenced by subqueries in the WHERE and WHEN clauses, and explains how the query optimizer will use indexes, process joins, and so forth, when the statement is executed. If the WHERE Clause or WHEN Clause include subqueries, information is given about each query block.

If you use EXPLAIN in an interactive UPDATE statement, after viewing the EXPLAIN message, press **Q** to quit, or press any other key to continue processing.

If you use EXPLAIN in an UPDATE statement executed by a client program, the statement is not processed. Instead, an SQLSTATE value of IA000 is returned, along with the EXPLAIN message as the message text.

Using NOWAIT

The NOWAIT condition applies to:

- All locks encountered by the UPDATE statement
- All cascaded updates and deletes that result from the UPDATE statement
- All SQL operations in trigger programs fired by the UPDATE statement

In these cases the UPDATE statement and all its dependent operations are terminated and rolled back, and an SQLSTATE value of 40001 is returned to client programs.

Set Expressions

Use the SET clause to specify update values. The SET clause can include one or more set expressions separated by commas. A set expression specifies the column to update and assigns new values to the column. The syntax for a set expression is as follows:

columnname = *values*

columnname is the name of the column containing values to update. If the column is multivalued, all values in each specified row are updated.

If several column names in a SET clause map to the same column in a view's underlying base table or view, the last column name in the SET clause determines the value of the column.

values is one of the following:

Value	Description
expression	Specifies the data elements with which to update the column. You cannot use set functions in a set expression. Any multivalued columns in *expression* must be associated with *columnname*.
NULL	Specifies the null value.
DEFAULT	For tables, specifies the default value. If no default value is defined, the default value is null. For multivalued columns, the default value is the empty string. For files, the default value is also the empty string.
multivalues	A set of comma-separated values enclosed in angle brackets, to update a multivalued column. The syntax is as follows: <*value* [, *value* …]> The angle brackets are part of the syntax and must be typed. A value can be NULL, DEFAULT, or an expression. For each row, the number and order of values must be the same as the number and order of the corresponding values in the association key. You cannot specify *multivalues* if the UPDATE statement includes a WHEN clause. **Note**: If *table_expression* references an association of multivalued columns or an unassociated multivalued column as a dynamically normalized table, you cannot use this syntax to update multivalues in association rows.

UPDATE Values

This example changes every value in the multivalued column COLOR to BLACK:

```
>UPDATE INVENTORY SET COLOR = 'BLACK';
```

The next example multiplies every value in the multivalued column QTY by 2:

```
>UPDATE ORDERS SET QTY = QTY * 2;
```

You can also use the SET clause to perform mathematical operations on associated multivalued columns. In the next example, the SET clause multiplies the first value in PRICE by the first value in QTY, and so on, and puts the result in ITEM.EXT, one association row at a time. You can perform mathematical operations using multivalued columns only if they belong to the same association:

>**UPDATE ORDERS SET ITEM.EXT = PRICE * QTY;**

The next example raises all prices in the PRODUCTS table by 10% and resets the accumulated sales to 0:

```
>UPDATE PRODUCTS
SQL+SET PRICE = PRICE * 1.1,
SQL+SALESTD = 0;
```

WHERE Clause

Use the WHERE clause to specify rows to update. If *table_expression* references an association of multivalued columns or an unassociated multivalued column as a dynamically normalized table, you can use the @ASSOC_ROW keyword to specify unique association rows if the association has no keys.

This example changes the department from SALES to SALES,EAST for employees 301 through 399:

```
>UPDATE EMPLOYEES
SQL+SET DEPT = 'SALES,EAST'
SQL+WHERE DEPT = 'SALES' AND EMPNO BETWEEN 301 AND 399;
```

The next example updates every value in the multivalued column QTY for order 10003. The REPORTING keyword displays the status of the update and the record ID of the updated row.

```
>UPDATE ORDERS SET QTY = QTY + 10
SQL+WHERE ORDER.NO = 10003 REPORTING;

UniVerse/SQL: Record "10003" updated.
UniVerse/SQL: 1 record updated.
```

The next example resets CUSTOMER values in specified rows of the ORDERS table to the default values:

```
>UPDATE ORDERS SET CUSTOMER = DEFAULT
SQL+WHERE CUSTOMER = 'AJAX';
```

You can update a table in a remote schema by specifying the name of the remote schema in the table expression. The next example modifies the DEPTS table in the remote schema OTHERSCHEMA:

```
>UPDATE OTHERSCHEMA.DEPTS
SQL+SET DEPTNAME = 'TELSALES', MANAGER = 'JONES'
SQL+WHERE DEPTNO = 316;
```

WHEN Clause

Use a WHEN clause to specify the association rows to update. The WHEN clause restricts selected association rows to those that pass the WHEN criteria. This lets you update specific values in multivalued columns.

***Note**: You cannot use a WHEN clause if a set expression specifies a set of multivalues for a column. You cannot use a WHEN clause if the table expression references an association of multivalued columns or an unassociated multivalued column as a dynamically normalized table.*

In this example the WHERE clause limits the update to order 10007. The WHEN clause further limits the update to values in association rows containing BOLT in the ITEM column. For those association rows, 10 is added to the quantity orders, and the name of the item is changed to HINGE. The QTY and ITEM columns are associated.

```
>UPDATE ORDERS SET QTY = QTY + 10, ITEM = 'HINGE'
SQL+WHERE ORDER.NO = 10007
SQL+WHEN ITEM = 'BOLT';
```

Referential Integrity Actions

If you update rows in a referenced table, rows in the referencing table may also be changed, depending on the referential constraint action prescribed by the referencing table's REFERENCES clause.

The following table shows what happens when the referencing table's REFERENCES clause contains the ON UPDATE clause:

If the referenced column is...	And if the referencing column is...	The following referential integrity action occurs:
Single-valued	Single-valued	ON UPDATE CASCADE replaces the value in the referencing column with the updated value in the referenced column. ON UPDATE SET NULL and ON UPDATE SET DEFAULT replace the value in the referencing column with a null value or the column's default value, respectively. If the corresponding columns are parts of a multipart column set, all corresponding part-columns must match for the corresponding values in the referencing columns to be updated.
Multivalued	Single-valued	ON UPDATE CASCADE is not allowed if the referenced column is multivalued. ON UPDATE SET NULL and ON UPDATE SET DEFAULT replace the value in the referencing column with a null value or the column's default value, respectively.
Single-valued	Multivalued	ON UPDATE CASCADE replaces the multivalue in the referencing column with the updated value in the referenced column. ON UPDATE SET NULL and ON UPDATE SET DEFAULT replace the multivalue in the referencing column with a null value or the column's default value, respectively.
Multivalued	Multivalued	ON UPDATE CASCADE is not allowed if the referenced column is multivalued. ON UPDATE SET NULL and ON UPDATE SET DEFAULT replace the multivalue in the referencing column with a null value or the column's default value, respectively.

Referential Integrity Actions

Column

Use column syntax when you see *column* in a syntax line. You can use column syntax in the following clauses of the SELECT statement:

- SELECT Clause
- GROUP BY Clause
- ORDER BY Clause

You can also use column syntax in conditional expressions in the following clauses of the SELECT statement:

- WHERE Clause
- WHEN Clause
- HAVING Clause

Syntax

You can specify a column in one of three ways: as a column name, an EVAL expression, or an alias. *column* has one of the following syntaxes:

[[*schema*.]*tablename*.]*columnname*
EVAL[**UATE**] [[*schema*.] *tablename*.]'*I_type_expression*'
alias

The following table describes each parameter of the syntax.

Parameter	Description
schema**.**	The name of a schema other than the current schema, followed by a . (period), used to reference a table in a remote schema.
*tablename***.**	The name of a table or table alias, followed by a . (period), used to distinguish between columns having the same name.
columnname	The name of a column as defined in the SICA of one of the tables in the FROM clause. If the table expression of a SELECT statement references an association of multivalued columns or an unassociated multivalued column as a dynamically normalized table, you can use the @ASSOC_ROW keyword to specify a column containing unique association row numbers when the association has no association keys. You can also use @ASSOC_ROW in WHERE clause comparisons. You can qualify @ASSOC_ROW with *tablename*.
EVAL [UATE]	Introduces an in-line I-type expression, sometimes called an EVAL expression.
I_type_expression	Any valid I-type expression, enclosed in single or double quotation marks. If you name more than one table in the FROM clause, you must precede *I_type_expression* with the name of the table containing the columns named in the expression. If you name a table in another schema in the FROM clause, you must precede any reference to that table with the schema name. In programmatic SQL you cannot use a parameter marker in place of an I-type expression.
alias	An alternate name for a column expression or a select expression, or the name of an EVAL expression or set function expression. Aliases are defined using the AS keyword.

Column Parameters

EVAL Expressions

An EVAL expression defines a new virtual column which exists only for the duration of the current SELECT statement. When the column is output, the following default conditions apply:

- The column heading is the text of the I-type expression, unless you use the DISPLAYNAME keyword (or one of its synonyms) to specify a column heading. If the text is longer than 20 characters, value marks are inserted in the text every 20 characters or at the width defined by the display format, whichever is greater. This creates a multiline column heading.
- The conversion, display format, single- or multivalue code, and association specifications are derived from the dictionary entry that defines the first column appearing in the I-type expression.
- If no column appears in the I-type expression, the temporary column has no conversion, its format is 10L, and it is singlevalued.

For more details about I-type expressions, see the *UniVerse System Description*.

Condition

Use condition syntax when you see *condition* in a syntax line. You use conditions to determine whether data meets specified criteria. You can use conditions in the following statements:

- ALTER TABLE (CHECK constraint)
- CREATE TABLE (CHECK constraint)
- DELETE (WHERE Clause)
- SELECT (WHERE Clause, WHEN Clause, and HAVING Clause)
- UPDATE (WHERE Clause and WHEN Clause)

A condition is one or more search conditions combined using AND, OR, NOT, and parentheses, which when evaluated give a logical result of true, false, or unknown. Conditions can do the following:

- Compare expressions to other expressions
- Compare expressions to values returned by a subquery

Syntax

This list summarizes the syntax of conditions that compare expressions:

expression operator expression
expression [**NOT**] **IN** ('*value*' [,'*value*' …])
expression [**NOT**] **BETWEEN** *expression* **AND** *expression*
expression [**NOT**] { **SAID** | **SPOKEN** } '*string*'
expression [**NOT**] { **LIKE** | **MATCHING** | **MATCHES** } '*pattern*' [**ESCAPE**
    '*character*']
expression **IS** [**NOT**] **NULL**

In WHERE clauses and CHECK constraints you can precede any of these conditions with the keyword EVERY. EVERY specifies that every value in a multivalued column must meet the criteria. If both expressions in a comparison are multivalued and you specify EVERY, EVERY is implied on the right side of the comparison. If you do not specify EVERY, any single pair of values can meet the criteria.

Subsequent sections explain each syntax line in detail.

CHECK Constraint Conditions

You cannot use CURRENT_DATE or CURRENT_TIME in a CHECK constraint condition.

If *expression* refers to several different multivalued columns, they must all belong to the same association. Only corresponding values in each association row are compared to each other.

If you later add or drop an association, you must drop and recreate any CHECK constraints.

Comparing Values

You can compare values using relational operators or the IN keyword.

Using Relational Operators

You can compare one expression to another using relational operators. The syntax is as follows:

expression operator expression

operator is one of the relational operators. In programmatic SQL, if you use a parameter marker in place of the first expression, you cannot use a parameter marker in place of the second expression; if you use a parameter marker for the second expression, you cannot use a parameter marker for the first expression.

You cannot use relational operators to test for the null value. To test for null, use the IS NULL keyword.

Here are some examples:

```
PROD.NO = 704
"DATE" < "06-01-94"
LNAME = "Smith" AND FNAME = "Barbara"
ORDERS.PARTNO = I.PARTNO
STATE <> 'VT'
```

The second expression can be a subquery.

Using the IN Keyword

You can compare an expression to a list of values using the IN keyword. The syntax is as follows:

expression [**NOT**] **IN (**'*value*' [**,**'*value*'...]**)**

value is a character string, bit string, numeric, date, or time literal. You cannot use character string functions with *value*. In programmatic SQL you cannot use parameter markers in place of both *expression* and the first *value* after the IN keyword.

Here are two examples:

```
STATE IN ('MA', 'NY', 'CT')
COST NOT IN (14.00, 3.40, 3.50)
```

You can also use the IN keyword to compare an expression to the values returned by a subquery.

Specifying a Range: BETWEEN

You can specify a range within which the selected data should be found using the BETWEEN keyword. The syntax is as follows:

expression [**NOT**] **BETWEEN** *expression* **AND** *expression*

The second and third expressions must be single-valued. In programmatic SQL, if you use a parameter marker in place of the first expression, you cannot use a parameter marker in place of the second or third expression. If you use a parameter marker for the second or third expression, you cannot use a parameter marker for the first expression.

Here are some examples:

```
"DATE" BETWEEN '01-01-94' AND '06-30-94'
ORDER.NO NOT BETWEEN 10100 AND 10400
PRICE BETWEEN 5.00 AND 20.00
```

Phonetic Matching: SAID

You can test whether a character string is phonetically like another string using the SAID or SPOKEN keyword. The syntax is as follows:

> *expression* [**NOT**] { **SAID** | **SPOKEN** } **'***string***'**

expression is described in the section "Expression." *string* is a character string literal. You cannot use character string functions with *string*.

