UniVerse

SQL Administration for DBAs

Version 10.3
February, 2009

IBM Corporation
555 Bailey Avenue
San Jose, CA 95141

Documentation Team: Claire Gustafson, Shelley Thompson, Anne Waite

Table of Contents

Preface

This manual is for database administrators who want to use the additional functionality of SQL in their UniVerse applications, and who need to administer other users of UniVerse SQL.

This document introduces the of *creating* and *modifying* schemas and tables. It also discusses the use of primary keys, constraints, referential integrity, transaction processing, and security. For a discussion of how to use SQL to query an existing database and to modify its data, see *UniVerse SQL User Guide*.

Organization of This Manual

This manual is organized as follows:

Chapter 1, "The UniVerse Environment," is an overview of the UniVerse environment, including a discussion of the differences between standard UniVerse and SQL.

Chapter 2, "Users, UniVerse Accounts, and Schemas," discusses SQL users, user login accounts, UniVerse accounts, and the differences among them.

Chapter 3, "The Command Processor and the VOC File," discusses the UniVerse command processor and the VOC file.

Chapter 4, "Creating and Dropping Schemas," discusses how to create and drop schemas.

Chapter 5, "Creating, Modifying, and Dropping Tables," describes how to create and modify tables, modify table dictionaries, use triggers for tables, list information about a table, and load bulk data from tables in other databases to UniVerse tables.

Chapter 6, "UniVerse Files and SQL," describes how file dictionaries affect SQL, the advantages and limitations of using SQL tables versus UniVerse files, and how to convert UniVerse files to tables.

Chapter 7, "Ensuring Data Integrity," deals with establishing and maintaining database integrity using column, table, and referential constraints.

Chapter 8, "Maintaining Database Security," describes database security.

Chapter 9, "Transactions, Recovery, and Concurrent Access," introduces the concepts of transaction processing, database recovery, and concurrent access.

Chapter 10, "Transferring Tables Across Schemas," describes how to transfer SQL tables from one schema to another on the same system or from one system to another.

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, **`<Enter>`**).
[ ]	Brackets enclose optional items. Do not type the brackets unless indicated.
{ }	Braces enclose nonoptional items from which you must select at least one. Do not type the braces.
itemA | itemB	A vertical bar separating items indicates that you can choose only one item. Do not type the vertical bar.
...	Three periods indicate that more of the same type of item can optionally follow.
ä	A right arrow between menu options indicates you should choose each option in sequence. For example, "Choose **File** ä **Exit**" means you should choose **File** from the menu bar, then choose **Exit** from the File pull-down menu.
`I`	Item mark. For example, the item mark (`I`) in the following string delimits elements 1 and 2, and elements 3 and 4: `1I2F3I4V5`

Documentation Conventions

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

Documentation Conventions (Continued)

The following conventions are also used:

Syntax definitions and examples are indented for ease in reading.

- All punctuation marks included in the syntax—for example, commas, parentheses, or quotation marks—are required unless otherwise indicated.
- Syntax lines that do not fit on one line in this manual are continued on subsequent lines. 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 Security Features: Describes security features in UniVerse, including configuring SSL through UniAdmin, using SSL with the CallHttp and Socket interfaces, using SSL with UniObjects for Java, and automatic data encryption.

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

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

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

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

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

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

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

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

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

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

Related Documentation

The following documentation is also available:

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

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

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

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

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

API Documentation

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

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

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

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

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

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

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

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

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

Chapter

1

The UniVerse Environment

This chapter introduces the main components that make up the UniVerse system and reviews different kinds of UniVerse databases.

Developers who prefer SQL can use UniVerse SQL to define, query, and control the data in the database (UniVerse files as well as tables) without changing existing UniVerse applications. UniVerse SQL conforms to the ANSI/ISO 1992 standard established for SQL (Entry SQL plus extensions), enhanced to take advantage of the extended relational database structure of UniVerse and seamlessly integrated into the UniVerse environment. It is both a database language and a set of capabilities.

Because this book is written for SQL database administrators (DBAs), it uses SQL terminology to discuss many UniVerse concepts. *UniVerse SQL User Guide* and *UniVerse SQL Reference* both contain information useful for database administrators.

UniVerse tables and files are similar in many ways. Both contain a matrix of data and typically comprise a data file and a file dictionary. An SQL schema is roughly equivalent to a UniVerse account.

Main Features

UniVerse is an integrated database management and application development system that runs on the UNIX and Windows platforms. Some of the more powerful features of UniVerse are the following:

- Variable table sizes, row lengths, column sizes, and number of columns
- Unlimited number of tables and files
- Several table access methods for the most efficient data storage, access, and retrieval
- Database facilities that let you create associations among columns in a table or file
- A dictionary-driven, interactive data entry processor for editing data in tables and files
- Online help for any command in most of the command languages
- SQL data definition and retrieval language
- Facilities to save SQL statements and UniVerse commands for future use and to create stored command sequences
- A powerful programming language (UniVerse BASIC) with built-in database management extensions, including the ODBC-based BASIC SQL Client Interface (BCI)
- Client/server access (UCI (Uni Call Interface), UniVerse ODBC, BASIC SQL Client Interface (BCI), InterCall, UniObjects, UniObjects for Java, and UniOLEDB)

The UniVerse Command Processor

The UniVerse command processor accepts statements and commands from the terminal or other sources and either processes the command itself or calls another UniVerse or system process. When you first enter the UniVerse environment, the command processor is in control of the terminal.

The command processor interprets SQL statements and UniVerse commands, performs certain substitutions on statement and command lines, and passes control to the proper process or utility. Other UniVerse processors, such as the UniVerse Editor, ProVerb, and ReVise, offer sets of commands tailored to their specific functions.

UniVerse also supports a procedural language that allows you to write a program, compile it, and then execute it. In addition to its own functions, UniVerse provides easy access to the operating system.

UniVerse SQL

Using UniVerse SQL you can query and update data in UniVerse tables and files. You can use UniVerse SQL interactively and in local and remote client programs.

UniVerse SQL provides the following enhancements to the UniVerse environment:

- Subqueries that let you nest queries
- Relational joins that let you work with data from more than one table or file in a single statement
- Added database security and integrity

UniVerse SQL conforms to the ANSI/ISO 1992 standard established for SQL, enhanced to take advantage of the extended relational 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 multiple values in a cell. UniVerse also supports nested tables called *associations*, which are made up of a group of related multivalued columns in a table.

Other UniVerse Utilities and Processors

The language processors and the run machine call UniVerse utilities and processes to create and delete UniVerse files, display the system status, locate rows in tables and files, control concurrent access to columns or tables, check table size, and so on. UniVerse also calls on system processes and utilities for certain tasks. These utilities and processes are documented in *UniVerse System Description* and the *UniVerse User Reference*.

The Operating System and UniVerse

UniVerse is one group of programs that runs in the UNIX or Windows operating system environment. However, because the operating system environment can be invisible to the end user, UniVerse can be perceived as the operating environment.

UniVerse has its own command processor, with a command vocabulary that includes some operating system commands and many data management commands that cannot be accessed from the operating system. UniVerse also has its own login procedure and account structure.

Although you can do almost everything from UniVerse, a good understanding of the operating system enhances your use of UniVerse. *UniVerse System Description* explains the processes, utilities, and commands that you use when working in the UniVerse environment.

UniVerse Tables

Tables are implemented as UniVerse files. Every UniVerse table logically comprises one data table and an associated table dictionary. Data tables contain columns that store data values in cells. Each data table also contains an encrypted area called the security and integrity constraints area (SICA) that stores the table's column definitions, integrity constraints, and access permissions. Table dictionaries contain records that define the contents of data tables, as well as the way data is processed and displayed.

The relationship between a data table and its associated dictionary is defined by an entry in the VOC (vocabulary) file that defines the table. Table dictionary records do the following:

- Define columns in the associated data table
- Define different descriptors for data stored in the same column
- Process data stored in columns
- Translate data from other tables
- Define output specifications and report formats

Rows in UniVerse tables are of variable length; so are the columns that make up the rows. The only limit to the number of rows that can be stored in a table is the size of your hard disk. Each row is identified by a unique key called the *record ID*.

UniVerse File Structure

UniVerse tables and files are implemented using operating system directories and files. UniVerse provides several kinds of file organization. This variety of storage methods simplifies application design and provides superior performance.

The following structures are available:

- Type 1 and type 19 files (nonhashed)
- Static hashed files
- Dynamic files
- B-tree files

A data table can be either nonhashed or hashed, depending on the kind of data you want to store. Unless otherwise specified, tables are created as dynamic UniVerse files. Table dictionaries are usually hashed.

The VOC File

Each UniVerse schema has a vocabulary file called the VOC file. The VOC file contains entries that identify every command, keyword, table, and file that you can use while you are in that schema. The command processor uses the VOC file to determine what action to take when you enter a statement.

UniVerse contains an account called UV that is used for system administration. When a schema or UniVerse account is created, the contents of a file called NEWACC in the UV account are copied to the VOC file in the newly created schema or account. This ensures that every new schema and account begins with a correct standard set of commands, sentences, paragraphs, file names, keywords, and menus. You can tailor the copied VOC file to the specified purposes of the schema by adding synonyms for standard commands and keywords, or storing often-used statements. For more information about the VOC file, see Chapter 3, "The Command Processor and the VOC File."

UniVerse SQL Statements and Commands

When you enter the UniVerse environment, the command processor displays a prompt (>). You now can enter any SQL statement or UniVerse command.

For example, when you enter the following statement, the UniVerse command processor looks up SELECT in the VOC file:

```
>SELECT * FROM ENGAGEMENTS.T;
```

Since SELECT * FROM is an SQL statement, control is passed to UniVerse SQL to execute the sentence.

You can store any statement for future use by creating a stored sentence record in the VOC file. You can also store a sequence of statements by creating a paragraph entry. Stored sentences and paragraphs let you repeat a statement or a sequence of statements often.

UniVerse and SQL Databases

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

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

Comparison of Traditional UniVerse Databases to SQL Databases

Database Concepts and Structures

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

Data Models

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

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

The implications of these differences in data modeling and the relational design of SQL are discussed further under "Table and File Structures" on page 11.

UniVerse Tables and Files

Tables are implemented as UniVerse files and can be accessed by UniVerse commands as well as by SQL statements. UniVerse tables and files share the following characteristics:

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

There are also differences. The following table summarizes the relationship between tables and UniVerse files.

	Traditional UniVerse Files	Tables
Created by:	CREATE.FILE	CREATE TABLE
Removed by:	DELETE.FILE	DROP TABLE
Components:	Data file + file dictionary.	Data table + table dictionary. A security and integrity constraints area (SICA) in the data table allows establishment and maintenance of data structure, permissions, and integrity constraints.
Structure:	Fields and records.	Columns and rows.
Accessed by:	UniVerse processors (RetrieVe, ReVise, and so forth), UniVerse BASIC, UniVerse Editor, SQL Client Interface, UCI, and other processes, and SQL statements.	UniVerse processors (RetrieVe, ReVise, and so forth), UniVerse BASIC, UniVerse Editor, SQL Client Interface, UCI, and other processes, and SQL statements.
Security:	Operating system permissions (read/write) granted and revoked by owners/groups/others.	In addition to operating system permissions, more extensive privileges—SELECT, INSERT, UPDATE, DELETE, ALTER TABLE, and REFERENCES Privilege—can be granted on tables.

Comparison of Traditional UniVerse Files to Tables

	Traditional UniVerse Files	Tables
Data integrity:	Checked during certain conversions.	Integrity constraints can be defined, which are enforced for all attempted writes.
Primary keys:	CREATE.FILE allows for only single-column record IDs.	CREATE TABLE allows for both single- and multicolumn primary keys.
Data types:	Not native to UniVerse, but certain output conversion and formatting codes can be included in a field definition.	An essential part of column definitions, and associated with precise default characteristics such as a restricted character set, alignment, and so forth.

Comparison of Traditional UniVerse Files to Tables (Continued)

Table and File Structures

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

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

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

Security and Authorization

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

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

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

Data Integrity

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

Primary Keys

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

Data Categories and Data Types

Unlike a field in a UniVerse file, a column in a table is defined as belonging to a particular data type. A data type defines a column in terms of the valid set of data characters that can be stored in the column, the alignment of the data, conversion characteristics, and so on. The *UniVerse SQL Reference* discusses data types extensively.

Data types can be grouped into seven data categories. The following table summarizes these data categories.

Data 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 2-28-95 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 bit string, without regard for characters.

UniVerse SQL Data Categories

The next table summarizes data types

Data Type	Description
BIT	Stores bit strings.
CHARACTER	Stores character strings (any combination of numbers, letters, and special characters).
DATE	Stores dates.

UniVerse SQL Data Types

Data Type	Description
DECIMAL	Stores decimal fixed-scale (fixed-point) numbers (same as NUMERIC).
DOUBLE PRECISION	Stores high-precision floating-point numbers.
FLOAT	Stores floating-point numbers.
INTEGER	Stores whole decimal numbers.
NCHAR	Stores national character strings.
NVARCHAR	Stores variable-length character strings.
NUMERIC	Same as DECIMAL.
REAL	Stores floating-point (real) numbers.
SMALLINT	Stores small whole decimal numbers.
TIME	Stores times.
VARBIT	Stores variable-length bit strings.
VARCHAR	Stores variable-length character strings.

UniVerse SQL Data Types (Continued)

Using UniVerse Help

There are three kinds of UniVerse online help:

- UniVerse Online Library
- Windows Help
- UniVerse command line help

UniVerse Online Library

UniVerse is shipped with a complete set of UniVerse documentation on CD-ROM. To read the manuals, use the Adobe Acrobat Reader.

Windows Help

Windows Help is available for some client applications running on platforms. Help is available for the following applications and APIs:

- UniAdmin
- InterCall
- UCI
- UniObjects

When you use Windows Help for any of these applications, you get the standard Help features, such as the menu bar and buttons. The following table describes the function of each Help button:

Use this button...	To...
Contents	View the Help Contents.
Index	View the Help Index.
Search	Find information about a help topic you specify.
Back	Return to the previously displayed topic.
Print	Print the currently displayed topic.

Windows Help Buttons

Use this button...	To...
Options	Display a list of Help options.
<<	Move to the previous topic in the browse sequence.
>>	Move to the next topic in the browse sequence.
See Also	View a list of related topics.

Windows Help Buttons (Continued)

UniVerse Command Line Help

Use the UniVerse HELP command to get help about any UniVerse command, keyword, UniVerse BASIC statement, and so forth. For information about the HELP command, enter **HELP HELP** at the UniVerse prompt. Enter **HELP SQL** to display a list of entries, which are topics and commands about which you can get HELP information:

```
>HELP SQL
```

Use the **Up** and **Down Arrow** keys to navigate through this list. Press **ENTER** to choose an item and display explanatory text. Press **ESC** to exit the list.

Use the following syntax to display help about a specific SQL statement:

HELP SQL *statement*

When you use the HELP command, explanatory text appears with menu choices at the bottom of the screen. There are three choices:

Choice	Action
More	Shows the next page of HELP text.
List Commands	Displays a list of entries.
End Help	Exits HELP.

HELP Menu Choices

Inside the HELP display screen, use the **Left** and **Right Arrow** keys to select a choice. Press **ENTER** to choose your selection.