Here are two examples:

```
LNAME SAID 'SMITH'
NAME SAID 'PIERCE'
```

Pattern Matching: LIKE

You can test whether data matches a pattern using the LIKE, MATCHING, or MATCHES keyword. The syntax is as follows:

> *expression* [**NOT**] { **LIKE** | **MATCHING** | **MATCHES** } **'***pattern***'**
> [**ESCAPE '***character***'**]

expression is described in the section "Expression." *pattern* is a character string literal. It can include the wildcard character % (percent), which matches zero or more characters. It can include the wildcard character _ (underscore), which matches any single character. You cannot use character string functions with *pattern*.

If you want to use % or _ as a literal character in *pattern*, you must remove its wildcard status by preceding it with an escape character. You must also use the ESCAPE clause to specify the escape character. You cannot specify any of the following characters as escape characters:

Character	Description
%	Percent
_	Underscore

Invalid Escape Characters

Character	Description
.	Period
'	Single quotation mark
"	Double quotation mark

Invalid Escape Characters (Continued)

Here are some examples:

```
"DESC" LIKE '%Ball%'
BILL.TO MATCHES '%Dallas%'
ADR3 LIKE '%CA%'
TABLE_NAME LIKE 'UV\_%' ESCAPE '\'
```

Testing for the Null Value: IS NULL

You can test whether a value is the null value using the IS NULL keyword. You can test whether a value is not the null value using IS NOT NULL. The syntax is as follows:

expression **IS** [**NOT**] **NULL**

expression is described in the section "Expression."

Here are two examples:

```
PAID IS NULL
CLOSED IS NOT NULL
```

The IS NULL keyword is the only way to test for the null value. You cannot use relational operators to test for null.

Data Type

The data type of a column specifies the kind of data the column contains. You specify data types in the following statements and functions:

- CREATE TABLE
- ALTER TABLE
- CAST Function

You must define a data type for each column in a table. UniVerse SQL recognizes 15 data types.

Syntax

To specify a data type, use the following syntax:

CHAR [**ACTER**] [(*n*)]
{ **VARCHAR** | **CHAR** [**ACTER**] **VARYING** } [(*n*)]
{ **NCHAR** | **NATIONAL CHAR** [**ACTER**] } [(*n*)]
{ **NVARCHAR** | **NCHAR VARYING** | **NATIONAL CHAR** [**ACTER**]
    VARYING [(*n*)]
BIT [(*n*)]
{ **VARBIT** | **BIT VARYING** } [(*n*)]
INT [**EGER**]
SMALLINT
FLOAT [(*precision*)]
REAL
DOUBLE PRECISION
DEC [**IMAL**] [(*precision* [,*scale*])]
NUMERIC [(*precision* [,*scale*])]
DATE
TIME

For detailed information about data types, see Chapter 3, “Data Types.”

Expression

An expression is one or more data elements combined using arithmetic operators, the concatenation operator, and parentheses. You can use expressions in the following statements:

- SELECT
- DELETE (WHERE Clause)
- UPDATE (SET clause, WHERE Clause, and WHEN Clause)
- CREATE TABLE (CHECK constraint)
- ALTER TABLE (CHECK constraint)

The order in which operations are performed is as follows, with expressions in parentheses being performed first:

1. Negation
2. Multiplication and division
3. Addition and subtraction

In programmatic SQL you cannot use parameter markers to replace data elements on both sides of an arithmetic operator.

Data elements you can use in an expression are as follows:

Data Element	**Description**
column	For the syntax of *column*, see "Column" on page 157."
set_function	For the syntax of *set_function*, see "Set Function" on page 178.
literal	A character string, a bit string, a number, a date, or a time. For information about literals, see "Literal" on page 175.
function expression	One of the following: SUBSTRING TRIM UPPER LOWER CHAR_LENGTH

Data Elements in Expressions

Data Element	Description
cast_function	For the syntax of *cast_function*, see CAST Function.
CURRENT_DATE	Specifies the current system date.
CURRENT_TIME	Specifies the current system time.
USER	Specifies the current effective user name.

Data Elements in Expressions (Continued)

Concatenation Operator

You can use the concatenation operator (| |) to concatenate two character string expressions.

You cannot use the concatenation operator in a CHECK condition that references multivalued columns.

CAST Function

You can use the CAST function to convert the data type of an expression. The syntax is as follows:

CAST (*expression* **AS** *datatype***)**

The following data conversions are not allowed:

- Scaled or approximate numbers to DATE or TIME
- Dates to scaled or approximate numbers, or to TIME
- Times to scaled or approximate numbers, or to DATE

datatype cannot be NCHAR, NVARCHAR, BIT, or VARBIT.

Function Expressions

You can use five function expressions in a character string expression:

- Substring function
- Upper function

- Lower function
- Trim function
- Length function

Substring Function

The substring function extracts a substring from a character string. The syntax is as follows:

SUBSTRING (*character_string* **FROM** *start* [**FOR** *length* **)**

The following table describes each parameter of the syntax.

Parameter	Description
character_string	A value expression that evaluates to a character string.
start	An integer specifying the character from which to start the extraction. If *start* is 0 or negative, it represents an imaginary character position to the left of the first real character. In this case, *length* counts from the imaginary position, but the returned substring starts at the first real character.
length	An integer specifying the number of characters to extract. If *length* is negative, SUBSTRING returns an empty string. If you do not specify *length*, the returned substring is from *start* to the end of the string.

SUBSTRING Parameters

Upper and Lower Functions

The upper function converts the letters a through z to A through Z. The lower function converts the letters A through Z to a through z. The syntax is as follows:

UPPER (*character_string***)**

LOWER (*character_string***)**

character.string is a value expression that evaluates to a character string.

Trim Function

The trim function removes all occurrences of a specified character from the beginning, the end, or both ends of a character string. The syntax is as follows:

> **TRIM ([{LEADING | TRAILING | BOTH} [*character*] FROM]** *character_string***)**

The following table describes each parameter of the syntax.

Parameter	Description
LEADING	Removes all instances of *character* from the left of *character_string*.
TRAILING	Removes all instances of *character* from the right of *character_string*.
BOTH	Removes all instances of *character* from both ends of *character_string*. This is the default.
character	The character you want to trim. The default is the space character.
character_string	A value expression that evaluates to a character string.

TRIM Parameters

Length Function

The length function returns the length of a character string as an integer. The syntax is as follows:

> **CHAR[ACTER]_LENGTH (***character_string***)**

character_string is a value expression that evaluates to a character string.

Examples

Here are some examples of expressions that use arithmetic operators:

```
(PRICE - COST)/COST
PRICE * 1.25
"DATE" + 30
```

Here are some examples of character string expressions that use function expressions and the concatenation operator:

```
SUBSTRING('String Expression' FROM 8 FOR 4)
UPPER('UniVerse')
TRIM('   word  ')
TRIM(BOTH ' ' FROM '   word  ')
CHAR_LENGTH('sub'||'zero')
```

The returned values are:

```
Expr
UNIVERSE
word
word
7
```

Identifier

Identifiers specify the names of the following SQL objects:

- Schemas
- Tables
- Views
- Indexes
- Triggers
- Columns
- Associations
- Constraints

Identifiers can also be the following:

- Correlation names
- Column aliases

Identifiers that are not delimited start with a letter and can include letters, digits, and underscores.

Delimited Identifiers

You can enclose any identifier in double quotation marks, making it a *delimited identifier*. In addition to letters, digits, and underscores, delimited identifiers can include the space character and all printable 7-bit ASCII characters except the following:

- The five system delimiters:
  - Item mark (CHAR(255))
  - Field mark
  - Value mark
  - Subvalue mark
  - Text mark
- All control characters (CHAR (0) through CHAR (32))

- A single double quotation mark
- The null value (CHAR (128))

To embed a single double quotation mark in a delimited identifier, use two consecutive double quotation marks within the identifier.

In UniVerse SQL you can use double quotation marks:

- To specify a delimited identifier
- After the following keywords and their synonyms, to enclose the required character string literals:
  - BREAK ON
  - BREAK SUPPRESS
  - CONV
  - DISPLAYNAME
  - EVAL
  - EXPLAIN
  - FMT
  - FOOTING
  - GRAND TOTAL
  - HEADING
  - LIKE
  - SAID

  You can also use single quotation marks to enclose literals that follow these keywords.

Note*: You cannot use double quotation marks to enclose character string literals anywhere else in UniVerse SQL.*

The following rules apply to delimited identifiers:

- Schema names cannot contain any periods.

- Table and view names cannot contain commas, slashes, or question marks. In addition, using the following characters in table and view names may make them inaccessible to some UniVerse commands:

  " (double quotation mark)' (single quotation mark)
  \ (backslash), (comma)
  Space

  - On UNIX systems, table and view names cannot begin with a period.
  - On Windows platforms, table and view names cannot contain the following characters:

" (double quotation mark)	> (right bracket)
* (asterisk)	% (percent)
: (colon)	\ (backslash)
< (left bracket)	

- Table and view names cannot duplicate the names of any VOC entries.
- Column names cannot contain any spaces.
- Column names used in CHECK constraints, UNIQUE constraints, the CREATE INDEX statement, and EVAL Expressions cannot contain the following characters:

" (double quotation mark)	[(left bracket)
~ (tilde)	] (right bracket)
! (exclamation point)	{ (left brace)
@ (at sign)	} (right brace)
# (hash sign)	\ (backslash)
^ (caret)	| (vertical bar)
& (ampersand)	; (semicolon)
* (asterisk)	: (colon)
((left parenthesis)	‘ (single quotation mark)
) (right parenthesis)	, (comma)
+ (plus sign)	< (left bracket)
– (minus sign)	> (right bracket)
= (equal sign)	/ (slash)
? (question mark)	

- Table and column names in EVAL Expressions should not be enclosed in quotation marks, unless this is proper UniVerse BASIC syntax.

You can use delimited identifiers in compound syntax items such as qualified column names or a dynamically normalized association. For example:

```
"PERSONNEL".EMPLOYEES."SALARY"
ORDERS_"ITEM#".PRICE
```

Literal

There are six kinds of literal in UniVerse SQL:

- Character strings
- Bit strings
- Hex strings
- Numbers
- Dates
- Times

Character Strings

A character string literal is a string of characters enclosed in single quotation marks. All characters except item marks, field marks, value marks, and subvalue marks are allowed. To specify a single quotation mark in a character string, use two consecutive single quotation marks within the quoted string. To specify an empty string (a character string of zero length), use two consecutive single quotation marks alone.

Bit Strings

A bit string literal is an arbitrary sequence of bits (0's and 1's) enclosed in single quotation marks and preceded by the base specifier B. An example of a bit string literal is:

```
B'01111110'
```

Hex Strings

A hex string literal is a sequence of extended digits (0 – 9, A – F, a – f) enclosed in single quotation marks and preceded by the base specifier X. An example of a hex string literal is:

```
X'1F3A2'
```

Numbers

A fixed-point number is a sequence of digits, which can contain a decimal point and can be prefixed by a + (plus) or – (minus) sign. You cannot include commas in a fixed-point number.

A floating-point number is a fixed-point number followed by the letter E and an integer from –307 through +307. You can prefix the number with a plus sign.

You can include leading and trailing zeros after a decimal point.

Dates

UniVerse SQL dates are like UniVerse dates. You must enclose a date in single quotation marks. A UniVerse SQL date must have two occurrences of its delimiter, and the only delimiters you can use are - (hyphen), / (slash), and   (space).

Use the keyword CURRENT_DATE to specify the current system date.

Some examples of date literals are:

```
'11 SEPT 1994'
'07/05/94'
'4-15-94'
```

Times

UniVerse SQL times are like UniVerse times. You must enclose a time in single quotation marks. A UniVerse SQL time must have only one or two occurrences of its delimiter, and the only delimiter you can use is : (colon).

Use the keyword CURRENT_TIME to specify the current system time.

Some examples of time literals are:

```
'9:34'
'15:50:30'
'0:00:00'
```

***Note**: When entering literals in UniVerse SQL statements, you cannot use the special control-character sequences* **Ctrl-^**, **Ctrl-]**, *and* **Ctrl-|** *to enter system delimiters, nor can you use* **Ctrl-N** *to enter the null value.*

Relational Operator

The relational operators are as follows:

Operator	Description
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
=	Equal to
<>	Not equal to
#	Not equal to

Relational Operators

When used with character strings, less than means before and greater than means after, following ASCII order. For example A is less than B and Z is greater than Y. Since lowercase letters follow uppercase letters, Z is less than a and z is greater than A; and since alphabetic characters follow numeric, A is greater than 1 and 9 is less than a.

Set Function

Use set function syntax when you see *set_function* in a syntax line. You can use set functions in the following clauses of the SELECT statement:

- SELECT Clause
- HAVING Clause

A set function lists one value for a set of rows. It takes either all the values in a column or a set of grouped values and summarizes the selected values in a single value.

Syntax

COUNT(*)
{ **AVG** | **MAX** | **MIN** | **SUM** | **COUNT** } **(DISTINCT** *column***)**
{ **AVG** | **MAX** | **MIN** | **SUM** | **COUNT** } **(** [**ALL**] *select_expression***)**

The following table describes each parameter of the syntax.

Parameter	Description
COUNT	COUNT(*) counts the number of selected rows. Used with a column expression or select expression, COUNT counts the number of values.
AVG	Averages selected or grouped column values, or values returned by a select expression.
MAX	Returns the highest value in a column or group of values.
MIN	Returns the lowest value in a column or group of values.
SUM	Totals selected or grouped column values, or values returned by a select expression.
DISTINCT	Eliminates duplicate values from a column or group of values before applying the set function.

COUNT Parameters

Parameter	Description
ALL	Specifies all values in a column or group of values, including duplicate values. ALL is the default.
column	For the syntax of *column*, see "Column" on page 157.
select_expression	One or more *column*s, literals, or the keyword USER, combined using arithmetic operators and parentheses.

COUNT Parameters (Continued)

Here are some set function examples:

```
SELECT COUNT(*) FROM ORDERS
SELECT SUM(ORDER.TOTAL) FROM ORDERS
SELECT SUM(QTY * SELL) FROM ORDERS
SELECT MAX(COST) FROM INVENTORY
```

Subquery

Use subquery syntax when you see *subquery* in a syntax line. You can use subqueries in the following statements:

- DELETE (WHERE Clause)
- SELECT (WHERE Clause, WHEN Clause, and HAVING Clause)
- UPDATE (WHERE Clause and WHEN Clause)

You use subqueries in conditions.

Syntax

There are three ways to include a subquery in a condition. The syntax is as follows:

expression operator [**ALL** | **ANY** | **SOME**] **(***subquery***)**
expression [**NOT**] **IN (***subquery***)**
[**NOT**] **EXISTS (***subquery***)**

The following table describes each parameter of the syntax.

Parameter	**Description**
expression	See "Expression" on page 166.
operator	One of the relational operators. If you do not specify ALL, ANY, or SOME, the subquery must return only one row.
ALL	Compares *expression* to all values returned by the subquery. The result of the condition is true only if *all* comparisons are true or if the subquery returns no values.
ANY	Compares *expression* to all values returned by the subquery. If *any* comparison is true, the result of the condition is true.
SOME	Same as ANY.
EXISTS	Tests for the existence of data specified in the subquery.

Subquery Syntax

In WHERE conditions, you can precede the first two of the preceding conditions with the keyword EVERY. EVERY specifies that every value in a multivalued column must meet the criteria for the row to be selected. You cannot use EVERY in an EXISTS condition.

In WHEN conditions, you cannot use the EXISTS keyword.

The syntax of a subquery is as follows:

(SELECT { [**ALL** | **DISTINCT**] *select_expression* } FROM clause
  [WHERE Clause]
  [GROUP BY Clause]
  [HAVING Clause] **)**

select_expression must return values in only one column.

Examples

This example uses the = (equal to) operator with a subquery to select the least expensive products, with their cost and selling price. The set function in the subquery returns one row.

```
>SELECT "DESC", COST, SELL FROM INVENTORY
SQL+WHERE SELL = (SELECT MIN(SELL) FROM INVENTORY);
Description..................    Cost...    Price..

Red/Blue/Yellow Juggling Bag       $3.40      $5.00
Red Juggling Bag                   $3.40      $5.00
Red Classic Ring                   $2.80      $5.00
Blue Juggling Bag                  $3.40      $5.00
Blue Classic Ring                  $2.80      $5.00
White Classic Ring                 $2.80      $5.00
Yellow Juggling Bag                $3.40      $5.00

7 records listed.
```

The next example uses the ANY keyword to select any orders which include a product whose selling price is greater than $25.00:

```
>SELECT * FROM ORDERS
SQL+WHERE PROD.NO = ANY(SELECT CAST(PROD.NO AS INT) FROM INVENTORY
SQL+WHERE SELL > 25);
Order No    Customer No    Product No    Qty.    Total.......

10002              6518           605       1          $55.00
                                  501       1
                                  502       1
                                  504       1
```

```
10004              4450             704      1         $205.00
                                    301      9
10001              3456             112      7         $265.00
                                    418      4
                                    704      1

3 records listed.
```

The next example uses the IN keyword with a subquery to extract the names and addresses of customers who have ordered item number 502 from June 30 through August 1, 1992:

```
>SELECT BILL.TO FROM CUSTOMERS
SQL+WHERE CAST(CUST.NO AS INT) IN (SELECT CUST.NO FROM ORDERS
SQL+WHERE PROD.NO = 502
SQL+AND "DATE" BETWEEN '30 JUN 92' AND '1 AUG 92');
Bill to......................

Parade Supply Store
6100 Ohio Ave.
Washington, D.C.  14567

1 records listed.
```

The next example uses the EXISTS keyword with a subquery to request billing addresses for all customers who have an outstanding order:

```
>SELECT BILL.TO FROM CUSTOMERS
SQL+WHERE EXISTS (SELECT * FROM ORDERS
SQL+WHERE CUST.NO = CAST(CUSTOMERS.CUST.NO AS INT))
SQL+DOUBLE SPACE;
Bill to......................

Mr. B. Clown
1 Center Ct.
New York, NY 10020

Ms. F. Trapeze
1 High Street
Swingville, ME 98765
   .
   .
   .
North American Juggling
Association
123 Milky Way
Dallas, TX  53485


5 records listed.
```

Table

Use table expression syntax when you see *table_expression* in a syntax line. You use table expressions in the following statements:

- DELETE
- INSERT
- SELECT
- UPDATE

A table expression can specify just the name of a table, view, or UniVerse file. For tables and files it can also specify which particular rows you want to process. And it can specify an alternate dictionary to use with the table.

Syntax

The syntax of a table expression is as follows:

table [*rows*] [*alt_dict*] [**NO.INDEX**]

The following table describes each parameter of the syntax.

Expression	Description
table	The name of a table, view, or UniVerse file. You can specify *table* in three ways: [*schema*.]*tablename*[*_association*] An identifier specifying the name of a table, view, or UniVerse file. *association* is an identifier specifying either the name of an association of multivalued columns in *tablename* or the name of an unassociated multivalued column. The _ (underscore) is part of the syntax and must be typed. If you specify *schema*, *tablename* cannot be the name of a UniVerse file. DICT [*schema*.]*tablename* Specifies the dictionary of *tablename* as the source of the data. If you specify *schema*, *tablename* cannot be the name of a UniVerse file. DATA *filename*,*datafile* Specifies a data file that is part of a file with multiple data files.

Table Expression Parameters

Expression	Description
rows	Specifies explicitly the rows you want to select from the table. You cannot specify *rows* in an INSERT statement. You can specify *rows* in three ways: '*primary_key*' ['*primary_key*' …] One or more primary key values, each enclosed in single quotation marks. If the table has no primary key, specify values from the @ID column. INQUIRING — The INQUIRING keyword specifies that you want the system to interactively prompt you to enter primary key or @ID values. Prompting continues until you press **Return**. You can use INQUIRING only once in a SELECT statement. You cannot using INQUIRING in a correlated subquery or in programmatic SQL. SLIST *list* — Specifies a select list containing primary key or @ID values. *list* is one of the following: • A number from 0 through 10. If select list 0 is active, you must specify **`SLIST 0`**, otherwise a fatal error occurs. • The name of a saved select list in the &SAVEDLISTS& file. The name must be enclosed in single quotation marks. You cannot specify select list 0 in programmatic SQL.
alt_dict	Specifies an alternate dictionary for *table*. Use the USING DICT keyword to specify an alternate dictionary: **USING DICT** *filename* *filename* is the name of a table, view, or UniVerse file whose dictionary you want to use with *table*.
NO.INDEX	Suppresses all use of a table’s secondary indexes.

Table Expression Parameters (Continued)

Table 6-180

If you specify *rows*, and if the table expression references an association of multivalued columns or an unassociated multivalued column as a dynamically normalized table, use the following format to specify primary key values:

*table_key***TM***assoc_key*

table_key is the primary key of the base table. If the table has no primary key, *table_key* is a value in the @ID column. *assoc_key* is the association key, and **TM** is a text mark.

If an association has no association keys, use the following format to specify the primary key values of a dynamically normalized table:

*table_key***TM**@ASSOC_ROW

UniVerse SQL Grammar

This appendix presents the syntax of the UniVerse SQL statements using Backus Naur Form (BNF) notation together with some explanatory notes.

The formal syntax uses BNF and comprises a series of symbols and production rules that successively break down a statement into its components. The production rules end when a symbol is defined in terms of terminal symbols. Terminals are the actual keywords and variables that you enter to form the statement.

BNF Conventions

BNF uses the following conventions:

Convention	Description
<…>	Angle brackets indicate nonterminal symbols that are further decomposed by production rules. For example, <column specification> is a nonterminal that appears in the production rule for an <interactive value expression> and which, in turn, has its own production rule.
{…}	Braces indicate a group of symbols that must appear together. For example, {, <user name>} means that a comma and then a <user name> must appear together (this is known as a comma-list).
[…]	Square brackets indicate optional symbols in the grammar. For example, [{, <user name>}] means that a comma and then a <user name> are optional here.
|	A vertical bar indicates a series of choices, one of which may be chosen. For example, the production rule for <action> is one of SELECT, INSERT, DELETE, and UPDATE. These are separated from each other by a vertical bar.
::=	The ::= symbol denotes the production rule that decomposes the symbol to the left of the ::= into its components to the right of the ::= .

BNF Conventions

Common Syntax Elements

This section describes common elements in UniVerse SQL syntax. These common elements form the basis for other UniVerse SQL language components.

Tokens, Characters, and Symbols

A token is a character string that is treated as a single unit when parsing an SQL statement. There are three kinds of token: keywords, delimiters, and identifiers. Keywords are UniVerse SQL reserved words. Delimiters terminate the current token. Identifiers are user-assigned names.

```
<token> ::=
            <nondelimiter token>
          | <delimiter token>

<nondelimiter token> ::=
            <identifier>
          | <key word>
          | <unsigned numeric literal>
          |   <bit string literal>
          |   <hex string literal>

<identifier> ::=
            <simple Latin letter> [ { [_] <letter or digit> } ... ]
          | <delimited identifier>

<simple Latin Letter> ::=
            <simple Latin upper case letter>
          | <simple Latin lower case letter>

<delimited identifier> ::=
            " { <letter or digit> | <""> | <SQL special character> } ... "

<letter or digit> ::=
            <simple Latin upper case letter>
          | <simple Latin lower case letter>
          | <digit>

<SQL special character> ::=
            The space character and all printable 7-bit ASCII characters
```

other than letters, digits, and the double quotation mark

```
<delimiter token> ::=
        <character string literal>
        | , | ( | ) | < | > | . | : | = | * | + | – | / | | | <> | >= | <= | # | <space> | <tab>
```

Keywords

Keywords can be entered in uppercase, lowercase, or mixed case letters. For a list of UniVerse SQL keywords, see Appendix B, "Reserved Words."

Delimiters

With the few exceptions noted in the section "Literals" on page 5, these are the <delimiter token>s supported in UniVerse SQL:

Token	Description
	One or more consecutive space, tab, or newline characters.
--	Two consecutive hyphens introduce a comment. A newline character terminates the comment. You can put a comment within, but not before or after, an SQL statement.
;	A semicolon ends an SQL statement.
, . :	Comma, period, colon.
( )	Left parenthesis, right parenthesis.
+ – * /	Plus, minus, times, divided by.
< > = #	Less than, greater than, equal, does not equal.
<> >= <=	Does not equal, greater than or equal, less than or equal.
literal	Character string. See "Literals" on page 5.

Delimiter Tokens

Delimiters in UniVerse SQL cannot appear in <identifier>s or <key word>s.

Literals

There are four kinds of literal: character string, numeric, bit string, and hex string. In addition, SQL accepts the keywords USER, CURRENT_DATE, and CURRENT_TIME in most places where it accepts a literal. The USER keyword represents the current effective user name. The CURRENT_DATE keyword represents today's date. The CURRENT_TIME keyword represents the current time.

```
<literal> ::=
          <signed numeric literal>
          | <character string literal>
```

Numeric Literals

An <unsigned numeric literal> is the same as in UniVerse, but should not be enclosed in quotation marks. A fixed-point number is a sequence of digits that can contain a decimal point and can be preceded by a plus or minus sign. Commas are not allowed. A floating-point number is a fixed-point number followed by the letter E and an integer from –307 through +307. A plus sign in the exponent is optional. Leading zeros and trailing zeros after a decimal point are allowed.

```
<signed numeric literal> ::= [+ | –] <unsigned numeric literal>

<unsigned numeric literal> ::=
          <exact numeric literal>
          | <approximate numeric literal>

<exact numeric literal> ::=
          <unsigned integer>[.[<unsigned integer>]]
          | .<unsigned integer>

<unsigned integer> ::= <digit>…

<approximate numeric literal> ::= <mantissa> E <exponent>

<mantissa> ::= <exact numeric literal>

<exponent> ::= <signed integer>

<signed integer> ::= [+ | –] <unsigned integer>
```

Bit String and Hex String Literals

Bit string and hex string literals are represented by a base specifier (B for binary, X for hexadecimal) followed by a string of extended digits (numbers or letters) within single quotation marks.

```
<bit string literal> ::= B'[<bit> …]'

<bit> ::= 0 | 1

<hex string literal> ::= X'[<hexit> …]'

<hexit> ::= <digit> | A | B | C | D | E | F | a | b | c | d | e | f
```

Character String Literals

<character string literal> is arbitrary text bounded at each end by a single quotation mark. All characters except the single quotation mark are allowed in the literal text. Two consecutive single quotation marks in the text represent one literal single quotation mark. The empty string is a valid character string, represented as ' '.

```
<character string literal> ::= '[<SQL language character>…]'

<SQL language character> ::=
            <simple Latin letter>
          | <digit>
          | <SQL special character>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<simple Latin letter> ::=
            <simple Latin upper case letter>
          | <simple Latin lower case letter>

<simple Latin upper case letter> ::=
            A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U
          | V | W | X | Y | Z

<simple Latin lower case letter> ::=
            a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
```

Dates

A <date literal> is similar to a UniVerse date, bounded at each end by a single quotation mark. It must have exactly two occurrences of its delimiter. Only the following characters are allowed as delimiters: - (hyphen), / (slash), and   (space).