Review of Terms

This chapter introduced the main components that make up the UniVerse system. Subsequent chapters describe these components in more detail. The following table summarizes the more important terms used in this book:

Term	Definition
column	A logical subdivision of a row that can contain data values. Same as *field*.
command processor	The UniVerse processor that interprets SQL statements and UniVerse commands and either executes them or passes them to the appropriate UniVerse or system process or utility. When other processing finishes, control usually returns to the command processor.
data table	A table, associated with a dictionary, containing rows that store data values in columns.
dictionary	A file that defines the contents and structure of the data table with which it is associated. The dictionary of *tablename* is an operating system file that is usually named D_*tablename*.
field	A logical subdivision of a record that can contain data values. Same as *column*.
keyword	An element of an SQL statement that modifies the action of the initial verb. For example, WHERE is a keyword that modifies the verb SELECT. Arithmetic, relational, and logical operators are also keywords.
phrase	In UniVerse SQL, a record in a table dictionary that defines an association of multivalued columns or specifies the default columns to display or insert data in.
primary key	The values in one or more columns that uniquely identify each row in a table.
record	A sequence of related data elements in a file. Same as *row*.
record ID	In UniVerse files, the key used to gain access to a record in a file. Each record ID must be unique in any file.

UniVerse Terms

Term	Definition
row	A collection of related data values stored as a record in a UniVerse file. Every row has an explicit or implicit primary key. The primary key can comprise one or more columns containing data values. Same as *record*.
schema	A group of related tables and files that are listed in the SQL catalog. Schemas are roughly equivalent to UniVerse accounts.
SICA	Security and integrity constraints area. 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 and users in the system. The SQL catalog is located in the CATALOG schema.
table	A matrix of rows and columns containing data. Tables are roughly equivalent to UniVerse data files.
UniVerse account	Working environment defined by a VOC file and all its related tables and files. When users invoke the UniVerse environment, they initiate a UniVerse session in the UniVerse account defined by the VOC file in the current working directory.
UniVerse file	A file that logically comprises a file dictionary and at least one data file. The relationship between the dictionary and the data file is defined by an entry in the VOC file.
VOC file	The master file in a schema or UniVerse account. The VOC file contains records that identify all commands, sentences, paragraphs, files, keywords, procs, and menus that you can use when you are logged on to the schema.

UniVerse Terms (Continued)

The Sample Database

Later chapters use the sample database called Circus, introduced in *UniVerse SQL User Guide*, to demonstrate some aspects of UniVerse SQL. The database comprises 10 tables. It is designed to demonstrate the use of industry-standard SQL access within UniVerse, SQL extensions implemented for UniVerse's multivalued column associations, and nested tables.

Chapter

2

Users, UniVerse Accounts, and Schemas

This chapter describes the following:

- User accounts (login accounts), UniVerse accounts, schemas, and the differences among them.
- Different kinds of UniVerse and SQL users.
- How to change certain UniVerse SQL environment variables.

In this manual, login accounts are called *user accounts*, accounts in the UniVerse environment are called *UniVerse accounts*, and UniVerse SQL accounts are called *schemas*.

User Accounts

Users typically log on first to the operating system. In order to log on to the operating system, each user must have a *user account* defined for them on that system. Users log on to the operating system by entering their user name and password.

At the operating system level, users generally have a home directory defined for them under which they can create their own private files. User accounts are like personal working environments that stay with users no matter where they are working on the system.

Once users log on to the system, they have access to all directories and files on the system, except those protected by permissions. Users can change their current working directories without changing other aspects of their working environment. Tables, files, and commands in other directories can be accessed by entering the path identifying the location in the file system's complete directory tree.

UniVerse Accounts

UniVerse users work in the UniVerse environment, either in standard UniVerse accounts or in special UniVerse accounts called *schemas*. Depending on how their accounts are set up, users either log on first to the operating system by entering their user name and password, then log on to a UniVerse account or schema. Or they log on directly to a UniVerse account or schema, bypassing the operating system.

Schemas and UniVerse accounts are more self-contained than operating system user accounts. The VOC file contained in each UniVerse schema and account defines all the tables and files, and all the statements and commands, that are available to users who are logged on to the schema.

Any UniVerse action or event, whether it is an interactive user session, a UniVerse BASIC program, or a server activity invoked by a client user or program, always involves one user who is attached to (or working in) one schema or UniVerse account.

Schemas

- Schemas are UniVerse accounts with special properties:
  - They are registered in the SQL catalog.
  - SQL users can create tables in them.

A schema, like a UniVerse account, is located in a directory containing a VOC file and other special UniVerse files.

Updating a Schema or UniVerse Account

The RELLEVEL entry in the VOC file of a schema contains the current release level of UniVerse. Each time a user logs on to the schema, this entry is checked to make sure the schema is up to date. If the schema is not current, the user can update it. To display the RELLEVEL entry, enter **`.L RELLEVEL`** at the UniVerse prompt.

Use the UPDATE.ACCOUNT command to update a schema. This command works only on changes made within the current release level of UniVerse.

Schema Structure

A UniVerse SQL database can comprise one or more schemas. A schema comprises one or more tables and their dictionaries. Data tables and their dictionaries are implemented using operating system files and directories. The path of each data table and its dictionary are contained in the VOC file entry for that table.

SQL Users

Ordinary UniVerse users (that is, those not defined as SQL users) can use the four SQL data manipulation language (DML) statements SELECT, INSERT, UPDATE, and DELETE. These statements can be used by any UniVerse users on the tables and files they have permission to access, in the schemas and UniVerse accounts to which they have access.

SQL users are defined to the system as having CONNECT database privilege, which allows them to do the following:

- Create their own tables and views
- Grant other users the right to access their tables and views
- Access tables and views owned by other SQL users if they have the necessary privileges
- Modify or delete their own tables and views

In addition to these tasks, SQL users with RESOURCE database privilege can also create and delete their own schemas.

SQL users with DBA database privilege have access to all schemas, tables, and views on the system. They can register and unregister SQL users and can modify and delete all schemas, tables, and views.

Setting the SQL Environment for a Session

Use the SET.SQL command to set SQL environment variables and other aspects of the SQL environment dynamically at run time. SET.SQL can do the following:

- Set the isolation level
- Turn first-normal-form mode on or off
- Turn empty-null mapping on or off
- Specify how long the system waits on a lock before returning an error
- Turn optimistic scanning on or off
- Specify the size of the select list buffer
- Specify the size of the join buffer
- Turn caching of VOC file verbs and keywords on or off

The syntax is:

SET.SQL { *options* }

For complete details about the options, see the *UniVerse User Reference*.

ODBC applications can use the CALL statement to run SET.SQL to do the following:

- Specify how long the system waits on a lock before returning an error
- Turn optimistic scanning on or off
- Specify the size of the select list buffer
- Specify the size of the join buffer
- Turn caching of VOC file verbs and keywords on or off

Chapter

3

The Command Processor and the VOC File

This chapter describes the UniVerse command processor and how it uses the VOC file.

The Command Processor

The command processor examines and processes every line entered at the system prompt, including all SQL statements and other UniVerse commands, all statements and commands from a stored command sequence, UniVerse BASIC programs, and so on.

The command processor parses a command and searches for the verb (the first word in the command line) in the VOC file. Depending on the verb, the command processor either executes the command or calls the proper processor to complete the execution. The actions taken by the command processor and other processors depend on definitions in the VOC file.

The command processor also lets you store sentences and paragraphs (a sequence of sentences) in the VOC file for execution later.

The command processor maintains a list of the most recent command lines entered at the system prompt. This list is called the *sentence stack*. You can use the sentence stack to recall, delete, change, or reexecute a previous command, or to save a sentence or paragraph in the VOC file.

For detailed information about VOC file entries, see the *UniVerse System Description*.

Special Character Interpretation

The UniVerse command processor recognizes special control characters that delimit fields, values, and subvalues in stored data. The following table lists these control characters.

Control Character	Description	Meaning	Value
Ctrl-^	The **Control** key and the **Caret** (or up-arrow) key	Field mark	^254
Ctrl-]	The **Control** key and the right bracket key	Value mark	^253
Ctrl-	The **Control** key and the **Backslash** key	Subvalue mark	^252
Ctrl-t	The **Control** key and the **t** key	Text mark	^251
Ctrl-n	The **Control** key and the **n** key	SQL NULL	^128

Control Characters

For more information about special characters in UniVerse, see the *UniVerse System Description*.

The SQL+ Prompt

When you enter an SQL statement at the UniVerse prompt (>), you terminate the statement with a ; (semicolon). If the statement is too long to fit on a single line, you can press **ENTER** to continue the statement on the next line. The command processor recognizes **ENTER** as a line continuator and displays an SQL-specific prompt (`SQL+`) until you terminate the multiline statement with a ; followed by **ENTER**, at which point it executes the statement.

Called Procedures

SQL client programs can use the SQL CALL statement to invoke UniVerse commands, paragraphs, stored sentences, and UniVerse BASIC programs and subroutines on a server system. For information about called procedures, see the *UniVerse BASIC SQL Client Interface Guide* and the *UCI Developer's Guide*.

The VOC File

Every schema has a VOC file containing a record for every verb, keyword, table name, stored sentence, and paragraph that you can use while in the schema. Records in the VOC file are also called VOC entries.

The VOC file is a UniVerse data file and has a dictionary associated with it. The names of the verbs, tables, keywords, and other items defined in the VOC file are unique IDs of records in the VOC file. For example, SELECT is the record ID for the SQL statement SELECT, and an equal sign (=) is the record ID for the arithmetic operator *equals*.

The VOC file contains entries for the following SQL statements:

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

The VOC file also contains the following UniVerse commands relevant to SQL users. These commands are documented in the *UniVerse User Reference*:

Command	Description
CONNECT	Establishes a connection to a local or remote UniVerse server.
CONVERT.SQL	Converts a UniVerse file to a table.
LIST.SICA	Displays SICA[1] information about a table.
SET.SQL	Defines certain attributes of the SQL environment.
VERIFY.SQL	Verifies and fixes SQL catalog inconsistencies.

UniVerse Commands for SQL Users

1. Security and integrity constraints area

Some VOC Entry Types

This section describes some of the VOC entry types that DBA administrators should know about. For more information about VOC entries, see the *UniVerse System Description*.

V: Verb

SQL statements are implemented as UniVerse verbs. A *verb* is a UniVerse command. The VOC entry for a statement specifies the processor that the statement invokes, the dispatch type, and the flags that the processor uses. The record ID of the VOC entry is the statement or command itself (such as CREATE or SELECT). To display all verbs in the VOC file, use the LISTV command.

S: Stored Sentence

A stored sentence is a complete SQL statement or UniVerse command. A sentence can include a table name, column names, selection and sort expressions, and keywords.

You can store sentences using the UniVerse sentence stack Save command (.S), the UniVerse Editor, or an INSERT statement. If you use the Editor or an INSERT statement, be sure to identify the VOC entry as a sentence by putting type code S in field 1. To examine the currently stored UniVerse sentences in the VOC file, use the LISTS command.

If you often use the same statement and want to avoid typing it each time, save the statement as an entry in the VOC file. A stored sentence can be one of the following:

- A complete SQL statement or UniVerse command
- The name of another stored sentence
- The name of a paragraph

PA: Paragraph

A *paragraph* is a series of SQL statements, other UniVerse commands, or both, stored together under one name. Paragraphs let you execute several statements and commands by entering the name of the paragraph at the system prompt or including it in a CALL statement in your client program.

The VOC paragraph entry contains the statements and commands that make up the paragraph. The paragraph can also contain special control statements and inline prompting to request input from a user when the paragraph is executed. The record ID of the VOC entry is the paragraph name.

The command processor executes each SQL statement and command in order, just as if each were entered at the keyboard. For more detailed information about paragraphs, see the *UniVerse System Description*.

F: File Pointers

UniVerse uses two kinds of VOC entry to point to tables and files: F-descriptors and Q-pointers. F-descriptors can point either to local tables and files stored in the same schema or account as the VOC file, or to remote tables and files stored in other schemas and accounts.

The VOC F-descriptor entry specifies the paths of the file dictionary and its associated data file. The record ID of the entry is the table or file name.

The VOC file descriptor entry contains the following:

Line	Field Contents
001:	F [*description*]
002:	*O/S path of the data table* (usually the same as the table name)
003:	*O/S path of the file dictionary* (usually the same as the table name with the D_ prefix)
004:	[M]

F-descriptor Fields

M in field 4 indicates that the table comprises multiple data files.

Q: Q-Pointer

A *Q-pointer* points to another file descriptor either in the local VOC file or in the VOC file of another schema or UniVerse account. The VOC Q-pointer entry contains the following:

Line	Field Contents
001:	Q [*description*]
002:	[*account*]
003:	*tablename*

Q-descriptor Fields

account is the name of a remote schema or UniVerse account. If *account* is blank, the file descriptor pointed to is assumed to be in the VOC file of the local schema or account. *tablename* is the record ID of the table's file descriptor in the VOC file of *account*.

K: Keyword

A *keyword* defines an operation that is to occur in the statement or modifies the action of a statement. Some examples of keywords in UniVerse are AS, UNION, and FROM. The VOC file entry for a keyword specifies the internal operation number for that keyword. The record ID of the VOC entry is the keyword itself.

UniVerse Sentence Stack

The command processor stores a copy of any SQL statement or UniVerse command that you enter at the system prompt. By default, the sentence stack preserves up to 99 sentences from your current session (the size of the sentence stack is configurable by the system administrator). Each sentence is numbered from 01 through 99. The most recently entered sentence is 01, the oldest is 99.

You can use sentence stack commands to list, save, edit, and delete sentences, and to recall, execute, and insert new sentences in the stack. The sentence stack commands are documented in the *UniVerse System Description*.

Chapter

4

Creating and Dropping Schemas

This chapter describes how to create and drop schemas.

What Is a Schema?

A *schema* is a special UniVerse account that is listed in the SQL catalog. A schema is a collection of related tables and views. Once you have created a schema, you can log on to it and begin creating tables and views. You can also create UniVerse files in a schema.

A schema is part of a hierarchical structure: schemas in the catalog, and tables, views, and files in schemas. From a practical viewpoint, one advantage to such a hierarchy is the ability to qualify like-named objects by their next higher level and thus make them unique. Tables and views in the same schema must have unique names, and the same rule applies to schemas in the catalog, but beyond that the duplication of names is common. Thus, if a table in the HEADQUARTERS schema and a table in the REGION schema are both named EMPLOYEES, you can qualify their names by their schema names, HEADQUARTERS.EMPLOYEES and REGION.EMPLOYEES, to indicate which table you want.

The SQL Catalog

The SQL catalog is a schema named CATALOG containing six tables that describe all SQL objects on the system (schemas, tables, views, columns, and associations), as well as SQL users and the privileges they have. These tables are:

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

You can retrieve data from these tables just as you can from the other tables in your database, but you cannot add, change, or delete anything in them. DBAs can, however, use the CONFIGURE.FILE and RESIZE commands on these files to change their dynamic file parameters.

Every VOC file contains file pointers to the SQL catalog tables, so you can refer to them without using a qualifier. For example, you can refer simply to UV_TABLES—you need not enter CATALOG.UV_TABLES.

Creating a Schema

You can define a schema in the UniVerse environment in two ways:

- Convert an existing UniVerse account to an SQL schema
- Create a new schema

Whichever method you use, any existing files in the schema directory are unaffected by the creation of the schema.

To create a schema, issue a CREATE SCHEMA statement. If you are logged on to an existing UniVerse account and you do not include the HOME clause (HOME *path*), the UniVerse account you are currently logged on to is converted to a schema. If you include the HOME clause, you can specify a UniVerse account other than the one you are logged on to.