Times

A <time literal> is similar to a UniVerse time, bounded at each end by a single quotation mark. It must have exactly one or exactly two occurrences of its delimiter. Only the : (colon) is allowed as a delimiter.

Identifiers and Names

Identifiers must start with a letter. They can include letters, digits, and underscores and be up to 18 characters long. Delimited identifiers can include the space character and all printable 7-bit ASCII characters except the following: the five system delimiters (item mark, field mark, value mark, subvalue mark, and text mark); all control characters (CHAR (0) through CHAR (32)); and the null value
(CHAR (128)).

Identifiers include the following:

<schema name>
<table name>
<view name>
<index name>
<column name>
<association name>
<constraint name>
<correlation name>
<tempname>
<trigger name>

A <schema name> cannot contain a . (period). <column name>s cannot contain a space. A <table name> or <view name> cannot contain a / (slash) or ? (question mark). On UNIX systems a <table name> or <view name> cannot begin with a . (period). On Windows platforms, a <table name> or <view name> cannot contain the following characters: " (double quotation mark), % (percent), * (asterisk), \ (backslash), : (colon), and < > (angle brackets).

User Names

User names must start with a letter (either upper- or lowercase) and can include letters, digits, and underscores. They can be any length allowed on the operating system being used, but 18 characters is the recommended limit. User names include the following:

<user name>
<schema owner>
<grantee>

File Names, Table Names, Field Names, and Column Names

The following kinds of name can be used in a UniVerse SQL SELECT statement when you refer to non-SQL files. You can include a period in these names (normally excluded for SQL applications).

<filename>
<table name>
<fieldname>
<column name>
<datafile>

Value Expressions

An SQL <value expression> is a combination of atomic values called <primary>s. <primary>s can be combined using the arithmetic operators (+, –, *, /), the concatenation operator (| |), and parentheses. The usual rules of arithmetic precedence apply. In programmatic SQL you cannot use parameter markers on both sides of an arithmetic operator, nor can you use parameter markers as the operand of a unary plus or minus.

Primaries

The following rule defines the most general form of <primary>:

```
<primary> ::=
        <column specification>
        | <set function specification>
        | <literal>
        | CURRENT_DATE
        | CURRENT_TIME
        | USER
        | <cast specification>

<cast specification> ::=
        CAST (<cast operand> AS <cast target>)

<cast operand> ::=
        <value expression>
        | NULL

<cast target> ::= <data type>
```

The CURRENT_DATE keyword represents today's date. The CURRENT_TIME keyword represents the current time. The USER keyword represents the current effective user name. For the <literal> rule, see "Literals" on page 5.

The <cast specification> converts data from one data type to another. The <cast target> data type cannot be NCHAR, NVARCHAR, BIT, or VARBIT.

In certain cases some <primary> formats are disallowed. For example, in a <check condition>, the <column specification> must be an unqualified <column name>, and <set function specification>s are not allowed.

Column Specifications

In UniVerse SQL a column specification can be either a qualified or unqualified column name, a qualified or unqualified EVAL expression, or a temporary column name.

```
<column specification> ::=
            | [<qualifier>.]<column name>
            | EVAL[UATE] [<qualifier>.]<i-type specification>
            | <tempname>
```

```
<qualifier> ::=
            | [<schema name>.]<table name>
            | <correlation name>
```

A <column name> is the name of a column (field) in the dictionary of one of the tables (files) named in the <from clause>. If it is not preceded by a <qualifier>, the <column name> must be the name of a column in only one of the tables specified in the <from clause>.

A <qualifier> is either a simple <table name> or a <table name> qualified by a <schema name>.

A <table name> is a simple base table name (without DICT or DATA), or a <correlation name>, that is *exposed* in the <from clause>. This table must contain the specified <column name> (or columns named in the <i-type specification> if this is an EVAL expression).

A <schema name> is a simple schema name. This schema must contain the specified <table name>. The <table name> and <schema name> must each be followed by a period.

An <i-type specification> is a character string, delimited by quotation marks, specifying a valid I-descriptor expression for one of the tables (files) named in the <from clause>. If the <from clause> names more than one table, the <i-type specification> must be preceded by a <qualifier>. In programmatic SQL you cannot use a parameter marker in place of an <i-type specification>.

Set Functions

```
<set function specification> ::=
            COUNT(*)
            | {AVG | MAX | MIN | SUM | COUNT} (DISTINCT <column
                   specification>)
            | {AVG | MAX | MIN | SUM | COUNT} ( [ ALL ] <value
                   expression> )
```

The <value expression> cannot contain a <set function specification>. In programmatic SQL you cannot use a parameter marker in place of a <column specification> or <value expression>.

Set functions are used with explicit or implicit groups.

Groups are formed *explicitly* with the GROUP BY clause (see WHEN Clause). The GROUP BY clause partitions a table into subtables (groups) each of which has the same value for all the specified grouping columns.

In the absence of a GROUP BY clause, if any set function appears in the SELECT clause, the entire table is treated implicitly as a group.

When a table has been explicitly or implicitly partitioned into groups, all operations on the table must deal with groups of rows instead of with individual rows. The set functions are designed to operate on groups of rows, returning a single value from the set of values in each group. Other than set functions, the only other <primary>s that can be specified for groups are (1) grouping columns and (2) constants.

The AVG and SUM set functions can be used only with numeric data. The other set functions can be used with any data type.

Character Value Expressions

Character value expressions can be used wherever a <value expression> is allowed and the data type can be CHAR, VARCHAR, NCHAR, or NVARCHAR.

```
<character value expression ::=
            <concatenation>
            | <character primary>

<concatenation> ::=
              <character value expression> || <character primary>
```

```
<character primary> ::=
            <character value expression primary>
          | <character value function>

<character value expression primary> ::=
            <unsigned value specification>
          | <column specification>
          | (<character value expression>)

<unsigned value specification> ::=
            <character string literal>
          | USER

<character value function> ::=
            <character substring function>
          | <fold>
          | <trim function>

<character substring function> ::=
            SUBSTRING (<character value expression> FROM <start
            position>
            [FOR <string length>])

<fold> ::=
            {UPPER | LOWER} (<character value expression>)

<trim function> ::=
            TRIM ([[<trim specification>] [<trim character>] FROM]
            <character value expression>)

<trim specification> ::=
            LEADING
          | TRAILING
          | BOTH
```

The SUBSTRING function returns a character string starting at <start position> and continuing for the specified <string length> or until the end of the <character value expression>. <start position> and <string length> must evaluate to integers. A <start position> of 0 or a negative number represents an imaginary character position to the left of the first real character; in that case the <string length> counts from the imaginary position, but the returned substring starts at the first real character. If <string length> is negative, SUBSTRING returns an empty string.

UPPER converts the letters a through z to A through Z. LOWER converts the letters A through Z to a through z. The <fold> functions do not convert any other characters.

The TRIM function removes all occurrences of <trim character> from the left, right, or both ends of a character string. The default is BOTH, and the default <trim character> is the space character.

Numeric Value Expressions

Numeric value expressions can be used wherever a <value expression> is allowed and the data type can be DEC, DOUBLE PRECISION, FLOAT, INT, NUMERIC, REAL, or SMALLINT.

```
<numeric value expression> ::=
          <term>
        | <numeric value expression> + <term>
        | <numeric value expression> – <term>

<term> ::=
          <factor>
        | <term> * <factor>
        | <term> / <factor>

<factor> ::=
          [ + | – ] <numeric primary>

<numeric primary> ::=
          <numeric value expression primary>
        | { CHAR[ACTER]_LENGTH } (<character value expression>)

<numeric value expression primary> ::=
           <unsigned value specification>
         | <column specification>
         | <set function specification>
         | (<numeric value expression>)
```

The CHAR_LENGTH function returns the length of a character string as an integer.

Data Types

UniVerse SQL supports fifteen data types:

```
<data type> ::=
        CHAR[ACTER] [ ( <length> ) ]
        | { VARCHAR | CHAR[ACTER] VARYING } [ ( <length> ) ]
        | { NCHAR | NATIONAL CHAR[ACTER] } [ ( <length> ) ]
        | { NVARCHAR | NCHAR VARYING | NATIONAL
        CHAR[ACTER] VARYING } [ ( <length> ) ]
        | BIT [( <length> )]
        | { VARBIT | BIT VARYING } [( <length> )]
        | INT[EGER]
        | SMALLINT
        | FLOAT [ (<precision>) ]
        | REAL
        | DOUBLE PRECISION
        | DEC[IMAL] [(<precision> [, <scale>] ) ]
        | NUMERIC [(<precision> [, <scale> ] ) ]
        | DATE
        | TIME
```

For details about these data types, see Chapter 3, "Data Types."

Tables

In UniVerse SQL, the <table reference> has been expanded to allow for unnesting tables in SELECT statements. The more limited <DML table reference>, which does not include the UNNEST clause or the AS clause, is used in INSERT, UPDATE, and DELETE statements.

```
<table reference> ::=
                <table extension>
                [ <explicit ID specification> ]
                [ <alternate dictionary specification> ]
                [ [ AS ] <correlation name> ]
                [NO.INDEX]

<table extension> ::=
                <base table name>
                | UNNEST <base table name> ON { <column name>
                | <association name> }

<DML table reference> ::=
<base table name>
                [<explicit ID specification>]
                [<alternate dictionary specification>]
                [NO.INDEX]

<base table name> ::=
                { [<schema name>.]<table name>[_<association name>]
                | DICT [<schema name>.]<table name>
                | DATA <filename>,<datafile> }

<explicit ID specification> ::=
                <character string literal> [ { ,<character string literal> } ... ]
                | INQUIRING
                | SLIST { <unsigned integer> | '<list name>' }

<alternate dictionary specification> ::=
                USING DICT [<schema name>.]<table name>
```

A <correlation name> is a temporary synonym for this table name, which can be used elsewhere in a SELECT statement. One use of a <correlation name> is for correlated subqueries. An <association_name> is the name either of an association of multi-valued columns or of an unassociated multivalued column. If the <base table name> includes the _<association name> extension, <correlation name> cannot be the same as a <column name> in the base table.

The <base table name> in an UNNEST clause cannot include the _<association name> extension. In programmatic SQL you cannot use INQUIRING or select list 0 in a DML statement.

You cannot use an <explicit ID specification> in an INSERT statement.

NO.INDEX suppresses all use of a table's secondary indexes.

Query Expressions

<query expression> ::=
    <query term>
    | <query expression> UNION [ALL] <query term>

<query term> ::=
    <query specification>
    | (<query expression>)

Query expressions take many forms. Release 9 of UniVerse uses two forms of query expression, the simplest form of which is the query specification (see Simple Query Specification). Query expressions can be used in two ways:

- As an interactive or programmatic SELECT statement
- As the SELECT statement that creates a view

The following sections describe three kinds of query specification. For information about query specifications that are used in the INSERT and UPDATE statements, see "INSERT Statement" on page 37 and "UPDATE Statement" on page 38.

A query specification takes the form of a SELECT statement. This appendix distinguishes three kinds of query specification:

- The standard SQL SELECT statement
- The interactive SQL SELECT statement
- The interactive report statement

Simple Query Specification

<query specification> ::=
    SELECT [ALL | DISTINCT] <select list> [TO SLIST
    {<unsigned
    integer> | '<list name>'}] <table expression> [<updatability
    clause>] [<processing qualifier list>]

<select list> ::=
    *
    | <value expression> [{,<value expression>}…]
    | {<table name>.* | <correlation name>.*}

```
<updatability clause> ::=
            FOR UPDATE [OF <column specification> [,<column
            specification>] ...]

<processing qualifier list> ::=
            <processing qualifier> [<processing qualifier>...]

<processing qualifier> ::=
            EXPLAIN
           | NO.OPTIMIZE
           | NOWAIT
           | REPORTING
           | SAMPLE [<unsigned integer>]
           | SAMPLED [<unsigned integer>]
```

Description and Rules

The <select list> specifies actual and derived columns to be selected from the intermediate table built by the <table expression>. For details about <table expression>, see “Table Expression” on page 25. For details about <value expression>, see “Value Expressions” on page 9.

In the <select list>, a <column name> is the name of a column (field) in the dictionary of one of the tables (files) named in the <table expression>. If it is not preceded by a <qualifier>, the <column name> must be the name of a column in only one of the tables specified in the <table expression>.

In the <select list>, an <i-type specification> specifies a valid I-descriptor expression for one of the tables (files) named in the <table expression>. If the <table expression> names more than one table, the <i-type specification> must be preceded by a <qualifier>.

In the <select list>, <value expression> can be the NULL keyword.

In programmatic SQL you cannot use a parameter marker in place of a <value expression> in the <select list>.

The keyword DISTINCT means that duplicate rows are eliminated from the result of the column-selection process, or from the rows in a group when computing a set function. DISTINCT can appear only once in a SELECT statement (not counting subqueries). The keyword ALL (the default) means that duplicate rows are not eliminated.

An * (asterisk), when specified instead of a <select list>, means "all columns." <table name>.* means all columns of <table name>. If a <table name> is given a <correlation name> in the <table expression>, you must use <correlation name>.* in the <select list>. You can also use <table name>_<association name>.*, <schema name>.<table name>.*, and <filename>.*, as long as each of these identifiers appear in the <table expression>. <filename>.* means all fields defined in the @ phrase for the file, plus the record ID (unless the @ phrase contains ID.SUP).

If the table dictionary contains an @SELECT phrase, "all columns" means all columns, real (defined by data descriptors) and virtual (defined by I-descriptors and by A- and S-descriptors that use correlatives) listed in the @SELECT phrase for this table. The list of columns in @SELECT takes precedence over the table's SICA as well as the file dictionary's @ phrase. When referring to a UniVerse file that is not a table, "all columns" means all fields defined in the @ phrase for this file, plus the record ID (unless the @ phrase contains ID.SUP). If there is no @ phrase, "all columns" means just the record ID.

The TO SLIST clause creates a UniVerse select list of entries, each of which comprises the columns of the <select list> separated by key separators. If a <query specification> includes the TO SLIST clause, <select list> cannot specify columns containing multivalued data (unless you are selecting from an association); the <table expression> cannot include the <when clause>, <group by clause>, or <having clause>; and the <query expression> cannot include the UNION keyword. If a <query specification> includes TO SLIST and its <table expression> references a single table, you can omit the <select list>.

In the <updatability clause>, the <column specification> cannot be an EVAL expression or @ASSOC_ROW. If a <query specification> includes an <updatability clause>, <select list> cannot include a <set function>, and <table expression> cannot include the <group by clause> or <having clause>. An <updatability clause> cannot be included in a trigger program.

SAMPLE or SAMPLED selects a limited number of output rows before the action of the <order by clause> if one is present. In programmatic SQL you cannot use a parameter marker in place of the <unsigned integer>.

EXPLAIN displays information on the strategy that will be used to process the query. In programmatic SQL this information is returned as error message text.

The keyword NOWAIT terminates and rolls back the SELECT statement when another user's record lock, group lock, or file lock is encountered.

In programmatic SQL you cannot use the keyword REPORTING.

Interactive Query Specification

An interactive query specification is a special kind of query specification that you execute from the UniVerse system prompt. You can use many of the same column functions that are available in RetrieVe.

```
<interactive query specification> ::=
          SELECT [ALL | DISTINCT] <interactive select list> [TO SLIST
          {<unsigned integer> | '<list name>' } ] <table expression>
          [<updatability clause>] [<processing qualifier list>]

<interactive select list> ::=
          <interactive value expression> [ { ,<interactive value
          expression> } … ]
          | *

<interactive value expression> ::=
          <value expression> [<field qualifier list>]
          | [<field modifier>] <column specification> [<field qualifier
           list>]

<field modifier> ::=
          TOTAL
          | { AVERAGE | AVG }
          | { PERCENT | PCT } [ <unsigned integer> ]
          | { BREAK ON | BREAK.ON } [ <character string literal> ]
          | {BREAK SUPPRESS | BREAK.SUP} [<character string
           literal>]
          | { CALCULATE | CALC }

<field qualifier list> ::=
          <field qualifier> [ <field qualifier> … ]
```

<field qualifier> ::=

```
[AS] <tempname>
| { DISPLAYLIKE | DISPLAY.LIKE } { [<qualifier>.]<column
 name>
| <tempname> }
| { DISPLAYNAME | DISPLAY.NAME | COL.HDG }
 <character
string literal>
| { CONVERSION | CONV } <character string literal>
| { FORMAT | FMT } <character string literal>
| { MULTIVALUED | MULTI.VALUE | SINGLEVALUED
| SINGLE.VALUE }
| { ASSOCIATION | ASSOC } <association name>
| { ASSOCIATED | ASSOC.WITH } { [<qualifier>.]<column
 name>
| <tempname> }
```

Description and Rules

The <field modifier>s have the same meaning as in RetrieVe. As the syntax definition shows, a <field modifier> can be used only when the <value expression> is a <column specification> (that is, a column name, an EVAL expression, or a tempname assigned to a column or an EVAL expression). In programmatic SQL you cannot use <field modifier>s.

The <field qualifier>s have the same syntax and semantics as in RetrieVe. When used with a <value expression> that is just a constant, the only meaningful <field qualifier>s are FORMAT and DISPLAYNAME (and their synonyms). Other <field qualifier>s are ignored when used with a constant. In programmatic SQL you cannot use ASSOCIATION, ASSOCIATED, MULTIVALUED, SINGLEVALUED, or their synonyms.

In programmatic SQL you cannot use a parameter marker in place of a <character string literal> after DISPLAYNAME, CONVERSION, FORMAT, or their synonyms.

Interactive Report Statement

An interactive report statement is an extended <interactive query expression> that you execute from the UniVerse system prompt. With it you can order output rows and use many of the report qualifiers that are available in RetrieVe.

```
<interactive report statement> ::=
        <interactive query expression> [<order by clause>]
        [<updatability
        clause>] [<report qualifier list>] [<processing qualifier list>]
```

```
<interactive query expression> ::=
        <interactive query specification> [{SUPPRESS DETAIL |
        DET.SUP}]
        | (<interactive query expression>)
```

ORDER BY Clause

```
<order by clause> ::=
        ORDER BY <sort specification> [ { , <sort specification> } … ]
```

```
<sort specification> ::=
        { <unsigned integer> | <column specification>} [ ASC | DESC ]
```

Description and Rules

The resulting rows of the query are displayed in the order specified by the first <sort specification>. If the first <sort specification> produces duplicate values, the rows are further sorted according to the second <sort specification>, and so on.

An <unsigned integer> can specify a sorting column by its ordinal position in the <interactive select list>. In programmatic SQL you cannot use a parameter marker in place of an <unsigned integer>.

If a <sort specification> is multivalued, the output is sorted according to the RetrieVe conventions for sorting multivalued data. That is, value marks are ignored, and the values are treated as a single field and sorted as a unit.

If the SELECT clause includes the TO SLIST clause but does not include a <select list>, <column specification> must be a <column name>. If the SELECT clause includes a <select list>, <column specification> can be either a <column name> or an <unsigned integer>. If the SELECT clause includes both the DISTINCT keyword and the TO SLIST clause, you cannot include an <order by clause> in the SELECT statement.

Report Qualifiers

```
<report qualifier list> ::=
              <report qualifier> [ <report qualifier> … ]

<report qualifier> ::=
              { HEADING | HEADER } { <character string literal>
              | DEFAULT }
              | { FOOTING | FOOTER } <character string literal>
              | { COUNT.SUP }
              | { SUPPRESS COLUMN HEADING | SUPPRESS COLUMN
              HEADER | COL.SUP }
              | { SUPPRESS DETAIL | DET.SUP }
              | { NOPAGE | NO.PAGE }
              | { DOUBLE SPACE | DBL.SPC }
              | { COLUMN SPACES | COL.SPCS | COL.SPACES }
              [<unsigned
              integer>]
              | { GRAND TOTAL | GRAND.TOTAL } <character string
               literal>
              | MARGIN <unsigned integer>
              | { VERTICALLY | VERT }
              | LPTR [<unsigned integer>]
              | AUX.PORT
```

Description and Rules

RetrieVe-style formatting is available for the output of the SQL SELECT statement via <report qualifier>s. The <report qualifier>s listed previously have the same meaning as in RetrieVe.

If the HEADING keyword is not specified, the output of the query has no heading line and starts on the next line of the display device. If HEADING is specified, the output starts on a new page or at the top of the screen of the display device. HEADING DEFAULT generates the standard RetrieVe heading line.

Table Expression

```
<table expression> ::=
          <from clause>
          [<where clause>]
          [<when clause> [ <when clause>… ]]
          [<group by clause>]
          [<having clause>]
```

Description and Rules

The <from clause> can be thought of as creating an intermediate table, which is the Cartesian product of its various <table reference>s. Later clauses of the SELECT statement work with this intermediate table to produce the final result of the query.

FROM Clause

```
<from clause> ::=
          FROM {<table reference> | <joined table>}
          [{,{<table reference> | <joined table>} }…]

<joined table> ::=
          {<table reference> | <joined table>} [<join type>] JOIN
          <table reference> <join specification>

<join specification> ::=
          ON <join condition>
          | USING (<join column list>)

<join type> ::=
          INNER
          | LEFT [OUTER]
```

Description and Rules

Each <table reference> is said to have an *exposed* name. This is its <correlation name>, if any, otherwise it is its <base table name>. The exposed names in the <from clause> must all be different.

Base Table Names

A <base table name> in a <from clause> can take three forms: <base table name> indicates the data file, DICT <base table name> indicates the file dictionary, and DATA <filename>,<datafile> specifies a data file in a file comprising multiple data files. An SQL SELECT query can be made against any UniVerse file whether or not it is an SQL table. However, if you qualify the <base table name> with a <schema name>, the table cannot be a UniVerse file.

UNNEST Clause

The UNNEST keyword unnests the multivalued data of an association. UNNEST expands each table row by generating an unnested row for each value mark of the unnested association up to the unnested association depth. The unnested association depth is determined (for SQL tables) by the maximum number of value marks in the unnested association key columns, as defined in the SICA, or (for UniVerse files) by the maximum number of value marks in the unnested association phrase columns or in the controlling attribute column. UNNEST acts before all other stages of query processing, and unnested columns become single-valued columns for the rest of the query processing.

Explicit IDs

An <explicit ID specification> names a source of known row identifiers. If the table from which you are selecting data is a view, you cannot use an <explicit ID specification>.

A <character string literal> list specifies that only records whose keys are in the list are considered as candidate rows. Each key must be specified as a quoted literal.

The INQUIRING keyword prompts the user for keys. You can use INQUIRING only once in a SELECT statement, and you cannot use it in a correlated subquery.

When you use the SLIST keyword, candidate rows are those records whose keys are either in a currently active UniVerse select list, specified by a number from 0 through 10, or in a saved select list stored in the &SAVEDLISTS& file, specified by <list name>. If the default select list (select list 0) is active when a UniVerse SQL SELECT is executed, SLIST 0 must be specified for a <table reference> in the <from clause>; otherwise the SELECT gives an error message. Select list 0 is always deactivated after a UniVerse SQL SELECT.

Alternate Dictionaries

An <alternate dictionary specification> allows the use of alternate, stored formatting and conversion definitions for named columns. The alternate dictionary has no effect on the catalog or SICA information that defines the table.

Joins

A <join condition> is one or more <where predicate>s using columns from the <table reference>s in the <joined table>. The <where predicate>s can be combined using AND, OR, NOT, and parentheses, and can be evaluated giving a logical result of true, false, or unknown. The <table reference>s in the <joined table> cannot include the INQUIRING keyword.

WHERE Clause

```
<where clause> ::=
            WHERE <where condition>
```

A <where condition> is one or more <where predicate>s, combined using AND, OR, NOT, and parentheses, that can be evaluated giving a logical result of true, false, or unknown.

<where predicate> ::=

[EVERY] <value expression> <comp op> <value expression>
| [EVERY] <value expression> [NOT] BETWEEN <value expression> AND <value expression>
| [EVERY] <value expression> [NOT] IN (<value list>)
| [EVERY] <value expression> [NOT] { SAID | SPOKEN <literal>
| [EVERY] <value expression> IS [NOT] NULL
| [EVERY] <value expression> [NOT] { LIKE | MATCHES MATCHING } <pattern> [ESCAPE <escape character>]
| [EVERY] <value expression> <comp op> <subquery>
| [EVERY] <value expression> <comp op> { ALL | { ANY | SOME } } <subquery>
| [EVERY] <value expression> [NOT] IN <subquery>
| EXISTS <subquery>

<subquery> ::=

(SELECT { [ALL | DISTINCT] <value expression> | * } <table expression>)

Description and Rules

The <where clause> selects those rows that match the <where condition>, from the Cartesian product of the tables named in the <from clause>.

A <value expression> in a <where clause> cannot contain a <set function specification>.

For more information about <where predicate>s, see "Predicates" on page 34 and "Subqueries" on page 31.