For example, to convert the current UniVerse account to a schema and assign it the schema name CIRCUS, enter:

```
>CREATE SCHEMA CIRCUS;
```

To create a new schema, you must use the HOME clause to specify the directory where you want the schema to reside. For example:

```
>CREATE SCHEMA CIRCUS HOME /usr/tom;
```

/usr/tom is the full path of an existing directory. This statement creates the schema CIRCUS in an empty directory, *tom*, and then sets up a UniVerse account, complete with a VOC and other necessary files.

You can include CREATE TABLE, CREATE VIEW, and GRANT statements as part of the CREATE SCHEMA statement. To reduce possible confusion, it is better to add those as separate steps later.

Note: You must have RESOURCE Privilege to create a schema for yourself, and you must have DBA Privilege to create schemas for others.

Dropping a Schema

The owner of a schema or a DBA can drop a schema and all its tables and views by issuing a DROP SCHEMA statement from a schema other than the one to be dropped. To drop the CIRCUS schema, go to another schema (or any UniVerse account) and enter:

```
>DROP SCHEMA CIRCUS CASCADE;
Deleting SCHEMA CIRCUS
```

If there are tables in the schema, you must include the CASCADE option. The alternative is to drop the tables before you issue the DROP SCHEMA statement.

If the schema is a converted UniVerse account, the schema reverts to a normal UniVerse account and no UniVerse files are deleted.

If the UniVerse account was originally created by a CREATE SCHEMA statement, DROP SCHEMA deletes any UniVerse files (such as &SAVEDLISTS&, VOC, and VOCLIB) created by CREATE SCHEMA. If the account contains other UniVerse files, such as UniVerse data files, they remain undisturbed.

Chapter

5

Creating, Modifying, and Dropping Tables

This chapter describes how to define the following:

- Tables
- Columns in tables
- Table indexes

It also discusses how to:

- Modify tables
- Create triggers for tables
- Get information about UniVerse tables (SICA and the SQL catalog)
- How to modify table dictionaries
- Load bulk data from tables in other databases to UniVerse tables

Creating a Table

Once you have created a schema, the next step is to populate it with tables.

Like a UniVerse file, a table contains columns and rows, and comprises a data file and a file dictionary. In addition, a CREATE TABLE statement defines data integrity constraints, which are stored in a security and integrity constraints area (SICA) that is part of the table's header information.

The CREATE TABLE statement is powerful and complex. In its most basic form CREATE TABLE names the table and includes a list of the columns and their characteristics (including column names, data types, and any necessary formats and conversions):

```
CREATE TABLE tablename (columnname1 description,
columnname2 description, …);
```

To use the CREATE TABLE statement to create the table OLD_ENGAGEMENTS.T into which you copy all 1994 dates from the ENGAGEMENTS.T table, enter:

```
>CREATE TABLE OLD_ENGAGEMENTS.T
SQL+(LOCATION_CODE CHAR(7),
SQL+"DATE" DATE FORMAT '10L' CONV 'D2/',
SQL+GATE_REVENUE DECIMAL(9,2) FORMAT '12R' MULTIVALUED,
SQL+RIDE_REVENUE DECIMAL(9,2) FORMAT '12R' MULTIVALUED,
SQL+CONC_REVENUE DECIMAL(9,2) FORMAT '12R' MULTIVALUED,
SQL+PRIMARY KEY(LOCATION_CODE, DATE));
Creating Table "OLD_ENGAGEMENTS.T"
Adding Column LOCATION_CODE
Adding Column DATE
Adding Column GATE_REVENUE
Adding Column RIDE_REVENUE
Adding Column CONC_REVENUE
```

Note: You must enclose the column name DATE in double quotation marks, because DATE is an SQL reserved word..

To use the CREATE TABLE statement to define a new table NEWTAB.T that uses data from columns in the ENGAGEMENTS.T, ACTS.T, and PERSONNEL.T tables, enter:

```
>CREATE TABLE NEWTAB.T
SQL+(ENG_ID CHAR(7),
SQL+ENG_DATE DATE FORMAT '10L' CONV 'D2/',
SQL+ACT_DESC VARCHAR FORMAT '6T',
SQL+EMP_ID INT FORMAT '5L',
SQL+EMP_NAME VARCHAR FORMAT '25T',
SQL+ACT_PAY DECIMAL(5,2) FORMAT '10L' MULTIVALUED,
```

```
SQL+PRIMARY KEY(ENG_ID, ENG_DATE, ACT_DESC, EMP_ID));
Creating Table "NEWTAB.T"
Adding Column ENG_ID
Adding Column ENG_DATE
Adding Column ACT_DESC
Adding Column EMP_ID
Adding Column EMP_NAME
Adding Column ACT_PAY
```

Everything after *tablename* is enclosed in one set of parentheses, including file type, column definitions, association definitions, and constraint definitions.

Naming a Table

Table names in UniVerse can contain any character except CHAR(0), including spaces and control characters. However, when you use special characters, enclose the table name in quotation marks. We recommend that table names contain only letters, numbers, and underscores, and that they be no longer than 18 characters. For compatibility with existing UniVerse file names, table names can contain periods. If the table name contains only letters, numbers, _ (underscore) and . (period), quotation marks are not needed to delimit the string.

For example, enter the following to name a table SALES.TRACKING:

```
>CREATE TABLE SALES.TRACKING
```

UniVerse usually uses the table name as the path of the data table, and the table name prefixed with D_ as the path of the dictionary. The following section describes instances where this may not be the case.

Spaces in a Table Name

If you use spaces in your table names, you must enclose the entire table name in quotation marks to tell UniVerse that the space is part of the name. For example:

```
>SELECT AMOUNT FROM "SALES TODAY" SQL+WHERE AMOUNT >= 2000;
```

For detailed information about how to use special characters in table names, see the *UniVerse SQL Reference*.

Length of a Table Name

You can enter up to 255 characters as a table name. However, there may be limitations on the length of paths at your site. We recommend that you use table names no longer than 18 characters.

On UNIX systems that allow only 14-character file names, UniVerse truncates the file name to 9 characters and adds a 3-digit sequencer.

On Windows platforms, UniVerse is designed to run in the NT file system (NTFS). But Windows platforms also support the DOS FAT file system, which limits file names to 8 characters with a 3-character extension and has a further set of characters that are not permitted in file names. UniVerse makes no special provision for these file names or special characters. If you want to store tables in a DOS FAT file system, you must follow the DOS conventions when you name tables.

On other systems the length of the table names is system-dependent. The UniVerse table name is not affected; only the paths are transformations of the UniVerse table name.

CREATE TABLE fails with table names greater than 255 characters. You see the following error message:

```
Attempted WRITE with record ID larger than 255 characters.
*** Processing cannot continue. ***
```

Defining the File Type

UniVerse supports 21 different file types. A file type is specified by numbers 1 through 19, 25, or 30.

- Type 1 and type 19 files are nonhashed files used to contain UniVerse BASIC programs and other data organized into records that are loosely structured. Tables cannot be type 1 or type 19 files.
- Types 2 through 18 are static hashed files. Different hashing algorithms are designed to distribute records evenly among the groups of a file. The distribution is based on characters and their positions in the record IDs.
- Type 25 is a B-tree file. Tables cannot be type 25 files.
- Type 30 is a dynamic hashed file.

Like the UniVerse CREATE.FILE command, CREATE TABLE lets you choose the type of file format to be used. If you omit the file type, the default is dynamic (file type 30).

For other than dynamic files, you can specify the modulo (number of groups in the file) and the separation (the group buffer size in 512K blocks). The modulo and separation that you specify with the CREATE TABLE statement allocate the disk space for a hashed file. For complete details about the modulo and separation of tables implemented as hashed files, see the *UniVerse System Description*.

To specify a dynamic file type for NEWTAB.T, enter the CREATE TABLE statement as either of the following:

```
>CREATE TABLE NEWTAB.T (TYPE 30,...
>CREATE TABLE NEWTAB.T (DYNAMIC,...
```

For dynamic files, you can specify parameters as with CREATE.FILE. The UniVerse parameter keywords have the following UniVerse SQL synonyms: SEQNUM, GROUP SIZE, MINIMUM MODULUS, SPLIT LOAD, MERGE LOAD, LARGE RECORD, RECORD SIZE, and MINIMUM SPACE.

You often can omit these entries and accept the defaults. For more information about file types, see the *UniVerse System Description*.

Defining a Column

Column descriptions follow the file type definition, separated by commas. As a minimum, a column description consists of a column name and column data type, as illustrated by these examples from the Circus database:

```
ITEM_TYPE CHAR(1)...
NAME VARCHAR...
ADVANCE DECIMAL (9,2)...
LABOR INTEGER...
```

Optionally, you can specify a format (FMT), a conversion code (CONV), any of the constraints discussed in Chapter 7, "Ensuring Data Integrity," and a default value. An overview of these elements follows.

Column Names

We recommend that column names, like table names, consist only of letters, numbers, and underscores (_), and that they be no longer than 18 characters. For compatibility with existing UniVerse field names, column names can also contain periods. As with table names, you can include special characters in a column name by enclosing all references to the column name in double quotation marks. For example:

```
>CREATE TABLE SALES ("PRIM$KEY" INT ...
```

Delimited Identifiers

Identifiers surrounded by double quotation marks are called *delimited identifiers* or *quoted identifiers*. Thus you can use reserved SQL words (such as DATE and TIME) and identifiers (such as the names of schemas, tables, views, columns, associations, constraints, indexes, table aliases, column aliases, or user names) as delimited identifiers.

The following characters are not allowed in a delimited identifier:

- System delimiters in the range hex FB through FF
  - Text mark
  - Subvalue mark
  - Value mark
  - Field mark
  - Item mark
- ASCII control characters in the range hex 00 through 1F
- SQL NULL (hex 80) if the identifier is only one character long

A delimited column name cannot contain spaces. On Windows platforms, delimited identifiers cannot contain the following:

- “ (double quotation marks)
- % (percent)
- * (asterisk)
- \ (backslash)
- : (colon)
- < (less than)
- > (greater than)

Data Types

The *data type* assigned to a column determines how its data is processed by a UniVerse SQL query, including the arithmetic operations, comparisons, and set functions that are used on it. There are 15 data types, as shown in the following table.

Data Type	Description
BIT [(*n*)]	Stores bit strings.
CHAR[ACTER] [(*n*)]	Stores character strings, which are any combination of numbers, letters, and special characters. *n* is any integer from 1 to 254 and specifies column length.
DATE	Stores dates.
DEC[IMAL] [(*precision* [,*scale*])]	Stores decimal fixed-scale (fixed-point) numbers. *precision* is the number of significant digits; *scale* is the number of digits to the right of the decimal point (same as NUMERIC).
DOUBLE PRECISION	Stores high-precision floating-point numbers.
FLOAT [(*precision*)]	Stores floating-point numbers. *precision* is the number of significant digits.
INT[EGER]	Stores whole decimal numbers.
NCHAR [(*n*)]	Stores national character strings.
NVARCHAR [(*n*)]	Stores variable-length national character strings.
NUMERIC [(*precision* [,*scale*])]	Same as DECIMAL.
REAL	Stores floating-point (real) numbers.
SMALLINT	Stores small whole decimal numbers.
TIME	Stores times.
VARBIT [(*n*)]	Stores variable-length bit strings.
VARCHAR [(*n*)]	Stores variable-length character strings, which are any combination of numbers, letters, and special characters. *n* is any integer from 1 to 65535 and specifies column length.

UniVerse SQL Data Types

Data types are closely associated with two output format specifications, FMT (or FORMAT) and CONV (or CONVERSION). These format specifications are discussed on the next several pages in terms of permanently specifying output formatting and conversion for columns.

Data Types and Empty Strings

In UniVerse the term *empty string* refers to a string of zero length (no data). It is represented in a query by two successive quotation marks (`''`).

In some cases, UniVerse SQL treats empty strings differently from traditional, or non-SQL, UniVerse.

When a WHEN Clause or WHERE Clause in an SQL SELECT query tests for an empty string in a *character-type* column (a column defined as CHAR or VARCHAR), it is treated in the same way as in traditional UniVerse—that is, as a zero-length character string.

However, unlike non-SQL UniVerse, in UniVerse SQL an empty string and a 0 in a *numeric-type* column are identical for all practical purposes. Thus, testing for 0 and testing for an empty string select the same rows, and an empty string in arithmetic expressions is treated as 0. So, to update TAX_LIFE in all rows where TAX_LIFE contains the numeric equivalent of 0, enter either of the following statements:

```
>UPDATE EQUIPMENT.T SET TAX_LIFE = 5 WHERE TAX_LIFE = 0;
>UPDATE EQUIPMENT.T SET TAX_LIFE = 5 WHERE TAX_LIFE = '';
```

Defining a Column's Data Structure and Form

In addition to data type, UniVerse SQL provides several other ways to define a column's data structure according to:

- Whether the column contains a single value or multiple values (SINGLEVALUED or MULTIVALUED)
- What column heading to use for the column on output displays and reports (DISPLAYNAME '*text*')
- What format to use for displaying the column's data in terms of number of characters and justification (FMT)
- What conversion processes to perform, both when putting data *into* the column and when outputting data *from* the column (CONV)

SingleValued or Multivalued

A singlevalued column can contain only a single value per row; a multivalued column can contain multiple values for each row. In the Circus database, multivalued columns are used in several ways.

For example, for each item in the INVENTORY.T table, information about each vendor from whom that item was ordered and the quantity ordered is stored in multivalued columns:

```
VENDOR_CODE INTEGER FORMAT '5R' MULTIVALUED...,
ORDER_QTY INTEGER FORMAT '5R' MULTIVALUED,
```

SINGLEVALUED is the default; you do not need to specify it when defining singlevalued columns.

Column Headers (DISPLAYNAME)

DISPLAYNAME (and its synonym COL.HDG) works the same way as when it is used in a SELECT statement, specifying text to be used as the column heading in place of the column name. However, by supplying these headings when creating the table, they are used automatically with every SELECT statement unless overridden. If you do not include the DISPLAYNAME keyword, the column name is used as the column header.

For example, to use “Type of Shot” as the header for VAC_TYPE in the LIVESTOCK.T table, define the column as:

```
VAC_TYPE CHAR(1) DISPLAYNAME 'Type of Shot' MULTIVALUED...
```

Formatting (FMT)

FMT (and its synonym FORMAT) also behaves in much the same manner as when used in a SELECT statement; it can be used to determine the number of characters and the justification of a column’s data for output. But again, by specifying formatting when you create a table, it applies the formatting automatically to the column for every SELECT statement unless otherwise overridden.

Because UniVerse uses a variable-length data structure, the physical storage of the data may be of any length, but you usually should impose some limit when displaying or printing the data. For example, a 25-character output column is allowed for vendor company name and for each line of address in the VENDORS.T table:

```
COMPANY VARCHAR FMT '25T'
ADR1 VARCHAR FMT '25T'
ADR2 VARCHAR FMT '25T'
ADR3 VARCHAR FMT '25T'
```

You just as easily could have made it 15 characters:

```
COMPANY VARCHAR FMT '15L'
ADR1 VARCHAR FMT '15L'
ADR2 VARCHAR FMT '15L'
ADR3 VARCHAR FMT '15L'
```

Output of data in numeric columns usually is right-justified; data in alphanumeric columns (CHAR) usually is left-justified. Use FORMAT to force either left-justification of numeric data or right-justification of character data in the output:

```
COMPANY VARCHAR FMT '15R'
```

In the case of columns containing multiple-word text, the T code (text justification: left-justify and break on space) is typical. This allows lines to break at the end of words when displayed or printed.