WHEN Clause

<when clause> ::=

WHEN <when condition>

A <when condition> is one or more <restriction criteria>, combined using AND, OR, NOT, and parentheses, that, when evaluated, gives a logical result of true, false, or unknown.

```
<restriction criteria> ::=
            <value expression> <comp op> <value expression>
            | <value expression> [NOT] BETWEEN <value expression>
             AND
            <value expression>
            | <value expression> [NOT] IN ( <value list> )
            | <value expression> [NOT] { SAID | SPOKEN } <literal>
            | <value expression> IS [NOT] NULL
            | <value expression> [NOT] { LIKE | MATCHES |
             MATCHING }
            <pattern> [ESCAPE <escape character> ]
            | <value expression> <comp op> <subquery>
            | <value expression> <comp op> { ALL | { ANY | SOME } }
            <subquery>
            | <value expression> [NOT] IN <subquery>

<subquery> ::=
            ( SELECT { [ALL | DISTINCT] <value expression> | * } <table
            expression>)
```

Description and Rules

The <when clause> extracts rows from an association (called a *when association*) without having to unnest the association first. WHEN excludes specific association rows from the when association.

The SELECT clause must specify at least one output column from the when association.

<restriction criteria> must name at least one multivalued column. All columns named in the <restriction criteria> must belong to the same association.

<restriction criteria> that name unnested columns must also include a multivalued column from outside the unnested association. This is because UNNEST, which makes the unnested columns singlevalued, is applied before WHEN, which must specify at least one multivalued column.

<restriction criteria> in a <when clause> cannot contain a <set function specification>.

You cannot use the <when clause> in a subquery.

For more information about <restriction criteria>, see “Predicates” on page 34 and “Subqueries” on page 31.

GROUP BY Clause

```
<group by clause> ::=
                GROUP BY <column specification> [ { ,<column
                 specification> } ... ]
```

Description and Rules

A <group by clause> partitions a table into subtables, or groups, so that the values of the specified columns are the same in each group. Each group becomes one row in the result table.

Each <column specification> in a <group by clause> is called a “grouping column” and must be singlevalued.

When a SELECT statement includes a <group by clause>, the following rules apply to <value expression>s in the <select clause>:

- Any <column name> in a <value expression> must be a grouping column.
- Any EVAL expression must have an AS <tempname> clause and <tempname> must be a grouping column.
- Any <set function specification> in the <select clause> is applied to each group to produce the final results of the SELECT.

See also “Set Functions” on page 11.

HAVING Clause

```
<having clause> ::=
                HAVING <group condition>
```

A <group condition> is one or more <group predicate>s, combined using AND, OR, NOT, and parentheses, that, when evaluated, gives a logical result of true, false, or unknown.

```
<group predicate> ::=
                    <value expression> <comp op> <value expression>
                    | <value expression> [NOT] BETWEEN <value expression>
                     AND
                    <value expression>
                    | <value expression> [NOT] IN (<value list>)
                    | <value expression> [NOT] { SAID | SPOKEN } <literal>
                    | <value expression> IS [NOT] NULL
                    | <value expression> [NOT] { LIKE | MATCHES |
                     MATCHING }
                    <pattern> [ESCAPE <escape character>]
                    | <value expression> <comp op> <subquery>
                    | <value expression> <comp op> { ALL | { ANY | SOME } }
                    <subquery>
                    | <value expression> [NOT] IN <subquery>
                    | EXISTS <subquery>
```

Description and Rules

A <having clause> selects specific groups in a query. Usually a <having clause> is preceded by a <group by clause>. If it is not, the whole table is considered to be a group.

For more information about <group predicate>s, see “Predicates” on page 34 and Subqueries (next section).

Subqueries

```
<subquery> ::=
                    ( <query specification> )
```

Subqueries are used in four types of predicate, as described in the next sections. The function of the subquery is somewhat different in each case. Generally speaking, a subquery generates intermediate results that are used in the predicate.

A subquery has the form of a simplified SELECT statement, and must be enclosed in parentheses. A subquery (except in an <exists predicate>) must return only one column of values.

<comparison-to-subquery predicate>

```
<comparison-to-subquery predicate> ::=
            <value expression> <comp op> <subquery>

<comp op> ::=
            = | <> | # | < | > | <= | >=
```

In this case, the subquery must return a single cell (row-column intersection). The cell can contain single-valued or multivalued data. The <value expression> is compared against these values.

<in-subquery predicate>

```
<in-subquery predicate> ::=
            <value expression> [NOT] IN <subquery>
```

In this case, the subquery must return an intermediate table comprising a single column of data. If the returned data is multivalued, it is unnested into a set of single values. The predicate (without NOT) is true if the specified expression equals any row in this intermediate table.

"<value expression> NOT IN <subquery>" is equivalent to "NOT <value expression> IN <subquery>".

<quantified predicate>

```
<quantified predicate> ::=
            <value expression> <comp op> { ALL | { ANY | SOME } }
            <subquery>

<comp op> ::=
            = | <> | # | < | > | <= | >=
```

In this case, the subquery must return an intermediate table comprising a single column of data. If the returned data is multivalued, it is unnested into a set of single values. The specified comparison is made against each row in this table. If the keyword ALL is specified, the predicate is true if the comparison is true for every row. With ANY or SOME, the predicate is true if the comparison is true for at least one row.

<exists predicate>

```
<exists predicate> ::=
        EXISTS <subquery>
```

This subquery can return a table of any structure. The predicate is true unless the returned table is empty (has no rows).

You cannot use an <exists predicate> in a WHEN subquery.

Predicates

Predicates are used in <check condition>s, <where condition>s, <when condition>s, <group condition>s, and <join condition>s. A predicate is an expression that, when evaluated, gives a logical result of true, false, or unknown. Most <where predicate>s and <check condition>s also allow the EVERY keyword.

Predicates that do not involve subqueries are described in this section. Predicates involving subqueries are described in "Subqueries" on page 31.

<comparison-to-value predicate>

```
<comparison-to-value predicate> ::=
        <value expression> <comp op> <value expression>

<comp op> ::=
        = | <> | # | < | > | <= | >=
```

The comparison operators <> and # mean "not equal".

In programmatic SQL you cannot use parameter markers in place of both <value expression>s.

<between predicate>

```
<between predicate> ::=
        <value expression> [NOT] BETWEEN <value expression>
         AND
        <value expression>
```

This predicate (without NOT) is true if the first expression is >= the second expression and the first expression is <= the third expression. The second and third expressions must be singlevalued.

In programmatic SQL you cannot use parameter markers in place of both the first <value expression> and the second or third <value expression>.

<in-value-list predicate>

<in-value-list predicate> ::=
    <value expression> [NOT] IN (<value list>)

<value list> ::=
    <literal> [{,<literal>}…]

The predicate "<value expression> IN (value_1, value_2, …)" is equivalent to "<value expression> = value_1 OR <value expression> = value_2 OR …".

In programmatic SQL you cannot use parameter markers in place of both the <value expression> and the first <literal> in the <value list>.

<soundex predicate>

<soundex predicate> ::=
    <value expression> [NOT] {SAID | SPOKEN} <character
    string
    literal>

This predicate (without NOT) is true if the expression (which must be a character string) has the same "sound" as the literal (using the Soundex algorithm).

<null predicate>

<null predicate> ::=
    <value expression> IS [NOT] NULL

This predicate (without NOT) is true if <value expression> is the null value. Note that a <null predicate> is the only correct way to test for NULL. A <comparison predicate> cannot be used to test for NULL.

<like predicate>

<like predicate> ::=
    <value expression> [NOT] {LIKE | MATCHES | MATCHING}
    <pattern> [ESCAPE <escape character>]

<pattern> is a character string literal, in which % (percent sign) means any string of zero or more characters, and _ (underscore) means any string consisting of one character. To use % or _ as a literal character in the search pattern, a user-defined <escape character> must precede the % or _ . An <escape character> is a single-character character string literal and cannot be a %, _ , . (period), ' (single quotation mark), or " (double quotation mark).

Data Manipulation

This section describes statements that add, change, or delete data from tables.

```
<SQL data manipulation statement> ::=
              <delete statement>
              | <insert statement>
              | <select statement>
              | <update statement>
```

DELETE Statement

```
<delete statement> ::=
              DELETE FROM <DML table reference> [<where clause>]
              [<processing qualifier list>]
```

For the <DML table reference> rule, see “Tables” on page 16. For the <where clause> rule, see “WHERE Clause” on page 27. For the <processing qualifier list> rule, see “Simple Query Specification” on page 18.

You cannot use the <processing qualifier>s SAMPLE or SAMPLED in a DELETE statement.

INSERT Statement

```
<insert statement> ::=
              INSERT INTO <DML table reference> [( <column list> )]
              {VALUES ( <attribute list> )
              | <query specification>
              | DEFAULT VALUES }
              [<processing qualifier list>]

<column list> ::=
              <column name> [ { ,<column name> } ... ]

<attribute list> ::=
              <attribute> [ { ,<attribute> } ... ]

<attribute> ::=
              <value expression>
              | <<value expression> [ { ,<value expression> } ... ]>
```

<value expression> ::=
    <value expression> | NULL

For the <DML table reference> rule, see “Tables” on page 16. For the <value expression> rule, see “Value Expressions” on page 9.

Angle brackets enclose an <attribute> comprising multiple <value expression>s. The brackets are part of the syntax and must be typed. When you specify multivalues as an <attribute>, the number and order of values must be the same as the number and order of values in the corresponding association key. If the <DML table reference> includes an <association name>, you cannot specify a multivalued <attribute>.

A <query specification> is a simple UniVerse SQL SELECT statement. For the <query specification> rule, see “Simple Query Specification” on page 18.

UPDATE Statement

```
<update statement> ::=
        UPDATE <DML table reference>
        SET <set clause list>
        [<where clause>] [<when clause>...]
        [<processing qualifier list>]

<set clause list> ::=
        <set clause> [{,<set clause>}...]

<set clause> ::=
        [<qualifier>.]<column name> = <update source>

<update source> ::=
        <value expression>
        |<<value expression> [{,<value expression}...]>
        |NULL
        |DEFAULT
```

Angle brackets enclose an <update source> comprising multiple <value expression>s. The brackets are part of the syntax and must be typed. When you specify multivalues as the <update source>, the number of values must be the same as the number of values in the rows to be updated. If the <DML table reference> includes an <association name> or if the UPDATE statement includes a <when clause>, you cannot specify multivalues as an <update source>.

For the <DML table reference> rule, see "Tables" on page 16. For the <where clause> rule, see "WHERE Clause" on page 27. For the <when clause> rule, see "WHEN Clause" on page 28. For the <value expression> rule, see "Value Expressions" on page 9. For the <processing qualifier list> rule, see "Simple Query Specification" on page 18.

You cannot use the <processing qualifier>s SAMPLE or SAMPLED in an UPDATE statement.

Schema Definition Statements

This section describes statements that create schemas, tables, and views.

```
<SQL schema statement> ::=
            <SQL schema definition statement>
          | <SQL schema manipulation statement>

<SQL schema definition statement> ::=
            <schema definition>
          | <table definition>
          | <view definition>
          | <index definition>
          | <trigger definition>
          | <grant statement>
```

Schema Definition

```
<schema definition> ::=
            CREATE SCHEMA <schema authorization clause> [HOME
            <pathname>] [<schema element>…]

<schema authorization clause> ::=
            <schema name>
          | AUTHORIZATION <schema owner>
          | <schema name> AUTHORIZATION <schema owner>
```

A <schema element> is a CREATE TABLE, CREATE VIEW, or GRANT <table privileges> statement (described in later sections) without its terminating semicolon.

```
<schema element> ::=
            <table definition>
          | <view definition>
          | <index definition>
          | <trigger definition>
          | <grant statement>
```

Description and Rules

The CREATE SCHEMA statement creates a new schema (UniVerse account) in a specified directory, or it catalogs an existing UniVerse account as an SQL schema. The user must have RESOURCE privilege to create a schema for himself, and must have DBA privilege to create a schema for another user.

The <schema name> must be a valid SQL identifier and must not duplicate another schema name. If <schema name> is not specified, AUTHORIZATION <schema owner> must be specified, and the schema name is set to <schema owner>.

AUTHORIZATION <schema owner> can be specified only if the user has DBA privilege or if the user names himself as the schema owner. The <schema owner> must be a user with CONNECT privilege and with the required operating system permissions to write into the directory specified in the HOME clause. If AUTHORIZATION <schema owner> is not specified, the schema owner is set to be the effective user name of the user issuing the CREATE SCHEMA statement.

The HOME clause specifies the directory where the new schema will reside. This directory must exist, and the user must have the required operating system permissions to write into this directory. If there is no HOME clause, the UniVerse account you are logged in to is made into a schema.

Tables, views, indexes, and triggers can be created and table privileges granted within a CREATE SCHEMA statement, as indicated in the syntax definition. Once a schema is created, users with CONNECT privilege and appropriate operating system permissions can log in to this account and use data definition statements such as CREATE TABLE, CREATE VIEW, GRANT <table privileges>, DROP TABLE, and DROP VIEW.

Table Definition

```
<table definition ::=
        CREATE TABLE <table name> [DATA <pathname>] [DICT
        <pathname>] (<table element> [{ ,<table element> }...] )

<table element> ::=
         <table description>
        | <column definition>
        | <association definition>
        | <table constraint definition>
```

This statement creates a new table (UniVerse file with SQL characteristics) in the current schema. Both the data file and its dictionary are created.