Conversions (CONV)

CONV (and its synonym CONVERSION), like FMT, affects how data is output, but its use has other implications. For example, it is the conversion code that—along with the data type—determines the precise format of a column containing a date or time.

In the database, DATE has a D conversion code ‘D2/’. This causes its contents to be converted to a date whose elements are separated by slashes: *mm/dd/yy*.

```
"DATE" DATE FMT '10L' CONV 'D2/' NOT NULL
```

The D identifies this as a date conversion, the 2 says you want a two-digit year (for example, 94 rather than 1994), and the / denotes the separator to be used. For dashes instead of slashes, and a four-digit year, use:

```
"DATE" DATE FMT '10L' CONV 'D4-' NOT NULL
```

For the European format (*dd.mm.yy*), use:

```
"DATE" DATE FMT '10L' CONV 'D2.E' NOT NULL
```

Many formatting options exist for dates (D), character strings (MC), decimals (MD), numerics (ML and MR), times (MT), and other kinds of data. Here are a few examples:

Option	Description
MD2,$CR	Converts a stored numeric value to a dollar value, with two decimal places (2) preceded by a dollar sign ($); inserts a comma (,) every three digits to the left of the decimal point. Adds the suffix CR to negative values.
D4MD[A,Z]L	Converts a stored date for output as follows: the name of the month (M[A]), retaining any lowercase letters (L), the day, with any leading zero suppressed (D[Z]), and the four digits of the year (4). The brackets are part of the conversion code.
MTHS:	Converts a stored time for output as follows: the hour in 12-hour format (H), minutes, and seconds (S), separated by a colon (:).

Formatting Examples

A conversion code must agree with the column's data type. In this context, think of data types as being divided into five classes:

- Class 1: DEC and NUM with a nonzero scale
- Class 2: DATE
- Class 3: TIME
- Class 4: BIT and VARBIT
- Class 5: Everything else

A conversion code used with class 1 must be MD, ML, or MR and must specify the same scale as the data type definition. For example, the conversion code MD2$ is consistent with the data type DECIMAL(9,2), because the scale is 2 in both cases. However, a conversion code of MD4$ is incorrect and produces unpredictable results.

The D conversion code must be used with class 2 (DATE). The MT conversion code must be used with class 3 (TIME).

The BB (binary) and BX (hexadecimal) conversion codes can be used with class 4 (BIT and VARBIT).

Never use any of the following conversion codes with a data type of class 5:

- MD, ML, or MR with a nonzero scale
- D
- MT

Although the system does not enforce these rules, violating them usually produces strange results.

Data Categories

The combination of a column's data type and conversion code designates a column's *data category*. For example, if a column is described with a DATE data type and a valid date conversion code, it falls into the Date category. This means that when a value is stored into that column, it is converted into an internal date format (the number of days from December 31, 1967). When the column is printed or displayed, its contents are converted to an appropriate date display format (for example, 3/12/95 or March 12, 1995) as requested by the conversion code.

UniVerse SQL recognizes seven data categories:

- Integer
- Scaled number
- Approximate number
- Date
- Time
- Character string
- Bit and hex string

Data categories are important because they determine how a column is used in practice and how the data in a column is treated in terms of input and output formatting and converting. They also determine what kinds of operation you can perform on them and the results.

For example, you can:

- Add, subtract, multiply, or divide any pair of number (integer, scaled number, approximate number) columns. The result will be an approximate number (unless both columns are integer, in which case the result will also be an integer).
- Add integers to (or subtract integers from) dates and times.

- Add and subtract times, and subtract a date from a date (but you cannot add a date to a date).
- Compare integers, scaled numbers, and approximate numbers to each other, dates to dates, times to times, and character strings to character strings.

Integer

Integers are positive or negative whole numbers, such as 0, 5, –735692, and +03. Integers represent counts, quantities, and the like. You can store integers in any column defined with a data type of INT or SMALLINT. The default FMT code is 10R (10 positions, right-justified), and the default CONV code is MD0 (no decimal positions). Examples of integer columns in the Circus database include:

```
ACRES INTEGER FORMAT '5R'
USE_LIFE INTEGER FORMAT '5R'
OPERATOR INTEGER FORMAT '5R' MULTIVALUED REFERENCES PERSONNEL.T
```

Scaled Number

Scaled numbers are positive or negative numbers with fixed-length fractional parts, such as 2.00, 19958.255, or –0.7556. They are used to store values like money amounts, percentages, and temperature and pressure readings. You can store scaled number values in any column defined with a data type of DEC(*p*,*s*) or NUMERIC(*p*,*s*,). Some examples from the database include:

```
GOV_RATE DECIMAL(3,3) FORMAT '7R' MULTIVALUED
COST DECIMAL(9,2) FORMAT '12R'
RIDE_REVENUE DECIMAL(9,2) FORMAT '12R' MULTIVALUED
```

Internally, scaled numbers are stored without the decimal point. The conversion code is MD*ss*, where *ss* is the scale.

Approximate Number

Approximate numbers are real numbers that can include fractional parts of unknown length and 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…). Approximate number values can be stored in any column defined as REAL, FLOAT(*p*), or DOUBLE PRECISION.

Date

Dates are stored as the number of days since December 31, 1967, and output in date form, such as 2/6/95 or February 6, 1995, according to a conversion code. You can define a column as a date by specifying the DATE data type. An example from the Circus database is:

```
"DATE" DATE FORMAT '10L' CONV 'D2/' NOT NULL
```

For an international date format, use a conversion code like ‘D4.E’, which displays February 6, 1995, as 6.2.1995.

You can also omit the CONV specification. If you do, a ‘D’ conversion is used as the default.

Time

Times are stored as a number of seconds, representing either the time of day or a time interval, and are output in time format, such as 2:30 p.m., 14:30:00, or 14h30, depending on the conversion code. You can define a column as a time by specifying the TIME data type. An example from the Circus database is:

```
"TIME" TIME FORMAT '10L' CONV 'MTH'
```

You can also omit the CONV specification. If you do, a ‘MTS’ conversion is used as the default.

Character String

Character strings are any mixture of characters, numbers, and special characters. They hold text, such as descriptions, names, addresses, or alphanumeric codes, and can be stored in any column described as CHAR(*n*) or VARCHAR. Examples include:

```
COMPANY VARCHAR FORMAT '25T'
ADR1 VARCHAR FORMAT '25T'
PHONE VARCHAR FORMAT '12L'
DEPRECIATION CHAR(1) FORMAT '1L'
```

Bit and Hex String

Bit strings are any arbitrary sequence of bits. They can be input or displayed in binary format as 0's and 1's enclosed in single quotation marks and preceded by the base specifier B (for example, `B'01111110'`). They can also be input or displayed in hexadecimal format as *hexits* (hexadecimal digits: 0 – 9, A – F, a – f) enclosed in single quotation marks and preceded by the base specifier X (for example, `X'1F3A2'`). Bit strings can be stored in any column described as BIT or VARBIT.

Column and Table Constraints

Constraints are mentioned here because they are an important part of column definition. Column constraints are used to ensure data integrity and include UNIQUE, ROWUNIQUE, NOT NULL, NOT EMPTY, CHECK, PRIMARY KEY, FOREIGN KEY, and REFERENCES.

One constraint that deserves special mention is the PRIMARY KEY constraint. A primary key is the values in one or more columns that together uniquely identify that row in a table. Except for ENGAGEMENTS.T, every table in the Circus database has a single column defined as PRIMARY KEY:

```
ACT_NO INTEGER FORMAT '5R' PRIMARY KEY
BADGE_NO INTEGER FORMAT '5R' PRIMARY KEY
ANIMAL_ID INTEGER FORMAT '5L' PRIMARY KEY
```

In UniVerse SQL you can define a multipart primary key, designating two or more columns as the primary key. You use a PRIMARY KEY table constraint to specify these columns.

The ENGAGEMENTS.T table is an example of a table with a multicolumn primary key:

```
LOCATION_CODE CHAR(7) FORMAT '7L' NOT NULL
"DATE" DATE FORMAT '10L' CONV 'D2/'
.
.
.
CONSTRAINT ENGAGEMENT_KEY PRIMARY KEY (LOCATION_CODE, "DATE")
```

Default Values

Default values are another column definition option. Use the DEFAULT clause to insert a value into a column whenever a value is not provided. Specify a literal value, NULL (which is the "default" default), or USER (which sets the effective user name of the current user as the default value).

To place the value "None currently" in the CONTACT column every time a row was inserted in VENDORS.T and no contact name was supplied, enter:

```
CONTACT VARCHAR FORMAT '10T' DEFAULT 'None currently'
```

Column Synonyms

Column synonyms allow you to define more than one format or conversion specification for a column. For example, to display or print the DATE in the ENGAGEMENTS.T table in two different date formats, American and European:

```
"DATE" DATE FORMAT '10L' CONV 'D2/' NOT NULL,
DATE1 SYNONYM FOR "DATE" FORMAT '10L' CONV 'D4.E'...
```

Then, if you ask for DATE (and the value was 12/31/94), you see:

```
DATE
12/31/94
```

If you ask for DATE1, you see:

```
DATE1
31.12.1994
```

Associations and Multivalued Columns

When you have a group of multivalued columns in a table, it is likely that two or more will be associated with one another. That is, the first value of one column is associated with the first value of another column; the second value of one with the second value of the other, and so on. Each row of associated values is called an *association row* to distinguish it from a row of the base table.

An association's definition controls the order in which new association rows are positioned. For example, any new association row can be positioned before or after any existing rows, or it can be assigned a stable position that will not change even when preceding association rows are deleted or repositioned.

The Circus database has several examples of associations. In the ENGAGEMENTS.T table, there are three associations: one for the gates/revenues/tickets, one for the concessions/revenues/tickets, and one for the rides/revenues/tickets. These associations and their attributes are defined as part of the ASSOCIATION clause of the CREATE TABLE statement:

```
ASSOCIATION GATE_ASSOC (GATE_NUMBER KEY, GATE_REVENUE,
   GATE_TICKETS)
ASSOCIATION CONCS_ASSOC (CONC_ID KEY, CONC_REVENUE,
   CONC_TICKETS)
ASSOCIATION RIDES_ASSOC (RIDE_ID KEY, RIDE_REVENUE,
   RIDE_TICKETS)
```

ACT_NO, a single multivalued column, is not associated with any other multivalued columns.

Likewise, the PERSONNEL.T table has four associations: one for the columns containing dependent data, one for the columns containing the equipment experience data, one for the columns containing the acts experience data, and one for the columns containing the rides experience data:

```
ASSOCIATION DEP_ASSOC (DEP_NAME KEY, DEP_DOB, DEP_RELATION)
ASSOCIATION EQUIP_ASSOC (EQUIP_CODE KEY, EQUIP_PAY)
ASSOCIATION ACTS_ASSOC (ACT_NO KEY, ACT_PAY)
ASSOCIATION RIDES_ASSOC (RIDE_ID KEY, RIDE_PAY)
```

Association Keys

You can specify one or more of the columns of an association as the *association key*. You can think of association keys as being to associations what primary keys are to tables. Therefore they are subject to the same constraints. This means that a column used as an association key must be declared NOT NULL.

Related UniVerse Associations

In addition to the associations of multivalued columns you can define for a table, UniVerse has two other kinds of association. And an unassociated multivalued column behaves in certain ways as if it were an association comprising one multivalued column.

Thus, UniVerse SQL can manipulate four kinds of multivalued data structures:

- Associations in tables
- Associations in UniVerse files that are not tables

- Unassociated multivalued columns in tables and UniVerse files
- Pick-controlling and dependent fields

For UniVerse SQL to access UniVerse file associations and unassociated multivalued columns and Pick associations, you need to know how to use the following:

- The @ASSOC_ROW keyword
- The @ASSOC_KEY.*mvname* X-descriptor

@ASSOC_ROW

When you select from a dynamically normalized association that has no key, the system generates a virtual column called @ASSOC_ROW containing unique values. By combining the values of @ASSOC_ROW with a base table's primary keys, you get a set of jointly unique association row keys that function as the primary keys of the dynamically normalized association.

@ASSOC_KEY.mvname

For a standard table you define an association of multivalued columns using the ASSOC clause in the CREATE TABLE or ALTER TABLE statement. The ASSOC clause defines the name and composition of the association as well as its attributes (association keys and positioning of association rows).

For UniVerse SQL to make full use of associations in UniVerse files, you need to add an X-descriptor called @ASSOC_KEY.*mvname* to the file dictionary. *mvname* is one of the following:

- For UniVerse files, the name of the phrase that defines the association
- For an unassociated multivalued field in a UniVerse file, the name of the field
- For a Pick association of controlling and dependent fields, @DC*n*, where *n* is the location of the controlling field

The content of this X-descriptor is as follows:

```
@ASSOC_KEY.mvname
0001 X
0002{ STABLE | UNSTABLE | KEY field [field]… }
```

Field 2 defines one of two different association attributes:

- A definition of the association's key—one or more columns that uniquely identify association rows in each base table row
- Whether the position of association rows is stable or unstable (if no association key can be defined)

One or more KEY field specifications define a subset of the associated fields as association keys. This specification is equivalent to the KEY clause of the ASSOC clause in the CREATE TABLE and ALTER TABLE statements. If the association has no keys, you can define the positioning of the association rows in the association as STABLE. This ensures the following:

- Newly inserted association rows do not displace existing association rows. If they are inserted in positions more than one row higher than the existing rows, empty rows are inserted.
- Deleted association rows are replaced with empty association rows, which prevents subsequent association rows from being renumbered.
- Updated association rows can be repositioned only to empty row positions, by using an UPDATE statement to change the value of @ASSOC_ROW. If they are repositioned more than one row higher than the existing rows, empty rows are inserted.

UNSTABLE is the default if you do not use an @ASSOC_KEY.*mvname* entry in the dictionary. This means that higher-numbered association rows are renumbered when lower-numbered rows are deleted, inserted, or repositioned by the SET clause of an UPDATE statement. Empty association rows are not inserted when new rows are inserted or repositioned at the end of existing rows, and deleted association rows are not replaced with empty association rows.

Pick Associations

Pick associations, like other associations, are defined in the file dictionary, but they are defined differently from associations in standard tables and UniVerse files. Each Pick association has one controlling field and one or more dependent fields. The dictionary definition of a controlling field has the following code in field 4:

```
C;field# [;field#]…
```

field# is the location of a dependent field.

The dictionary definition of a dependent field has the following code in field 4:

```
D;control.field#
```

control.field# is the location of the controlling field.

To use the @ASSOC_KEY.*mvname* X-descriptor with Pick associations, use @DC*n* as *mvname* (where *n* is the field number of the controlling field).

Modifying a Table

Use the ALTER TABLE statement to:

- Add columns, column synonyms, table constraints, associations
- Remove table constraints, associations, and default values
- Change a column's default value
- Enable or disable a table's triggers

Consequently, the statement includes an ADD clause, a DROP clause, an ALTER clause, and a TRIGGER clause. Note that you cannot use ALTER TABLE to delete *existing* columns, add *column* constraints, or change the definition of existing columns in ways other than just described. If a column is no longer required, use an UPDATE statement to clear that column's data.

Adding Columns, Table Constraints, and Associations

Sometimes you need to add new columns, synonyms, constraints, or associations to a table, either because you forgot to do so when you created the table, or because changes to the application make such additions necessary. Use the ADD clause of the ALTER TABLE statement, with the syntax of the actual definition itself identical to the syntax you would use in a CREATE TABLE statement.

Adding a New Column

Adding a column to an existing table is a simple matter of issuing an ALTER TABLE statement with an ADD Clause: Column that includes a column definition identical to the column definition you would use in a CREATE TABLE statement. For example, to add a new column called AGENT to the ACTS.T table, use the following statement:

```
>ALTER TABLE ACTS.T ADD COLUMN AGENT VARCHAR FORMAT '10L';
Adding Column AGENT
```

Adding a Column Synonym

Adding a column synonym is equally straightforward. Taking the example of using a column synonym to define an alternative date format for the DATE column in the ENGAGEMENTS.T table, you could add the same alternate format to VAC_DATE in the LIVESTOCK.T table:

```
>ALTER TABLE LIVESTOCK.T
SQL+ADD DATE1 SYNONYM FOR VAC_DATE CONV 'D2.E';
Adding Synonym DATE1
```

Adding a Table Constraint

To add a table constraint (CHECK, UNIQUE, or FOREIGN KEY) to a table, use an ALTER TABLE statement with the ADD CONSTRAINT syntax. For example, to add a table constraint, called LIFECHK, that ensures that a tax life (TAX_LIFE) value is 5, 10, or 25, use the following statement:

```
>ALTER TABLE EQUIPMENT.T
SQL+ADD CONSTRAINT LIFECHK CHECK (TAX_LIFE IN (5, 10, 25));
Adding Constraint LIFECHK
```

Remember that if you try to add a constraint to an existing table, but values already in the table violate that constraint, ALTER TABLE is rejected.

Adding a New Association

To define a new association of multivalued columns, use the ADD ASSOCIATION option of ALTER TABLE. Associations group related multivalued columns together. In each row, the first value in one multivalued column of an association has a one-to-one relationship to the first values in all the other associated columns, the second value has a one-to-one relationship to the second values in the other associated columns, and so on. An association, therefore, is an array of columns containing related multivalues and, in effect, can be thought of as a *nested table* or a *table within a table*.

For example, if the four vaccination-related multivalued columns in LIVE-STOCK.T were not already defined as an association, you could associate them by entering:

```
>ALTER TABLE LIVESTOCK.T
SQL+ADD ASSOC VAC_ASSOC (VAC_TYPE KEY, VAC_DATE, VAC_NEXT,
SQL+VAC_CERT);
Adding Association VAC_ASSOC
```

The ADD ASSOC clause of ALTER TABLE also can specify the order in which new rows are to be added to the association. INSERT FIRST adds new rows to the beginning of existing association rows, INSERT LAST (the default) adds new rows to the end of existing rows, and INSERT IN *columnname* BY *sequence* orders new rows according to the value in the named column of the association. For example, to add new rows to VAC_ASSOC before existing rows, enter the previous definition as:

```
>ALTER TABLE LIVESTOCK.T
SQL+ADD ASSOC VAC_ASSOC INSERT FIRST
SQL+(VAC_TYPE KEY, VAC_DATE, VAC_NEXT, VAC_CERT);
Adding Association VAC_ASSOC
```

Many associations are generated specifying a key (but a key is not required). An association key can be thought of as the primary key of the "table within a table" that the association represents.

Associations usually comprise two or more columns. For example, the CREATE TABLE statement for the ENGAGEMENTS.T table includes association definitions for gates, concessions, and rides. You can also define associations that have only one multivalued column.

Single-column associations are useful when applying dynamic normalization to a file. Dynamic normalization explodes multivalued columns (associated or unassociated) so that they appear as singlevalued. In other words, dynamic normalization allows you to process a nonfirst-normal-form (NF^2) table as if it were a first-normal-form (1NF) table.

Use dynamic normalization on ACT_NO to do such things as insert a new act for an upcoming engagement. For example, if you look at the acts scheduled for East Atlanta for February 15, 1996, you see that only one act is booked:

```
>SELECT ACT_NO FROM ENGAGEMENTS.T
SQL+WHERE LOCATION_CODE = 'EATL001' AND "DATE" = '02/15/96';
ACT_NO

     4

1 records listed.
```

To add act 2 to the roster of acts, enter:

```
>INSERT INTO ENGAGEMENTS.T_ACTS_ASSOC
SQL+VALUES ('EATL001', '02/15/96', 2);
UniVerse/SQL: 1 record inserted.

>SELECT ACT_NO FROM ENGAGEMENTS.T
SQL+WHERE LOCATION_CODE = 'EATL001' AND "DATE" = '02/15/96';
ACT_NO

     4
     2

1 records listed.
```

Removing Constraints, Associations, and Default Values

Just as you can add table and column constraints, associations, and default values to an existing table, you can remove them using the DROP clause of the ALTER TABLE statement. To remove the previously added LIFECHK constraint, EP_PERFS association, and the default value for QOH, the following statements suffice:

```
>ALTER TABLE EQUIPMENT.T DROP CONSTRAINT LIFECHK;
Dropping Constraint LIFECHK
>ALTER TABLE ENGAGEMENTS.T DROP ASSOC EP_PERFS;
Dropping Association EP_PERFS
>ALTER TABLE INVENTORY.T ALTER QOH DROP DEFAULT;
Dropping DEFAULT on column QOH
```

When using the DROP Clause: Integrity Constraint to remove a UNIQUE constraint, there are two options: RESTRICT and CASCADE. RESTRICT, the default, prevents the removal of a UNIQUE constraint if the column is referenced by a foreign key column. CASCADE removes the UNIQUE constraint and removes any referential constraints from any foreign key columns that are dependent on it.

Changing a Column's Default Value

Besides column constraints, the only other element you can change in a column definition is its default value. The default value may be "no default value," in which case the implied default value is NULL. You can change a column's default value through the SET DEFAULT clause of the ALTER TABLE statement.

To add a default value of 12/31/99 to the DATE column, perhaps to indicate a tentatively booked engagement for which a date has not yet been determined, enter:

```
>ALTER TABLE ENGAGEMENTS.T ALTER "DATE" SET DEFAULT '12/31/99';
Setting DEFAULT on column DATE
```

This also ensures that DATE is never NULL (the implicit default value), which would cause the rejection of an entry because the column is part of the PRIMARY KEY. Primary key columns cannot contain null values.

Dropping a Table

Removing a table from your database can be as simple as issuing a DROP TABLE statement. To be consistent with the previous examples, assume that the table to be deleted is RIDES.T:

```
>DROP TABLE RIDES.T;
Dropping Table RIDES.T
```

DROP TABLE removes the table and deletes the UniVerse data file, the file dictionary, and any indexes. It also revokes all privileges on the table.

Dropping a Table with a Dependent View

If the table has views dependent on it, you must add the keyword CASCADE to drop the views as well. Otherwise, an error results because you cannot remove a table without removing the views associated with the table at the same time. Views are fully discussed in the *UniVerse SQL User Guide*.

Dropping a Referenced Table

Sometimes a table you want to drop is a referenced table (a table containing one or more columns that are referenced by other tables, as is the case when some table contains a foreign key that references a column in the table being dropped). In such cases, resolve these references before dropping the table.

Take an example in which you have two tables, TABLE1 and TABLE2, with TABLE2's ID referenced by a foreign key in TABLE1:

```
>CREATE TABLE TABLE1 (TID_1 INT PRIMARY KEY,
SQL+FORKEY_2 INT, DESC_1 CHAR (10));
>CREATE TABLE TABLE2 (TID_2 INT PRIMARY KEY,
SQL+DESC_2 CHAR (10));
>ALTER TABLE TABLE1 ADD CONSTRAINT F1 FOREIGN KEY (FORKEY_2)
SQL+REFERENCES TABLE2 (TID_2);
```

If you later attempt to drop TABLE2, the following error message appears:

```
Dropping Table TABLE2
UniVerse SQL: TABLE2 is a referenced table. DROP the referencing
tables or constraints first.
```

One solution is to drop TABLE1 first:

```
>DROP TABLE TABLE1;
Dropping Table TABLE1
>DROP TABLE TABLE2;
Dropping Table TABLE2
```

However, if you want to keep TABLE1, the solution is to drop the foreign key table constraint (F1) from TABLE1 and then drop TABLE2:

```
>ALTER TABLE TABLE1 DROP CONSTRAINT F1;
Dropping table constraint F1
>DROP TABLE TABLE2;
Dropping Table TABLE2
```

Remember that a constraint must be named in order to drop it, so it is a good idea to assign names to all constraints when you define them.

Indexes

An index is a sorted list of the values in a column. Indexes work like B-tree files in that they provide optimum processing speed when columns other than the primary key are used as the key column in WHERE, WHEN, and ORDER BY clauses. Take as an example the following statement:

```
>SELECT COL2 FROM TABLE
SQL+WHERE COL2 < 100;
```

Without indexing, every row in the table named TABLE is examined for rows with the value of column 2 less than 100. If column 2 has an index, however, each item in the index is examined in increasing order until an item is found that is greater than or equal to 100. All other items are assumed not to match and the search is ended.

If all the rows in the table have a value less than 100 in column 2, the search time is the same as without an index, but if only a few items have a value less than 100, the search time is greatly reduced.

Once you have created an index, it contains a sorted list of the values in the indexed column—the keys of the index—along with the IDs of the records in the indexed table. Each unique value in the indexed column is stored as the record ID of a record in the index. Each index record comprises one or more fields containing the keys to the rows in the indexed table.

For example, the indexed table can contain the following columns:

```
PRIMARY KEYS...    LNAME......

111-888-3333       SMITH
222-555-6666       JONES
888-444-9999       SMITH
```

An index on the column LNAME is organized as follows:

```
PRIMARY KEYS...    COL1.....................

JONES              222-555-6666
SMITH              111-888-3333F888-444-9999
```

The F represents a field mark.

When an index is first created, it contains no values. As new values are added to the indexed table, corresponding values are also added to the index. If the indexed table contains data at the time the index is created, the data is indexed immediately.

Creating an Index

Use the CREATE INDEX statement to create an index on one or more columns of a table. The CREATE INDEX statement creates one index on one column or a set of columns. The syntax of the CREATE INDEX statement is as follows:

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

Dropping an Index

Use the DROP INDEX statement to delete an index from a table. The syntax of the DROP INDEX statement is as follows:

```
DROP INDEX tablename.indexname;
```

Using Triggers on a Table

You can augment and regulate the modification of data in tables by creating a trigger for the table. A trigger specifies actions to perform before or after the execution of certain events that change the database. You can define up to six triggers for a table. The names of all triggers and their corresponding UniVerse BASIC programs are stored in the table's SICA. For more information about using Triggers, see the *UniVerse SQL Reference*.

Adding a Trigger

You write trigger programs in UniVerse BASIC. Use the CREATE TRIGGER statement to create a trigger for a table, calling the trigger program that you want to execute. You must be the table's owner or have ALTER Privilege on the table, or you must be a DBA to create a trigger.

You can set a trigger to fire (execute) *before* an INSERT, UPDATE, or DELETE event can change data. A BEFORE trigger can examine the new data and determine whether to allow the INSERT, UPDATE, or DELETE event to proceed; if the trigger rejects a data change, UniVerse rolls back the entire transaction. The trigger is evaluated for each row to be modified.

You can also set triggers to fire *after* an INSERT, UPDATE, or DELETE event, for example, to change related rows, audit database activity, and print or send messages.

Enabling and Disabling Triggers

In order to fire a table's trigger, it must be enabled. When you create a trigger, it is enabled by default. You can use the ALTER TABLE statement to disable and reenable a table's triggers. To disable a trigger associated with a table, use the DISABLE TRIGGER clause:

```
>ALTER TABLE EMPLOYEES DISABLE TRIGGER AUDIT_EMPLOYEES;
Disabling trigger "AUDIT_EMPLOYEES"
```

To reenable the same trigger, use the ENABLE TRIGGER clause:

```
>ALTER TABLE EMPLOYEES ENABLE TRIGGER AUDIT_EMPLOYEES;
Enabling trigger "AUDIT_EMPLOYEES"
```

You can use ALL instead of the trigger name to enable or disable all of a table's triggers:

```
>ALTER TABLE EMPLOYEES DISABLE TRIGGER ALL;
Disabling all triggers
```

Dropping a 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.

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

Listing Information About a Table

A table comprises:

- A data file
- A table dictionary
- A SICA (security and integrity constraints area)

You can examine all of these as sources of information about the table.

Examining a Table's Data File

To see the contents of a table's data file, use the UniVerse SQL SELECT statement. You can ask to see the data in the following ways:

- All of the rows and all of the columns:

  ```
  >SELECT * FROM ENGAGEMENTS.T;
  ```

- All of the columns and selected rows:

  ```
  >SELECT * FROM ENGAGEMENTS.T
  SQL+WHERE "DATE" < '12/31/95';
  ```

- Selected columns and all of the rows:

  ```
  >SELECT LOCATION_CODE, "DATE", "TIME"
  SQL+FROM ENGAGEMENTS.T
  SQL+ORDER BY LOCATION_CODE, "DATE";
  ```

- Selected columns and selected rows:

  ```
  >SELECT LOCATION_CODE, "DATE", "TIME"
  SQL+FROM ENGAGEMENTS.T
  SQL+WHERE "DATE" < '12/31/95'
  SQL+ORDER BY LOCATION_CODE, "DATE";
  ```

Alternatively, use the RetrieVe LIST command, which features much of the functionality of SELECT.

Examining a Table's Dictionary

To see the contents of a table's dictionary, use the SELECT statement with the DICT keyword:

```
>SELECT * FROM DICT ENGAGEMENTS.T ORDER BY CODE, LOC;

                Type &
Field.......    Field. Field.......Conversion..Column......Output Depth &
Name........    Number Definition..Code........Heading.....Format Assoc..

@ID             D    0                          ENGAGEMENTS 7L     S
TIME            D    1             MTH                      10L    S
ADVANCE         D    2             MD22                     12R    S
GATE_NUMBER     D    3             MD0                      5R     M GATES_ASSOC
GATE_REVENUE    D    4             MD22                     12R    M GATES_ASSOC
GATE_TICKETS    D    5             MD0                      5R     M GATES_ASSOC
ACT_NO          D    6             MD0                      5R     M
RIDE_ID         D    7             MD0                      3R     M RIDES_ASSOC
RIDE_REVENUE    D    8             MD22                     12R    M RIDES_ASSOC
RIDE_TICKETS    D    9             MD0                      5R     M RIDES_ASSOC
CONC_ID         D   10             MD0                      5R     M CONCS_ASSOC
Press any key to continue...
```

For each column in the table, you see the column's name, type, column number, definition, conversion code, optional column headings, output format, and depth and association. You also could have used:

```
>LIST DICT ENGAGEMENTS.T
```

If you want to print the dictionary, use the following statement to send a report listing the contents of a dictionary to the printer:

```
>SELECT * FROM DICT ENGAGEMENTS.T LPTR;
```

Or, again, you could have entered the PRINT.DICT command:

```
>PRINT.DICT ENGAGEMENTS.T
```

Examining a Table's SICA

A special list command, LIST.SICA, is available for viewing the information in the SICA, which is stored in the file header and is exclusive to tables. The SICA contains much of the same information found in the file dictionary, but it also includes information relevant only to tables (in particular, constraints and permissions).