The <table name> must not already exist within the schema. (<table name> must not be in the VOC). <pathname> must be absolute. If you do not use the DATA or DICT clauses, the table's data file pathname is <current directory>/<table name>, and the dictionary pathname is <current directory>/D_<table name>.

This statement cannot be used to create a UniVerse file with multiple data files.

The various <table element>s are described in the following sections.

Table Description

```
<table description> ::=
              <type clause>
              | MODULO <unsigned integer>
              | SEPARATION <unsigned integer>
              | <dynamic clause>

<type clause> ::=
              TYPE <unsigned integer>
              | DYNAMIC

<dynamic clause> ::=
              { GENERAL | SEQ NUM | SEQ.NUM }
              | { GROUP SIZE | GROUP.SIZE } <unsigned integer>
              | { MINIMUM MODULUS | MINIMUM.MODULUS }
               <unsigned
              integer>
              | { SPLIT LOAD | SPLIT.LOAD } <percent>
              | { MERGE LOAD | MERGE.LOAD } <percent>
              | { LARGE RECORD | LARGE.RECORD } <percent>
              | { RECORD SIZE | RECORD.SIZE } <unsigned integer>
              | { MINIMIZE SPACE | MINIMIZE.SPACE }
```

Description and Rules

The type of a file is specified either by the TYPE keyword followed by an integer specifying one of 19 file types, or by using the DYNAMIC keyword for type 30 files. The supported file types are 2 through 18, and 30. SQL tables cannot be type 1, 19, 25, or distributed files. If the type is not specified, the file is created as a type 30 file.

The modulo and separation of a hashed file are specified by the keywords MODULO and SEPARATION, followed by an integer from 1 through 8,388,608.

If a table is specified as a type 30 file, the <dynamic clause> can be used to specify various parameters for type 30 files. The keywords defined in the <dynamic clause> work as defined in the CREATE.FILE documentation.

<percent> is an unsigned integer ranging from 0 through 100.

Column Definition

```
<column definition> ::=
            <column name> <data type> [<column description> … ]
            | <column name> SYNONYM FOR <column name> [<report
            format> … ]

<column description> ::=
             <default clause>
             | { SINGLEVALUED | SINGLE.VALUE | MULTIVALUED
             | MULTI.VALUE }
             | <report format>
             | <column constraint definition>

<default clause> ::=
             DEFAULT <default option>

<default option> ::=
             <literal>
             | <signed integer>
             | <signed realnum>
             | NEXT AVAILABLE
             | CURRENT_DATE
             | CURRENT_TIME
             | USER
             | NULL

<report format> ::=
             { DISPLAYNAME | DISPLAY.NAME | COL.HDG }  <character
              string
             literal>
             | { CONVERSION | CONV } <character string literal>
             | { FORMAT | FMT } <character string literal>
```

Description and Rules

The <column definition> clause is for defining the actual data fields in a table, not for defining virtual fields (I-descriptors) or phrases. All columns defined by <column definition>s become entries in the file dictionary generated for this table.

A column can be specified as a SYNONYM FOR another column (which must also be defined in this CREATE TABLE statement). The <report format> clause specifies output formatting for a column. Synonyms can be defined with zero or more <report format> specifications, allowing synonyms to have their own display names and format masks. Synonym definitions go into the file dictionary generated for this table, not into the SQL catalog.

The default <length> for the CHARACTER data type is 1. The default <precision> for the FLOAT data type is 15. The default <precision> for DECIMAL or NUMERIC data is 9, and the default <scale> for DECIMAL or NUMERIC data is 0. <length>, <precision>, and <scale> are all unsigned integers.

If the <default option> is NEXT AVAILABLE, the <data type> must be INT, NUMERIC, or DECIMAL. If the <default option> is CURRENT_DATE, the <data type> must be DATE. If the <default option is CURRENT_TIME, the <data type> must be TIME. If the <default option> is USER, the <data type> must be a character string. If the <default option> is NULL, the <column constraint definition> cannot be NOT NULL. If the <default option> is a <literal>, the <column constraint definition> cannot be NOT EMPTY. If you omit the <default clause>, NULL is the default.

All columns are singlevalued unless MULTIVALUED (or MULTI.VALUE) is specified. The singlevalued attribute creates an implicit column constraint. Any attempt to store multivalued data in the column fails.

<column constraint definition> is described in the next section.

Column Constraints

```
<column constraint definition> ::=
            [ CONSTRAINT <constraint name> ] <column constraint>
```

```
<column constraint> ::=
            NOT NULL [ UNIQUE | ROWUNIQUE ]
            [ NOT NULL ] PRIMARY KEY
            | NOT EMPTY
            | CHECK ( <check condition> )
            | REFERENCES [<schema name>.] <table name> [(<referenced
            column>)] [ON UPDATE <referential action>] [ON DELETE
            <referential action>]

<referential action> ::=
            CASCADE
            | SET NULL
            | SET DEFAULT
            | NO ACTION
```

Description and Rules

Constraints are rules that maintain the integrity of the data in the table. Any attempt to modify the table that violates one of its constraints is prohibited.

You can give a column constraint a name by specifying CONSTRAINT <constraint name> before the definition.

NOT NULL means that the column cannot take on a null value. If the column is multivalued, no individual value can be null.

UNIQUE or ROWUNIQUE specified in a <column constraint> must be immediately preceded by the keyword NOT NULL.

PRIMARY KEY can be specified only once for a table, either as a column constraint or as a table constraint. If it is a column constraint, that column is the table's primary key, which cannot contain duplicate values. The primary key must also be singlevalued.

A UNIQUE column constraint, for a singlevalued column, means that no two rows can have the same value for that column. For a multivalued column, UNIQUE means that no value within any row of the column can be duplicated in the same or another row of that column.

ROWUNIQUE applies only to multivalued columns. It means that the values within each row must be unique but can be duplicated in other rows.

NOT EMPTY means that the empty string (' ') value cannot occur in any row of this column (or any individual value within a row if this column is multivalued).

The CHECK constraint allows an arbitrary test to be made before putting a new value into this column. See the next section.

The REFERENCES constraint means that a nonnull value is allowed in the referencing column (or foreign key) only if the value also exists in the <referenced column>. If no <referenced column> is specified, the value must exist as a primary key in <table name>.

The foreign key column can be singlevalued or multivalued. The referenced column can also be singlevalued or multivalued.

ON UPDATE and ON DELETE respectively define what actions to take when executing an UPDATE or DELETE statement on the referenced table. CASCADE means if a value in the referenced table is updated, the value in the corresponding row of the referencing table is updated. If a row in the referenced table is deleted, the corresponding row in the referencing table is deleted. SET NULL and SET DEFAULT set the corresponding value in the referencing table to the null value or the default value. NO ACTION causes no change to occur in the referencing table.

Check Conditions

A <check condition> is one or more <check predicate>s, combined using AND, OR, NOT, and parentheses, that can be evaluated giving a logical result of true, false, or unknown (using the SQL rules for three-valued logic).

```
<check predicate> ::=
                [EVERY] <value expression> <comp op> <value expression>
                | [EVERY] <value expression> [NOT] BETWEEN <value
                expression> AND <value expression>
                | [EVERY] <value expression> [NOT] IN ( <value list> )
                | [EVERY] <value expression> [NOT] { SAID | SPOKEN }
                 <value
                expression>
                | [EVERY] <value expression> IS [NOT] NULL
                | [EVERY] <value expression> [NOT] { LIKE | MATCHES
                | MATCHING } <pattern> [ESCAPE <escape character>]

<comp op> ::=
            = | <> | # | < | > | <= | >=
```

A <value expression> is one or more <primary>s, combined using the arithmetic operators (+, –, *, /), the concatenation operator (||), and parentheses.

```
<primary> ::=
            <column name>
          | <literal>
          | USER
          | <cast specification>
```

If the <check condition> is defined in a <column constraint>, only that column's <column name> is permitted in any <primary>. If the <check condition> is defined in a <table constraint>, only <column name>s belonging to this table are permitted in any <primary>. In either case, the <column name>s must be defined in this CREATE TABLE statement.

<pattern> and <escape character> are described in the section "<like predicate>" on page 35.

Description and Rules

A table's <check condition>s are tested each time there is an attempt to add or modify data in the table. Only if the result is true will the write operation be allowed.

See "Predicates" on page 34 for more information about <check predicates>.

Association Definition

```
<association definition> ::=
              { ASSOCIATION | ASSOC } <association name> [<position> ... ] (
<column name> [KEY] [ { ,<column name> [KEY] } ... ])

<position> ::=
              { INSERT LAST
              | INSERT FIRST
              | INSERT IN <column name> BY <sequence>
              | INSERT PRESERVING }

<sequence> ::=
              { AL | AR | DL | DR }
```

Description and Rules

The columns named in an <association definition> must be defined in a <column definition> in this CREATE TABLE statement (but not as a synonym), must be multivalued, and must not appear in any other <association definition> for this table.

Within an <association definition>, the keyword KEY identifies the columns that, when combined with the base table's primary key or @ID column, form the primary key of this association's virtual table. All association key columns must be declared NOT NULL, just like primary key columns. If more than one association key column are defined as association keys, and none of them are defined as ROWUNIQUE, they are treated as jointly rowunique at run time.

If only one association key column is specified, that column automatically assumes the ROWUNIQUE property (see "Column Constraints" on page 45).

The <position> determines where new association rows are added. INSERT LAST puts a new association row after the last association row in the base table two. This is the default if you do not specify <position>. INSERT FIRST puts a new association row before the first association row. INSERT IN positions a new association row among existing association rows according to the sequential position of the value in <column name>. The sequence can be ascending left-justified (AL), ascending right-justified (AR), descending left-justified (DL), or descending right-justified (DR). Existing association rows are not sorted.

Table Constraints

```
<table constraint definition> ::=
        [ CONSTRAINT <constraint name> ] <table constraint>

<table constraint> ::=
        { PRIMARY KEY [ '<separator>' ] | UNIQUE }
        (<column name> [ { ,<column name> } ... ])
        | CHECK (<check condition>)
        | FOREIGN KEY (<referencing column> [ { , <referencing
        column> } ... ]) REFERENCES [ <schema name>. ]<table name>
        [ (<referenced column> [ { ,<referenced column> } ... ]) ]
        [ ON UPDATE <referential action> ] [ ON DELETE <referential
        action> ]
```

```
<referential action> ::=
              CASCADE
            | SET NULL
            | SET DEFAULT
            | NO ACTION
```

Description and Rules

A table constraint can be used to define a multicolumn PRIMARY KEY. All primary key columns must be singlevalued. No two rows in the table can have the same values in all primary key columns. That is, the columns must be *jointly unique*.

All primary key columns must be defined in this CREATE TABLE statement. <separator> is a one-character literal specifying the character used in the @ID column to separate the <column name>s of a multicolumn primary key. The default is a text mark (CHAR(251)). Except for the text mark, <separator> must be a member of the 7-bit character set. It cannot be ASCII NUL (CHAR(0)). If you specify <separator>, the PRIMARY KEY clause must include at least two <column name>s.

You can give a table constraint a name by specifying CONSTRAINT <constraint name> before the definition.

For the <check condition> rule, see "Check Conditions" on page 47.

The FOREIGN KEY constraint means that a nonnull value is allowed in the referencing columns (or foreign key) only if the value also exists in the <referenced column>s. If no <referenced column>s are specified, the values must exist as a primary key in <table name>.

If you define only one column as a foreign key, whether it is singlevalued or multivalued, the referenced column can also be singlevalued or multivalued. If you define several columns as a foreign key, they must be singlevalued and the corresponding referenced columns must also be singlevalued. You cannot include a multivalued column in a multicolumn foreign key.

ON UPDATE and ON DELETE respectively define what actions to take when executing an UPDATE or DELETE statement on the referenced table. CASCADE means if a value in the referenced table is updated, the value in the corresponding row of the referencing table is updated. If a row in the referenced table is deleted, the corresponding row in the referencing table is deleted. SET NULL and SET DEFAULT set the corresponding value in the referencing table to the null value or the default value. NO ACTION causes no change to occur in the referencing table.

View Definition

<view definition> ::=
    CREATE VIEW <table name> [(<column list>)] AS <query specification> [WITH [<levels clause>] CHECK OPTION]

<column list> ::=
    <column name> [{,<column name>}…]

<levels clause> ::=
    CASCADED
    | LOCAL

Description and Rules

This statement creates a new view of a table in the current schema. A file dictionary for the view is also created.

The <table name> must not already exist within the schema, and there must not be a file whose file name is either <table name> or D_<table name> in the current account directory. <table name> must not be in the VOC.

A <query specification> is a simple UniVerse SQL SELECT statement, except for the following:

- The <field qualifier>s DISPLAYLIKE, DISPLAYNAME, CONV, FMT, and their synonyms are allowed
- A <tempname> specified in an AS clause cannot be used in subsequent clauses of the <query specification>.
- The <from clause> cannot name a <table name> that is a UniVerse file.

If the view is updatable, the following additional <query specification> rules apply:

- The <query specification> must not contain the UNION keyword.
- The <select clause> must not contain the keyword DISTINCT.
- The <from clause> must identify only one table.
- The <from clause> cannot contain a JOIN clause or an UNNEST clause
- The <table reference> must refer to a base table or an updatable view.
- The <base table name> cannot include the _<association name>; that is, the base table cannot be dynamically normalized.