To examine the SICA for the EQUIPMENT.T table, enter:

```
>LIST.SICA EQUIPMENT.T

LIST.SICA EQUIPMENT.T 09:52:31AM  02 May 1995   Page    1
========================================
Sica Region for Table "EQUIPMENT.T"

  Schema:          CIRCUS
  Revision:        2
  Checksum is:     27767
    Should Be:     27767
  Size:            920
  Creator:         719
  Total Col Count: 10
    Key Columns:   1
    Data Columns:  9
  Check Count:     0
  Permission Count:0
  References Count:1
  Referenced Count:5
  History Count:   0

  Data for Column "EQUIP_CODE"

    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:        5R
Press any key to continue...
```

Examining a Table's SQL Catalog Information

The SQL catalog, comprising six tables in the CATALOG schema, contains information about all schemas, tables, columns, associations, views, and users on the system. For information about What Is the SQL Catalog?, see the *UniVerse SQL Reference*.

You can use the SELECT command to read from the SQL catalog tables, but you cannot use the INSERT, UPDATE, or DELETE commands to modify them. For example, to see the UV_TABLES information for the table ENGAGEMENTS.T in schema DEMO_glenn, enter:

```
>SELECT * FROM UV_TABLES
SQL+WHERE TABLE_SCHEMA = 'DEMO_glenn'
SQL+AND TABLE_NAME = 'ENGAGEMENTS.T';
Schema.......DEMO_glenn
Table........ENGAGEMENTS.T
Owner........210
Table Type...BASE TABLE
Base Table...
Columns...... LOCATION_CODE
              . DATE
              . TIME
              . ADVANCE
              . GATE NUMBER
              . GATE_REVENUE
              . GATE_TICKETS
              . ACT_NO
              . RIDE_ID
              . RIDE_REVENUE
              . RIDE_TICKETS
              . CONC_ID
              . CONC_REVENUE
              . CONC_TICKETS
              . LABOR
              . PAY
Views........
Path........./rd2/glenn/Sql/ENGAGEMENTS.T
Dict Path..../rd2/glenn/Sql/D_ENGAGEMENTS.T
Associations.CONC_ASSOC
              . GATES_ASSOC
              . RIDES_ASSOC
Remarks......

1 records listed.
```

Modifying Table Dictionaries

***Warning**: This section discusses tables only, not regular UniVerse files.With a few exceptions, you should not modify a table's dictionary, since this may result in inconsistencies between the contents of the dictionary and the contents of the table's SICA and the SQL catalog.*

Among the cases in which you can safely modify a table's dictionary are:

- To add I-descriptors (virtual columns) to the table definition
- To change the set of columns to be displayed by a SELECT * FROM *table* statement
- To declare that an unassociated multivalued column has a stable key

The following sections discuss these cases in detail, with examples. For these examples, assume that you have defined a table called SALES96:

```
>CREATE TABLE SALES96
SQL+(REGION VARCHAR PRIMARY KEY,
SQL+QBONUS DEC(9,2) CONV 'MD2$' MULTIVALUED,
SQL+SALE_CODE INT NOT NULL MULTIVALUED,
SQL+SALESREP VARCHAR MULTIVALUED,
SQL+AMOUNT DEC (7,2) CONV 'MD2$' MULTIVALUED,
SQL+ASSOC DETAIL (SALE_CODE KEY,SALESREP,AMOUNT));
```

This table contains one row for each sales region. In each row there is a four-valued column called QBONUS showing the region's bonus amount for each quarter of the year, and there is a multivalued association called DETAIL containing detailed information about each 1996 sale in the region. Suppose the initial contents of table SALES96 are:

```
>SELECT * FROM SALES96;

REGION....    QBONUS....    SALE_CODE.    SALESREP..    AMOUNT..

EAST            $2200.00          9611    SMITH         $2500.00
                 $500.00          9624    GARCIA        $1250.87
                $4000.00          9617    SMITH          $985.00
                 $975.00
```

Adding I-Descriptors to the Table Definition

I-descriptors define virtual (calculated) columns in a table. They have names, formatting characteristics, and data types just like real (stored) columns. I-type expressions are written in a subset of the UniVerse BASIC language and are stored in the table's dictionary. When an I-descriptor is used as a column specification in a SELECT clause, its value is calculated by executing the I-descriptor's compiled UniVerse BASIC object code.

Consider the SALES96 table again. Suppose you want to define a virtual column for the commission paid for each sale. Suppose the commission rate for each sales representative is stored in a table called COMM96:

```
>SELECT * FROM COMM96;

SALESREP..    COMM......

GARCIA              0.25
SMITH               0.15

2 records listed.
```

You can use the UniVerse BASIC TRANS function to determine each commission rate and multiply it by the amount of the sale. This is done by inserting an I-descriptor (called COMMISSION) into the dictionary of SALES96:

```
>INSERT INTO DICT SALES96
SQL+(@ID,CODE,EXP,SM,ASSOC,DATATYPE,FORMAT,CONV)
SQL+VALUES ('COMMISSION','I',
SQL+'TRANS(COMM96,SALESREP,1,"X");@ *  AMOUNT',
SQL+'M','DETAIL','DEC,7,2','10R','MD2$');
```

After adding an I-descriptor to a dictionary, you should compile the I-descriptor immediately to be sure the UniVerse BASIC code is correct:

```
>CD SALES96
Compiling "COMMISSION".
TRANS ( COMM96 , SALESREP , 1 , X ) ; @ * AMOUNT
```

If you do not compile the I-descriptor, it is compiled the first time you use it.

Since COMMISSION is a multivalued column whose values are calculated for each individual sale, COMMISSION is made part of the DETAIL association. Since the DETAIL association is also represented by a phrase in the dictionary called DETAIL, that phrase should be augmented to contain the name of the new virtual column COMMISSION. Use the following UPDATE statement to do this (note that the literal constant being concatenated at the end of the DETAIL phrase starts with a blank character to separate the word COMMISSION from the existing phrase):

```
>UPDATE DICT SALES96 SET EXP = EXP ||' COMMISSION' WHERE @ID =
'DETAIL';
```

Now each sales commission can be displayed as a column in the SALES96 table:

```
>SELECT REGION,SALE_CODE,SALESREP,AMOUNT,COMMISSION FROM SALES96;
    REGION....    SALE_CODE.    SALESREP..    AMOUNT..
COMMISSION

    EAST                9611    SMITH         $2500.00
$375.00
                        9624    GARCIA        $1250.87
$312.72
                        9617    SMITH          $985.00
$147.75
```

Changing the Default Set of Displayed Columns

Now observe what is displayed by SELECT * FROM SALES96:

```
>SELECT * FROM SALES96;
REGION....    QBONUS....    SALE_CODE.    SALESREP..    AMOUNT..

EAST           $2200.00          9611    SMITH         $2500.00
                $500.00          9624    GARCIA        $1250.87
               $4000.00          9617    SMITH          $985.00
                $975.00
```

After adding the COMMISSION I-descriptor, you may want to display the commission when you execute SELECT * FROM SALES96. Perhaps you also want to hide the region's quarterly bonus by removing it from the set of column values that are displayed by SELECT * FROM SALES96. And you may want to display the SALESNAME synonym instead of the SALESREP column. You can do all this by adding a special phrase called @SELECT to the table dictionary. @SELECT defines all of the columns to be displayed by SELECT * FROM SALES96:

```
>INSERT INTO DICT SALES96 (@ID,CODE,EXP)
SQL+VALUES ('@SELECT','PH',
SQL+'REGION SALE_CODE SALESNAME AMOUNT COMMISSION');

>SELECT * FROM SALES96;
REGION....    SALE_CODE.    SALESNAME.    AMOUNT..    COMMISSION

EAST               9611    Smith         $2500.00       $375.00
                   9624    Garcia        $1250.87       $312.72
                   9617    Smith          $985.00       $147.75
```

Defining a Stable Unassociated Multivalued Column

Sometimes a multivalued column contains a fixed number of values, where the position of each value is meaningful. An example is the QBONUS column in the SALES96 table. QBONUS is not just an unsorted set of numbers, it contains each region's bonus amounts for the four quarters of the year in chronological order. But suppose the vice president of Sales chooses to delete the East region's bonus for the third quarter after some customer cancels an anticipated sale? You need a way to force each quarterly bonus amount into the correct sequential position in this four-valued column. You can do this by writing a UniVerse BASIC program to update the table, but you can also do it with SQL statements by properly defining the ordering property of the unassociated column QBONUS.

First consider what happens if you delete the third value in QBONUS before modifying the dictionary:

```
>DELETE FROM SALES96_QBONUS WHERE QBONUS = 4000;
UniVerse/SQL: 1 record deleted.

>SELECT REGION,QBONUS FROM SALES96;

REGION....    QBONUS
EAST        $2200.00
             $500.00
             $975.00
```

What had been the fourth-quarter bonus now appears to be the third-quarter bonus, which is not what you intended. Now restore the previous data in QBONUS:

```
>UPDATE SALES96 SET QBONUS = <2200,500,4000,975> WHERE REGION =
'EAST';
UniVerse/SQL: 1 record updated.
>SELECT REGION,QBONUS FROM SALES96;
REGION....    QBONUS....

EAST            $2200.00
                 $500.00
                $4000.00
                $975.00
```

To get the behavior you want, add an X-descriptor to SALES96's dictionary which declares QBONUS to be STABLE (which is equivalent to the INSERT PRESERVING property of an association):

```
>INSERT INTO DICT SALES96 (@ID,CODE,EXP)
SQL+VALUES ('@ASSOC_KEY.QBONUS','X','STABLE');
```

Now you can delete the third-quarter bonus without affecting the fourth quarter:

```
>DELETE FROM SALES96_QBONUS WHERE QBONUS = 4000;
UniVerse/SQL: 1 record deleted.
>SELECT REGION,QBONUS FROM SALES96;
REGION....    QBONUS....

EAST            $2200.00
                 $500.00
                   $0.00
                 $975.00
```

What happens here is that the synthetic column @ASSOC_ROW is preserved in the dynamically normalized table SALES96_QBONUS. When the third association row (@ASSOC_ROW = 3) is deleted, it is actually filled with an empty value, thus preserving the fourth association row's position as the fourth-quarter bonus.

Loading Data into a Table

At times you may want to load data en masse into UniVerse from another database, such as ORACLE. UniVerse SQL provides a way to do this in a few short steps; you need not reenter the data row by row into the new database.

***Note**: The data-loading process handles data from first-normal-form tables only. It does not support nonfirst-normal-form data—that is, it does not support the importing of multivalued data.*

The basic steps for using the data loader are:

1. Copy the data from the original source to an operating system file on your system.
2. Write a configuration file specifying the location and format of the exported data. The data loader collects information about the format of input data from this single configuration file.
3. In UniVerse, create a table in UniVerse SQL compatible with the format of the exported data.
4. Run the data loader to load the data in bulk from one or more input files to the UniVerse table.

The following table describes the content of the configuration file, including possible content of specified data.

Configuration File Element	Description
File number	Repeated for each file defined as input.
Location	Path of the input file, relative to the current working directory.
Row separator characters	A number indicating the decimal ASCII value of the row separator character, or the character itself.
Column separator characters	A number indicating the decimal ASCII value of the column separator character, or the character itself.

Content of the Data Loader Configuration File

Configuration File Element	Description
Quote character	A decimal ASCII value or the character itself that can be used in the input file to signify that enclosed characters are taken literally (that is, it does not contain a row or column separator).
Alternate quote character	A decimal ASCII value or the character itself that can enclose a quote character to indicate that it should be treated as a literal.
Escape character	A character signifying that the next character should not be interpreted as a row or column separator, quote character, or alternate quote character.
Column number	The column number in the source file. Only the columns being used need to be described.
Source format	INTEGER (32 bit), SMALLINT (16 bit), FLOAT (IEEE format only), DOUBLE (64 bit), BYTEINT (8 bit), DECIMAL (packed decimal), CURRENCY (possible leading or trailing $ or £, or comma separators) or RAW. RAW is optional.
Width	Specifies a fixed column width. (Optional)
Next separator characters	Specifies a column separator that is different from the file's column separator. (Optional)

Content of the Data Loader Configuration File (Continued)

Configuration File Element	Description
Destination file	Specifies the start of the information about the output file.
Location	Either a file name in the VOC of the account where the program is run, a Q-pointer (account and file), a path specifying the path of the data file to which to write, or a UV/Net file pointer (*host!path*).
Create file	Yes/No or True/False, case-insensitive. Indicates that the CREATE.FILE command will be used to create the file.
Create table	Yes/No or True/False, case-insensitive. Indicates that the CREATE TABLE statement will be used to create the table.
Parameters	Indicates parameters to be passed to the CREATE.FILE command or CREATE TABLE statement.
Autosize	Yes/No or True/False, case-insensitive. Indicates whether the first autosize rows will be examined to determine table sizing parameters.
Autosize rows	If not specified, defaults to 10.
Key columns	Specifies the column names of the key columns in the destination file.
Column number	Indicates which field in the output record will contain the data described by source file number and source column number, or it can be 0 or KEY to specify that the data being described will act only as a key.
Column name	Indicates a name that may or may not correspond with a specified key column.

Content of the Data Loader Configuration File (Continued)

Configuration File Element	Description
Source file numbers	
Source column numbers	
Conversion type	Indicates the type of conversion applied to the imported data before it is written to the output file, either I (ICONV), O (OCONV), or B (BASIC).
Conversion code	I, O, or B, specifying the conversion type.

Content of the Data Loader Configuration File (Continued)

Only one of the following can be specified as True:

- Create file
- Create table
- Autosize

If none are specified as True, the file specified by location is assumed to exist.

Column name and column number must be repeated for each column in the output file.

When the configuration file is complete, enter the following command at the UniVerse prompt:

DATALOAD *pathname*

pathname is the location of the configuration file. The data loader application does the rest.

The following is an example of a data file:

```
tomr:"Tom" '"'Thomas Rand^29:77:240:"blond"green:Kevin
lisam:Lisa Michaels^28:66:130:"brown"brown:Troy
kellyv:Kelly Verock^37:72:180:"black"brown:none
paulv:Paul Vander^33:63:110:"brown"brown:Trisha
andyu:Andy Andrews^28:74:250:"blond"blue:none
ken:Ken Thompson^35:73:170:"grayy"blue:Peter
tim:Tim Tarks^35:75:180:"black"brown:none
steve:Steve Gough^35:68:180:"black"green:Nancy
ellen:Ellen Peters^33:75:220:"black"brown:Michael
derek:Derek Starks^34:66:120:"brown"brown:Karissa
```

The following example shows the corresponding configuration file:

```
FILE NUMBER: 1
   LOCATION: ../dataloader/names2.data
   Row Separator CHARACTER(S): 10
   COLUMN SEPARATOR CHARACTER(S): :
   QUOTE CHARACTER: "
   ALTERNATE QUOTE CHARACTER: '
      COLUMN NUMBER: 1
      SOURCE FORMAT: RAW
      COLUMN NUMBER: 2
      SOURCE FORMAT: RAW
      NEXT SEPARATOR CHARACTER(S): ^
      COLUMN NUMBER: 3
      SOURCE FORMAT: RAW
      COLUMN NUMBER: 4
      SOURCE FORMAT: RAW
      COLUMN NUMBER: 5
      SOURCE FORMAT: RAW
      COLUMN NUMBER: 6
      SOURCE FORMAT: RAW
      WIDTH: 7
      COLUMN NUMBER: 7
      SOURCE FORMAT: RAW
      COLUMN NUMBER: 8
      SOURCE FORMAT: RAW

DESTINATION FILE:
   LOCATION: TMP5950
   AUTOSIZE: n
   AUTOSIZE ROWS:
   CREATE FILE: N
   PARAMETER(S):
   KEY COLUMN(S): KEY1
      COLUMN NUMBER: KEY
      COLUMN NAME: KEY1
      SOURCE FILE NUMBER(S): 1
      SOURCE COLUMN NUMBER(S): 1
      COLUMN NUMBER: 1
      COLUMN NAME: COL1
      SOURCE FILE NUMBER(S): 1
      SOURCE COLUMN NUMBER(S): 1
      COLUMN NUMBER: KEY
      COLUMN NAME: KEY2
      SOURCE FILE NUMBER(S): 1
      SOURCE COLUMN NUMBER(S): 2
      COLUMN NUMBER: 2
      COLUMN NAME: COL2
      SOURCE FILE NUMBER(S): 1
      SOURCE COLUMN NUMBER(S): 2
      COLUMN NUMBER: KEY
      COLUMN NAME: KEY3
      SOURCE FILE NUMBER(S): 1
      SOURCE COLUMN NUMBER(S): 3
```

```
COLUMN NUMBER: 3
COLUMN NAME: COL3
SOURCE FILE NUMBER(S): 1
SOURCE COLUMN NUMBER(S): 3
COLUMN NUMBER: 4
COLUMN NAME: COL4
SOURCE FILE NUMBER(S): 1
SOURCE COLUMN NUMBER(S): 4
CONVERSION TYPE: O
CONVERSION CODE: MD2
COLUMN NUMBER: 5
COLUMN NAME: COL5
SOURCE FILE NUMBER(S): 1
SOURCE COLUMN NUMBER(S): 8
```

To load the data into a table, first create the table TMP5950:

```
>CREATE TABLE TMP5950 3 1 1
```

Then run the data loader, specifying the configuration filename 5950.CONFIG:

```
>DATALOAD 5950.CONFIG
```

Chapter

6

UniVerse Files and SQL

This chapter describes how file dictionaries affect SQL, the advantages and limitations of using SQL tables versus UniVerse files, and how to convert UniVerse files to tables.

How File Dictionaries Affect SQL

This section applies only to UniVerse files that are not tables.

You can issue SQL DML (data manipulation language) statements against traditional UniVerse files as well as tables. The behavior of these SQL statements (SELECT, INSERT, UPDATE, and DELETE) is affected by the contents of the file dictionaries involved.

UniVerse ODBC and UniJDBC applications use SQL statements exclusively when accessing the UniVerse database; they do not directly use RetrieVe or UniVerse BASIC. Thus it is important for users of UniVerse ODBC and UniJDBC to understand how their file dictionaries can affect the operation of SQL statements. Of course, users can also issue SQL statements from the UniVerse prompt and from within UniVerse BASIC programs (using BCI) and C programs (using UCI).

The following table shows the nine characteristics of a file or a field that are determined by the dictionary when UniVerse processes SQL statements. These characteristics are discussed individually, along with guidelines for setting up your dictionaries to control these characteristics explicitly.

Characteristic	Controlled by Dictonary Component
Data type	SQLTYPE field (8 for D- and I-descriptors; 6 for A- and S-descriptors)
Singlevalued or multivalued	SM field (6 for D- and I-descriptors; 5 for A- and S-descriptors)
Multipart record ID	@KEY phrase and @KEY_SEPARATOR X-descriptor
Association definition	ASSOC field (7 for D- and I-descriptors; 4 for A- and S-descriptors) and association phrase
Association behavior	@ASSOC_KEY.*mvname* X-descriptor
Visible fields	All D-, I-, A-, and S-descriptors are visible to SQL, but I-descriptors and fields defined by correlatives are read-only

SQL Characteristics Controlled by the Dictionary

Characteristic	Controlled by Dictonary Component
Default selected fields	@SELECT phrase
Default inserted fields	@INSERT phrase
Empty-null mapping	@EMPTY.NULL X-descriptor

SQL Characteristics Controlled by the Dictionary (Continued)

Data Types of Fields

The data type of a field affects such things as:

- What type of literal or other value can be inserted into this field
- What type of literal or data in another field can be compared to data in this field (WHERE clause)
- What type of SQL expression this field can participate in

UniVerse determines a data category for each field referenced in an SQL statement. These categories are:

- Character
- Bit
- Integer
- Scaled number
- Approximate number
- Date
- Time

Consider the following examples.

```
>INSERT INTO FILEX (FIELD1, FIELD2) VALUES (345, 'ABC');
```

This INSERT works correctly if FIELD1 has a numeric data category (integer, scaled number, or approximate number) and FIELD2 is a character field. The INSERT statement is rejected if FIELD1 is nonnumeric (character, date, or time) or if FIELD2 is noncharacter (integer, scaled or approximate number, date, or time).

```
>SELECT...FROM FILEX WHERE FIELD1 = '01/01/97'
SQL+OR FIELD2 = '12:30';
```

This SELECT statement works correctly if FIELD1 is either a date or character field and FIELD2 is either a time or character field; it will be rejected otherwise.

```
>SELECT FIELD1+5, SUBSTRING(FIELD2 FROM 1 FOR 4) FROM
FILEX;
```

This SELECT statement works correctly if FIELD1 is any category except character and if FIELD2 is character. This is because you can add an integer (5) to any number, date or time, but you can only extract a substring from a character field.

When executing an SQL statement, UniVerse determines a field's data category from information in the field's dictionary definition. You can control this explicitly by setting the SQLTYPE field (field 8 for D- and I-descriptors, field 6 for A- and S-descriptors). To set the data category for D- and I-descriptors, use the following UPDATE syntax:

UPDATE DICT *filename* **'***fieldname***' SET F8 = '***datatype***';**

To set the data category for D- and S-descriptors, use the following syntax:

UPDATE DICT *filename* **'***fieldname***' SET F6= '***datatype***';**

The values of *datatype* to use in the UPDATE statement are described in the following table.

DATATYPE	SQL Data Type Category
INT[EGER]	Integer
SMALLINT	Integer
DEC[IMAL][, *n*[, *s*]]	Integer if *s* = 0, otherwise scaled number
NUMERIC [, *n*[, *s*]]	Integer if *s* = 0, otherwise scaled number
REAL	Approximate number
FLOAT[,*n*]	Approximate number
DOUBLE	Approximate number
CHAR[ACTER][,*n*]	Character
VARCHAR [,*n*]	Character
NCHAR[ACTER][,*n*]	National character

SQL Data Types in the Dictionary

DATATYPE	SQL Data Type Category
NVARCHAR [,*n*]	National character
DATE	Date
TIME	Time
BIT [,*n*]	Bit
VARBIT [,*n*]	Bit

SQL Data Types in the Dictionary (Continued)

For example, assuming that FIELD3 is defined by a D-descriptor, to make FIELD3 of FILEX a scaled number field with a scale factor of 2, enter:

```
>UPDATE DICT FILEX 'FIELD3' SET F8 = 'DEC,9,2';
```

The next example shows how to make field SB_DESC (an A-descriptor) of file SDCONS a character field with data up to 350 characters wide:

```
>UPDATE DICT SDCONS 'SB_DESC' SET F6 = 'VARCHAR,350';
```

If the SQLTYPE field in the dictionary is empty, the conversion and format (justification) are used to determine the data type, as follows:

- A field with a D conversion is assumed to be a date.
- A field with an MT conversion is assumed to be a time.
- For a field with an MD, ML, or MR conversion, the conversion's scale factor is examined. If the scale factor is not zero, the data category is assumed to be scaled number using the scale factor from the conversion, whereas if the scale factor is zero, the data category is assumed to be integer.
- A field with a Q conversion is assumed to be an approximate number.
- A field with an MB, MO, MX, or NR conversion is assumed to be an integer.
- A field with a BB conversion is assumed to be for bit strings. A field with a BX conversion is assumed to be for hex strings.
- A field with any other conversion (except empty) is assumed to belong to the character data category.
- For a field whose conversion is empty, the format (justification) is examined. If the justification is R, the field is assumed to be an integer, if it is Q the field is assumed to be an approximate number; otherwise the field is assumed to be a character field.

Singlevalued or Multivalued Fields

Whether a field is singlevalued or multivalued affects such things as:

- Whether multivalued data can be inserted into the field using the <*x,y,z*> syntax
- How comparisons of this field to a literal or another field are performed
- Whether values in the field can be dynamically normalized by referring to a virtual table called *filename_fieldname*

Consider the following examples.

```
>INSERT INTO FILEX (NMBR1, NMBR2) VALUES (<77,88>, 99);
```

This INSERT works correctly if NMBR1 is multivalued but not if NMBR1 is singlevalued, because <77,88> denotes a multiset consisting of the two values 77 and 88. NMBR2, on the other hand, can be either singlevalued or multivalued, because 99 denotes just one value which is acceptable in either case.

```
>SELECT...FROM FILEX WHERE NMBR1 = NMBR2;
```

Suppose, in some record of FILEX, field NMBR1 contains the multiset <77,88> and NMBR2 contains the single value 88. This record will be selected by the previous query if NMBR1 is defined as multivalued, since a comparison between a multivalued field and a single quantity is considered to be true if it is true for *any* value in the multiset.

If, on the other hand, NMBR1 is defined as a singlevalued field then the record in question would not be selected by the previous SELECT statement.

```
>SELECT @ID, NMBR1, NMBR1+7 FROM FILEX_NMBR1;
```

FILEX_NMBR1 is a special notation in UniVerse SQL that refers to a dynamically normalized table, which is the virtual table created by exploding each record of FILEX into a number of rows equal to the number of rows in the field NMBR1. This SELECT statement will not work unless NMBR1 is defined as a multivalued field (and NMBR1 must not belong to an association; see "Association Definition" on page 11). UniVerse ODBC and UniJDBC applications *must* use dynamic normalization to read or write multivalued data in a UniVerse file; such applications cannot use the <*x,y,z*> syntax.

When processing SQL statements, UniVerse uses the SM field (field 6 for D- and I-descriptors, field 5 for A- and S-descriptors) to determine if a field is multivalued or singlevalued, where a value of M means multivalued and anything else means singlevalued.

***Note**: An A- or S-descriptor which defines the record ID is always treated as being singlevalued, and an A- or S-descriptor that is part of an association (field 4 = C;x or D;x) is always treated as being multivalued, regardless of the contents of field 5.*

Multipart Record IDs

UniVerse SQL assumes that the record ID for a file has just one logical part (equivalent to one column in SQL terms) unless there is special information in the dictionary, stored in an @KEY phrase and an @KEY_SEPARATOR X-descriptor. Unless this information is present, you cannot use SQL INSERT or UPDATE statements to write or replace the individual parts of the record ID.

The @KEY phrase names the computed fields (I-descriptor or a field defined by a correlative) that extract the individual parts of a multipart record ID. The @KEY_SEPARATOR X-descriptor specifies the character used in the record ID to separate the key parts (if @KEY_SEPARATOR is omitted, the separator is assumed to be a text mark).

Suppose the record ID of FILEX consists of a customer number and a date, separated by an asterisk. These fields are defined in the dictionary as I-descriptors with appropriate conversions and formats (note that ORDDATE is stored in internal date format):

```
CUSTNO      I   FIELD(@ID,'*',1)          10R
ORDDATE     I   FIELD(@ID,'*',2)    D2-   10R
```

Without any special dictionary entries these fields can be read successfully by SQL. For example:

```
>SELECT DISTINCT CUSTNO FROM FILEX;
```

However, I-descriptors and fields defined by correlatives are ordinarily treated as read-only fields by SQL and you cannot write to them. Suppose you want to use SQL to insert a new order into FILEX. Since you are not allowed to write directly to ORDDATE, you must write the entire record ID using a statement such as:

```
>INSERT INTO FILEX (@ID, ITEM,...) VALUES ('123*xxx', 'BOLT',...);
```

xxx represents the internal form of the date. This is obviously inconvenient since you would have to compute the internal form of the order's date. In order to allow full use by SQL of the I-descriptors CUSTNO and ORDDATE, you should add the following entries to the dictionary:

```
    @KEY
001 PH
002 CUSTNO ORDDATE

@KEY_SEPARATOR
001 X
002 *
```

With these entries in the dictionary, the following statements are legal:

```
>INSERT INTO FILEX (CUSTNO, ORDDATE, ITEM,...) VALUES (123,
'6/1/97',...);

>UPDATE FILEX SET ORDDATE = '5/31/97' WHERE CUSTNO = 123 AND ITEM
= 'BOLT';
```

Association Definition

Whether a multivalued field belongs to an association or not affects:

- How you can access the multivalued data in exploded form
- The results of an SQL SELECT from the dynamically normalized virtual table *filename_assocname* because the field will be padded (if necessary) with empty values up to the depth of the association

Suppose the file ORDERS contains an association called ITEMS consisting of the two multivalued fields PART and QTY with the following data:

```
ORDER_NO..  CUST_NO  PART......  QTY...

         1      123  BOLT           500
                     FLANGE         250
         2      456  SCREW         1000
         3      789  BOLT           400
                     SCREW          850
```

To explode the values of PART into individual rows of singlevalued data, enter:

```
>SELECT ORDER_NO, PART FROM ORDERS_ITEMS;
ORDER_NO..  PART......

         1  BOLT
         1  FLANGE
         2  SCREW
         3  SCREW
         3  BOLT

5 records listed.
```

The entire association ITEMS is exploded as one virtual table called ORDERS_ITEMS. To obtain all fields of the association, including the virtual column @ASSOC_ROW, enter:

```
>SELECT * FROM ORDERS_ITEMS;

ORDER_NO..  PART......  QTY...  @ASSOC_ROW

         1  BOLT           500           1
         1  FLANGE         250           2
         2  SCREW         1000           1
         3  BOLT           400           1
         3  SCREW          850           2

5 records listed.
```

But the following is not legal because PART is allowed to belong to only one virtual table, so there is no such thing as ORDERS_PART:

```
>SELECT ORDER_NO, PART FROM ORDERS_PART;
UniVerse/SQL: Table "ORDERS_PART" does not exist.
```

Suppose now you were to add order 4 (in which the QTY field for FLANGE is left empty) to the ORDERS file:

```
ORDER_NO..  CUST_NO  PART......  QTY...

         1      123  BOLT           500
                     FLANGE         250
         2      456  SCREW         1000
         3      789  BOLT           400
                     SCREW          850
         4      800  SCREW         2000
                     FLANGE
and then issue the following query:
>SELECT ORDER_NO, QTY FROM ORDERS_ITEMS;

ORDER_NO..    QTY...

         1       500
         1       250
         2      1000
         3       850
         3       400
         4      2000
         4         0

7 records listed.
```

Because QTY belongs to an association, the number of QTY values for each ORDER_NO is determined from the “depth of the association” in that record. In the case of order number 4, the depth is 2 (since the ITEMS field has two values, SCREW and FLANGE), so the virtual table ORDERS_ITEMS has two rows with ORDER_NO equal to 4. Since QTY is a numeric field, its value is displayed as 0 even though the actual stored value is empty.

When processing SQL statements, UniVerse uses the ASSOC field (dictionary field 7 for D- and I-descriptors, field 4 for A- and S-descriptors) to determine whether a field belongs to an association and, if so, to which association it belongs. For an association of D- and I-descriptor fields there must also be a phrase, naming the fields in the association, whose record ID is the association name. Also the field (if D- or I-descriptor) must be marked as multivalued in the SM field.