- The <where clause> cannot contain a <subquery> that references the same table as the <from clause>.
- The <when clause> is not allowed.
- The <group by clause> is not allowed.
- The <having clause> is not allowed.

For the <query specification> rule, see "Simple Query Specification" on page 18.

A <column list> is required if two columns in the SELECT clause have the same name, or if any of the columns is the result of a <select expression>, or if the column is the result of a <set function specification>. If all of these columns are given unique names (using the AS field modifier), the <column list> is optional.

Two column names cannot have the same name in the <column list>. The number of columns in the column list and the SELECT clause must be the same.

Index Definition

```
<index definition> ::=
          CREATE [UNIQUE] INDEX <index name> ON <table name>
          (<columnname> [ASC | DESC]
          [{,<columnname> [ASC | DESC]} ...])
```

Description and Rules

Secondary indexes can be created only on base tables, but not on views, associations, or UniVerse files. indexes can be created on up to 16 columns in a table. <index name> cannot contain **Space** characters. An <index name> concatenated to its corresponding <table name> with a period must be unique within a schema.

<table name> cannot contain **Space** characters or any of the following characters: , (comma), " (double quotation mark), or \ (backslash).

In a unique index, columns must be defined as NOT NULL, cannot be part of the primary key or a unique constraint, and cannot be part of another unique index.

For information about character restrictions in <columnname>, see "Identifier" in Chapter 6, "UniVerse SQL Statements." All columns in a multicolumn index must be singlevalued.

Trigger Definition

```
<trigger definition> ::=
        CREATE TRIGGER <trigger name> <trigger action time>
        <trigger event list> ON <table name> FOR EACH ROW
        CALLING "<triggered BASIC program>"

<trigger action time> ::=
        BEFORE
        | AFTER

<trigger event list> ::= <trigger event> [OR <trigger event] ...

<trigger event> ::= INSERT
        | UPDATE
        | DELETE
```

<trigger name> is an identifier unique in the set of trigger names associated with a table. <table name> is the name of a table in the current schema. <triggered BASIC program> is the name of a normally or globally cataloged BASIC program.

Only the owner of <table name>, or a user with ALTER privilege on <table name>, or a DBA can use the CREATE TRIGGER statement.

Privilege Definition

```
<grant statement> ::=
        GRANT <privileges> ON <table name> TO <grantee>
        [{,<grantee>}...] [WITH GRANT OPTION]

<privileges> ::=
         ALL PRIVILEGES
         | <action> [{,<action>}...]

<action> ::=
         SELECT
         | INSERT
         | DELETE
         | UPDATE [( <column list> )]
         | REFERENCES [( <column list> )]
         | ALTER
```

```
<column list> ::=
        <column name> [ { ,<column name> } … ]

<grantee> ::=
        PUBLIC
        | <user name>
```

Description and Rules

When a table is created, its owner has all possible privileges for that table and no one else has any (except, implicitly, DBAs). The owner can use the GRANT <table privileges> statement to let other users read and/or write his table, and even to pass these privileges to others. Any <grantee> other than PUBLIC must have CONNECT privilege. PUBLIC means all users, including UniVerse users who are not registered as UniVerse SQL users.

Schema Manipulation Statements

This section describes the ALTER TABLE statement and statements that delete schemas, tables, views, indexes, and triggers.

```
<SQL schema manipulation statement> ::=
          <drop schema statement>
        | <alter table statement>
        | <drop table statement>
        | <drop view statement>
        | <drop index statement>
        | <drop trigger statement>
        | <revoke statement>
```

DROP SCHEMA Statement

```
<drop schema statement> ::=
          DROP SCHEMA <schema name> [CASCADE]
```

Description and Rules

DROP SCHEMA deletes the specified schema from the SQL catalog. The CASCADE option deletes all SQL tables belonging to the specified schema. The keyword CASCADE is required except when dropping a schema that contains no SQL tables.

If the schema's VOC file was created by a CREATE SCHEMA statement, the VOC and other basic UniVerse files such as &SAVEDLISTS& are also deleted.

DROP SCHEMA must be issued from a schema other than the one being dropped. The user must either be the <schema owner> of the specified schema or have DBA privilege.

ALTER TABLE Statement

```
<alter table statement> ::=
          ALTER TABLE <table name> <alter table action>
```

```
<alter table action> ::=
            ADD [COLUMN] <column definition>
            | ALTER [COLUMN] <column name> <alter column action>
            | ADD <table constraint definition>
            | ADD <association definition>
            | {ENABLE | DISABLE} TRIGGER {<trigger name> | ALL}
            | DROP CONSTRAINT <table constraint> [<drop behavior>]
            | DROP {ASSOCIATION | ASSOC} <association name>

<column constraint> ::=
            NOT NULL [ROWUNIQUE ]
            | NOT EMPTY
            | CHECK ( <check condition> )
            | REFERENCES [<schema name>.] <table name> [(<referenced
            column>)]

<table constraint> ::=
            UNIQUE (<column name> [{,<column name>}...])
            | FOREIGN KEY (<referencing column> [{, <referencing
            column>}...]) REFERENCES [<schema name>.]<table name>
            [(<referenced column> [{,<referenced column>}...])]
            | CHECK (<check condition>)

<drop behavior> ::=
            RESTRICT | CASCADE

<alter column action> ::=
            SET <default clause> | DROP DEFAULT
```

DROP TABLE Statement

```
<drop table statement> ::=
            DROP TABLE <table name> [CASCADE]
```

Description and Rules

A table can be dropped only by the table’s owner or a DBA. All <table privileges> are automatically revoked when the table is dropped.

The CASCADE option deletes all SQL views that derive from the specified table. The keyword CASCADE is required except when dropping a table that has no views derived from it.

DROP TABLE deletes both the data file and its dictionary. The file must be an SQL table.

DROP VIEW Statement

```
<drop view statement> ::=
            DROP VIEW <table name> [CASCADE]
```

Description and Rules

A view can be dropped only by the view's owner or a DBA. All <table privileges> are automatically revoked when the view is dropped. The CASCADE option deletes all views derived from the specified view. The keyword CASCADE is required except when dropping a view that has no views derived from it.

DROP VIEW deletes the file dictionary associated with the view.

DROP INDEX Statement

```
<drop index statement> ::=
            DROP INDEX <table name>.<index name>
```

Description and Rules

DROP INDEX can be used only against indexes created by the CREATE INDEX statement.

DROP TRIGGER Statement

```
<drop trigger statement> ::=
            DROP TRIGGER <table name> {<trigger name> | ALL}
```

REVOKE Statement

```
<revoke statement> ::=
            REVOKE [GRANT OPTION FOR] <privileges> ON <table
             name>
            FROM <grantee> [{,<grantee>}…]
```

Description and Rules

Users can revoke only the privileges for which the users have the grant option, and cannot revoke privileges from themselves.

For the <privileges> and <grantee> rules, see “Privilege Definition” on page 53.

User Definition Statements

This section describes statements that grant and revoke database and table privileges.

```
<SQL user definition statement> ::=
        <grant database privilege statement>
      | <revoke database privilege statement>
```

Grant Database Privilege Statement

```
<grant database privilege statement> ::=
        GRANT { CONNECT | RESOURCE | DBA } TO <user name>
        [ { ,<user name> } … ]
```

Description and Rules

This statement registers UniVerse SQL users and specifies their database privilege level. GRANT CONNECT registers new UniVerse SQL users. Users must have CONNECT privilege before they can be granted RESOURCE or DBA privilege.

RESOURCE privilege grants permission to create new schemas (UniVerse accounts).

DBA privilege is the SQL equivalent of a UNIX *root* user or Windows Administrator, that is, such a user has permission to execute all SQL statements on all files as if this user were the owner. A user with DBA privilege automatically has RESOURCE privilege. When UniVerse is first installed on UNIX systems, there is only one registered user: either *uvsql*, *root*, or *uvadm*. When UniVerse is first installed on Windows platforms, the first registered user is NT AUTHORITY\SYSTEM. On both systems this user has DBA privilege.

Only a user with DBA privilege can issue the GRANT <user privileges> statement.

On UNIX systems, the <user name> must be a valid entry in the */etc/passwd* file. On Windows platforms, the <user name> must be defined as a valid Windows user.

Revoke Database Privilege Statement

```
<revoke database privilege statement> ::=
            REVOKE { CONNECT | RESOURCE | DBA } FROM <user
             name>
            [ { ,<user name> } … ]
```

Description and Rules

Only a user with DBA privilege can issue the REVOKE <user privileges> statement. A user cannot revoke his own authorities.

If users' CONNECT privilege is revoked, all other privileges granted to them are automatically revoked. Also, all schemas and tables they owned have their ownership changed to the owner of the SQL catalog (on UNIX systems, *uvsql*, *root*, or *uvadm*; on Windows platforms, NT AUTHORITY\SYSTEM).

Calling Procedures

```
<call statement> ::=
        CALL <routine invocation>

<routine invocation> ::=
        <routine name> [<argument list>]

<argument list> ::=
        (<positional arguments>)
        | ( )
        | <UV argument list>

<positional arguments> ::=
        <argument> [{,<argument>} …]

<UV argument list> ::=
        <argument> [<argument> …]
```

Description and Rules

The CALL statement has two syntaxes. The first follows the ODBC pattern, in which a comma-separated list of arguments is enclosed in parentheses. The second follows the UniVerse syntax pattern, in which a space-separated list of arguments not enclosed in parentheses follows the <routine name>.

The CALL statement can be used to call a procedure only in the **SQLExecDirect** and **SQLPrepare** function calls in the BASIC SQL Client Interface and UniVerse Call Interface APIs.

Reserved Words

This appendix lists reserved words in UniVerse SQL. To use them as identifiers in SQL statements, enclose them in double quotation marks.

SQL reserved words (statement names and all keywords) are case-insensitive. You can type them in uppercase, lowercase, or mixed case letters. In this book they are always shown in uppercase letters.

@NEW
@OLD
ACTION
ADD
AL
ALL
ALTER
AND
AR
AS
ASC
ASSOC
ASSOCIATED
ASSOCIATION
AUTHORIZATION
AVERAGE
AVG
BEFORE
BETWEEN
BIT
BOTH
BY
CALC
CASCADE
CASCADED
CAST
CHAR

CHAR_LENGTH
CHARACTER
CHARACTER_LENGTH
CHECK
COL.HDG
COL.SPACES
COL.SPCS
COL.SUP
COLUMN
COMPILED
CONNECT
CONSTRAINT
CONV
CONVERSION
COUNT
COUNT.SUP
CREATE
CROSS
CURRENT_DATE
CURRENT_TIME
DATA
DATE
DBA
DBL.SPC
DEC
DECIMAL
DEFAULT
DELETE
DESC
DET.SUP
DICT
DISPLAY.NAME
DISPLAYLIKE
DISPLAYNAME
DISTINCT
DL
DOUBLE
DR
DROP
DYNAMIC
E.EXIST
EMPTY
EQ
EQUAL

ESCAPE
EVAL
EVERY
EXISTING
EXISTS
EXPLAIN
EXPLICIT
FAILURE
FIRST
FLOAT
FMT
FOOTER
FOOTING
FOR
FOREIGN
FORMAT
FROM
FULL
GE
GENERAL
GRAND
GRAND.TOTAL
GRANT
GREATER
GROUP
GROUP.SIZE
GT
HAVING
HEADER
HEADING
HOME
IMPLICIT
IN
INDEX
INNER
INQUIRING
INSERT
INT
INTEGER
INTO
IS
JOIN
KEY
LARGE.RECORD

LAST
LE
LEADING
LEFT
LESS
LIKE
LOCAL
LOWER
LPTR
MARGIN
MATCHES
MATCHING
MAX
MERGE.LOAD
MIN
MINIMIZE.SPACE
MINIMUM.MODULUS
MODULO
MULTI.VALUE
MULTIVALUED
NATIONAL
NCHAR
NE
NO
NO.INDEX
NO.OPTIMIZE
NO.PAGE
NOPAGE
NOT
NRKEY
NULL
NUMERIC
NVARCHAR
ON
OPTION
OR
ORDER
OUTER
PCT
PRECISION
PRESERVING
PRIMARY
PRIVILEGES
PUBLIC

REAL
RECORD.SIZE
REFERENCES
REPORTING
RESOURCE
RESTORE
RESTRICT
REVOKE
RIGHT
ROWUNIQUE
SAID
SAMPLE
SAMPLED
SCHEMA
SELECT
SEPARATION
SEQ.NUM
SET
SINGLE.VALUE
SINGLEVALUED
SLIST
SMALLINT
SOME
SPLIT.LOAD
SPOKEN
SUBSTRING
SUCCESS
SUM
SUPPRESS
SYNONYM
TABLE
TIME
TO
TOTAL
TRAILING
TRIM
TYPE
UNION
UNIQUE
UNNEST
UNORDERED
UPDATE
UPPER
USER

USING
VALUES
VARBIT
VARCHAR
VARYING
VERT
VERTICALLY
VIEW
WHEN
WHERE
WITH

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 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