Association Behavior

Additional control of association behavior can be provided by adding an @ASSOC_KEY.*mvname* X-descriptor to the dictionary, where *mvname* is the name of an association or the name of an unassociated multivalued field. The @ASSOC_KEY.*mvname* record lets you specify one of the following modes of behavior:

- A specified field or fields serve as a unique key for association rows in each record.
- This association (or unassociated field) has no key, and therefore is given an artificial key called @ASSOC_ROW which is STABLE.
- This association (or unassociated field) has no key and @ASSOC_ROW is treated as an UNSTABLE key (this is the default mode of behavior).

The syntax and definition of @ASSOC_KEY.*mvname* are discussed under @ASSOC_KEY.mvname in Chapter 5, "Creating, Modifying, and Dropping Tables."

In the ORDERS table used in the previous examples you might decide that PART is the key field of association ITEMS. This means that within any one order the set of PART values must all be different. You would enter the following into the dictionary of ORDERS:

```
    @ASSOC_KEY.ITEMS
001 X
002 KEY PART
```

An example of an association with a STABLE key might be the quarterly budget for each department in a DEPTS file, where there are two fields CAPITAL and EXPENSE belonging to association BUDGET:

```
>SELECT * FROM DEPTS_BUDGET;

DEPT_NO...  CAPITAL...  EXPENSE...  @ASSOC_ROW

       100       65000      840000           1
       100       70000      865000           2
       100       70000      905000           3
       100       70000      955000           4
       200       40000      350000           1
       200       40000      380000           2
       200       40000      410000           3
       200       40000      440000           4

8 records listed.
```

This data shows the capital and expense budgets for departments 100 and 200, by quarter. The capital budget for department 100 is 65000 for the first quarter and 70000 for the second, third, and fourth quarters. Suppose the company controller decides to delete the second quarter budgets for all departments, because he needs to recompute and reenter the figures, by issuing the following SQL statement:

```
>DELETE FROM DEPTS_BUDGET WHERE @ASSOC_ROW = 2;
```

If the BUDGET association is defined as UNSTABLE then the result would be:

```
>SELECT * FROM DEPTS_BUDGET;

DEPT_NO...  CAPITAL...  EXPENSE...  @ASSOC_ROW

       100       65000      840000           1
       100       70000      905000           2
       100       70000      955000           3
       200       40000      350000           1
       200       40000      410000           2
       200       40000      440000           3

6 records listed.
```

The third and fourth quarter budgets have been shifted forward into the second and third quarters, which is not what the controller wanted. If, on the other hand, the BUDGET association had been defined as STABLE then the previous DELETE statement would cause the second quarter budget figures to be removed but their position to be retained:

```
>SELECT * FROM DEPTS_BUDGET;

DEPT_NO...  CAPITAL...  EXPENSE...  @ASSOC_ROW

       100       65000      840000           1
       100           0           0           2
       100       70000      905000           3
       100       70000      955000           4
       200       40000      350000           1
       200           0           0           2
       200       40000      410000           3
       200       40000      440000           4

8 records listed.
```

The dictionary of DEPTS should contain the following X-record to define the BUDGET association as STABLE:

```
    @ASSOC_KEY.BUDGET
001 X
002 STABLE
```

Visible Fields (Stored and Computed)

Any field defined in the dictionary can be selected by name or referred to in an SQL WHERE clause. This includes stored fields (all D-descriptors and noncorrelative A- and S-descriptors) as well as computed fields (I-descriptors and A- and S-descriptors containing a correlative in field 8). Computed fields are considered to be read-only fields unless they belong to a multipart key. Fields defined in the dictionary that are not read-only can be inserted and updated.

***Note**: Some fields defined in the dictionary may not be visible to some UniVerse ODBC applications, since the list of visible fields is created by determining the default selected fields. For details, see "Default Selected Fields."*

The data type of the dictionary entry controls the behavior of the SQL statement; for example, if field 1 has two dictionary definitions called FLD1 and ONE, which have different data types, an SQL statement referring to FLD1 will use FLD1's data type whereas a reference to ONE will use ONE's data type.

Default Selected Fields

Consider the following SQL queries:

SELECT * FROM *filename***;**

SELECT *filename***.* FROM** *filename***;**

The asterisk (*) means all columns. For a UniVerse file, the fields selected by the previous queries is determined by the contents of a phrase called @SELECT if it exists. The @SELECT phrase names all fields (whether stored or computed) to be selected by SELECT * or SELECT *filename*.*. If there is no @SELECT phrase, the @ phrase is used (plus @ID unless the phrase contains ID.SUP), and if there is neither an @SELECT nor an @ phrase, only @ID is selected.

An @SELECT phrase must contain the record ID field name (or key-part field names) and should not include the token ID.SUP. Thus, the following two phrases are equivalent to each other as far as their effect on SELECT *:

```
    @SELECT
001 PH
002 @ID FLD1 FLD2 ITYPE1

   @
001 PH
002 FLD1 FLD2 ITYPE1
```

The following phrases are also equivalent to each other:

```
    @SELECT
001 PH
002 KEYFLD FLD1 FLD2 ITYPE1

   @
001 PH
002 ID.SUP KEYFLD FLD1 FLD2 ITYPE1
```

Default Inserted Fields

If you do an INSERT into a file without naming the columns into which you are inserting, then the file's dictionary must contain an @INSERT phrase. The contents of the @INSERT phrase are used, in the order specified, as the insert-column-list for such inserts. Of course, the fields named in the @INSERT phrase should not be read-only fields.

Empty-Null Mapping

The SQL language includes the concept of a null value, meaning that the value is not known. For example, an EMPLOYEES file might have a DEPT_NAME column, which would contain null for a new employee who is in a training program until being assigned to a permanent department. In SQL, null values form the basis for what is called three-valued logic, in which a condition can be either true, false, or unknown. The clause WHERE DEPT_NAME = 'SALES' would be true for employees in the SALES department, false for employees in other departments, and unknown for trainees who have not been assigned to a department. The clause WHERE DEPT_NAME <> 'SALES' would also be unknown for unassigned employees.

UniVerse files often contain empty values whose meaning is roughly equivalent to the SQL null value. But the SQL language does not recognize an empty string as a valid piece of data. In order to allow SQL statements (issued by UniVerse ODBC, for example) to deal with empty strings in UniVerse files, UniVerse SQL provides a special mode of operation called empty-null mapping, which is activated by an X-descriptor called @EMPTY.NULL in the file's dictionary:

```
    @EMPTY.NULL
001 X
```

This mode is available only for SQL statements issued from client programs (such as those written using UCI or BCI) which connect to the UniVerse database with a particular option set. The UniVerse ODBC server sets this option by default, so empty-null mapping is normally available when you are running a UniVerse ODBC application.

When the connect option is set and the @EMPTY.NULL X-descriptor is present in the dictionary, empty values read from the file are presented to the client program as null values, and null values sent by the client are written to the database as empty values.

Suppose you have an EMPLOYEES file with the following dictionary and data (note that the dictionary contains an @EMPTY.NULL X-descriptor and that MARTINEZ's department name is empty):

```
                Type &
Field......... Field. Field........ Conversion.. Column.........
Output
Name.......... Number Definition... Code........ Heading........
Format

@ID             D    0                          EMPLOYEES
10L
@EMPTY.NULL     X
EMP_NAME        D    0
10L
DEPT_NAME       D    1
10L
@INSERT         PH   EMP_NAME DEPT_NAME


EMP_NAME..    DEPT_NAME.

HOGAN         SALES
MARTINEZ
WOODS         SALES
YANG          ACCTG

4 records listed.
```

As a result of empty-null mapping, a user of a UniVerse ODBC client program sees the following results for this SELECT statement:

```
>SELECT EMP_NAME FROM EMPLOYEES WHERE DEPT_NAME <> 'SALES';

EMP_NAME..

YANG

1 records listed.
```

The previous query does not select MARTINEZ because his department is unknown. Since it is unknown it might be SALES, so it would be wrong to report that MARTINEZ is not in the SALES department. The following query shows how to explicitly check for a null department name:

```
>SELECT EMP_NAME FROM EMPLOYEES WHERE DEPT_NAME IS NULL;

EMP_NAME..


MARTINEZ

1 records listed.
```

The following statement inserts a new employee by name only. Since no value is supplied for DEPT_NAME, this SQL statement generates a null value for that column. What is actually written into the UniVerse file is an empty string because of empty-null mapping. This can be seen from the ED command at the end.

```
>INSERT INTO EMPLOYEES (EMP_NAME) VALUES ('COHEN');
UniVerse/SQL: 1 record inserted.

>ED EMPLOYEES 'COHEN'
0 lines long.
```

Converting a UniVerse File to a Table

UniVerse lets you refer to both files and tables in your SQL (DML) statements. Many users are content to continue to use files, but there are many advantages to using tables. Working with tables, you can:

- Define views
- Define integrity constraints such as CHECK and REFERENCES
- Define multicolumn indexes
- Specify default values for columns
- Specify fine-granularity access control (by granting SELECT, UPDATE, INSERT, or DELETE privileges to named users)
- Use client ODBC programs to create, alter, and drop tables
- Use the SQL catalog that describes all schemas, tables, columns, views, and authorized users on the system

The CONVERT.SQL Command

CONVERT.SQL is a utility program that converts a UniVerse file into a table by generating a CREATE TABLE statement based on an analysis of the file's dictionary. For details about CONVERT.SQL, see the *UniVerse User Reference*. You can control the structure of the generated table (column and association definitions) by putting certain information in the dictionary before running CONVERT.SQL.

For example, suppose FILEX has two different dictionary entries that define the same field (say field 1):

```
            Type &
Field...... Field. Field....... Conversion.. Output Depth &
Name....... Number Definition.. Code........ Format Assoc.. SQL
DATA TYPE..

ONE         D    1                          10R    S
FLD1         D    1                            10L    S
```

Based on this dictionary information, CONVERT.SQL defines a data type of INT for field ONE (because it is right-justified) and a data type of VARCHAR(10) for FLD1, and chooses one or the other as the preferred column definition for field 1; let's say it chooses ONE. When the table is generated, column 1 is given a numeric data type of INT. Then, in SQL DML statements, references to either FLD1 or ONE use the data type of column 1 (INT) from the SICA. This means that a query such as the following will not be allowed because you cannot compare a numeric column to a character-string literal:

```
>SELECT * FROM FILEX WHERE FLD1 = 'ABC';
```

Because of this you may want to force CONVERT.SQL to choose FLD1, instead of ONE, as the preferred definition of column 1.

Although CONVERT.SQL analyzes any file dictionary and converts it into a well-structured table, the previous example shows a reason why you may want to force CONVERT.SQL to behave in a particular way. The following general procedure can be used to control the behavior of CONVERT.SQL so that the generated table will have the exact characteristics you want:

1. Include an @SELECT phrase in your dictionary, listing the dictionary names of all stored data fields in the file (plus whatever I-descriptors and A- and S-descriptors containing correlatives you want to show when you run a SELECT * from the table).

   This @SELECT phrase should contain exactly one name for each stored data field (so, in the previous example, it should contain the name FLD1 or the name ONE, but not both).

2. For every stored data field named in the @SELECT phrase, specify an SQL data type in that field's dictionary entry. This should be the data type that you want the generated column at that location to have.

3. For every field named in the @SELECT phrase that belongs to an association, be sure the association name appears in its ASSOC field for D-descriptors and I-descriptors, and that controlling and dependent fields are defined properly for A-descriptors and S-descriptors. Also be sure the dictionary contains an association phrase for every association of D-descriptors and I-descriptors.

4. Include an @ASSOC_KEY.*mvname* X-descriptor in the dictionary for every association. This @ASSOC_KEY.*mvname* X-descriptor defines whether the association has a key, and, if not, defines whether it should be treated as having stable or unstable @ASSOC_ROW numbers.

5. If the file has a multipart record ID, include an @KEY phrase and an @KEY_SEPARATOR X-descriptor in the dictionary, and be sure the names of the key-part fields are included in the @SELECT phrase.
6. Be sure that the dictionary entries for all fields named in the @SELECT phrase are properly marked in the SM field as being singlevalued or multivalued. For example, a field in an association should always be marked as multivalued.
7. For unassociated multivalued fields named in the @SELECT phrase, include an @ASSOC_KEY.*mvname* X-descriptor if you want to define a stable @ASSOC_ROW for this field. The default is an unstable @ASSOC_ROW.

Using CONVERT.SQL

The most straightforward way to use CONVERT.SQL is first to run the command:

CONVERT.SQL *filename* **TEST GEN**

For a list of options for the CONVERT.SQL command, see the *UniVerse User Reference*. Examine the results to see if the column and association definitions are what you want. Look especially at the data types generated for all of the columns. If a field has a right-justified FORMAT it will probably be given a numeric data type such as INT, whereas a left-justified FORMAT will result in VARCHAR. If you know that the field actually contains nonnumeric information, you will want the data type to be VARCHAR, whereas if the data is all numeric you will want INT or DEC. You can easily force the correct data type to be chosen for a column by modifying the field's dictionary entry, specifying the correct value in the SQLTYPE field:

UPDATE DICT *filename* **'***fieldname***' SET** { **F8** | **F6** } **= '***datatype***';**

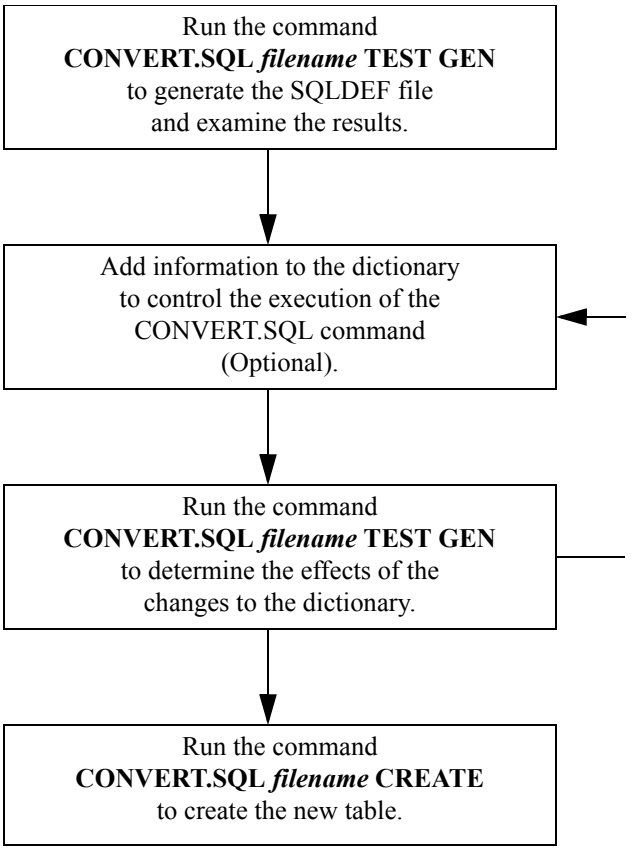
You can also force CONVERT.SQL to choose a particular field definition (among several definitions of the same field location) by naming all of your preferred field definitions in an @SELECT phrase in the dictionary:

```
>INSERT INTO DICT filename (@ID,CODE,EXP)
SQL+VALUES ('@SELECT','PH','xx yy zz');
```

After you have done this, run the CONVERT.SQL command again and iterate through the process until you are ready to do the actual conversion. At that time, use the following command to create the new table:

CONVERT.SQL *filename* **CREATE**

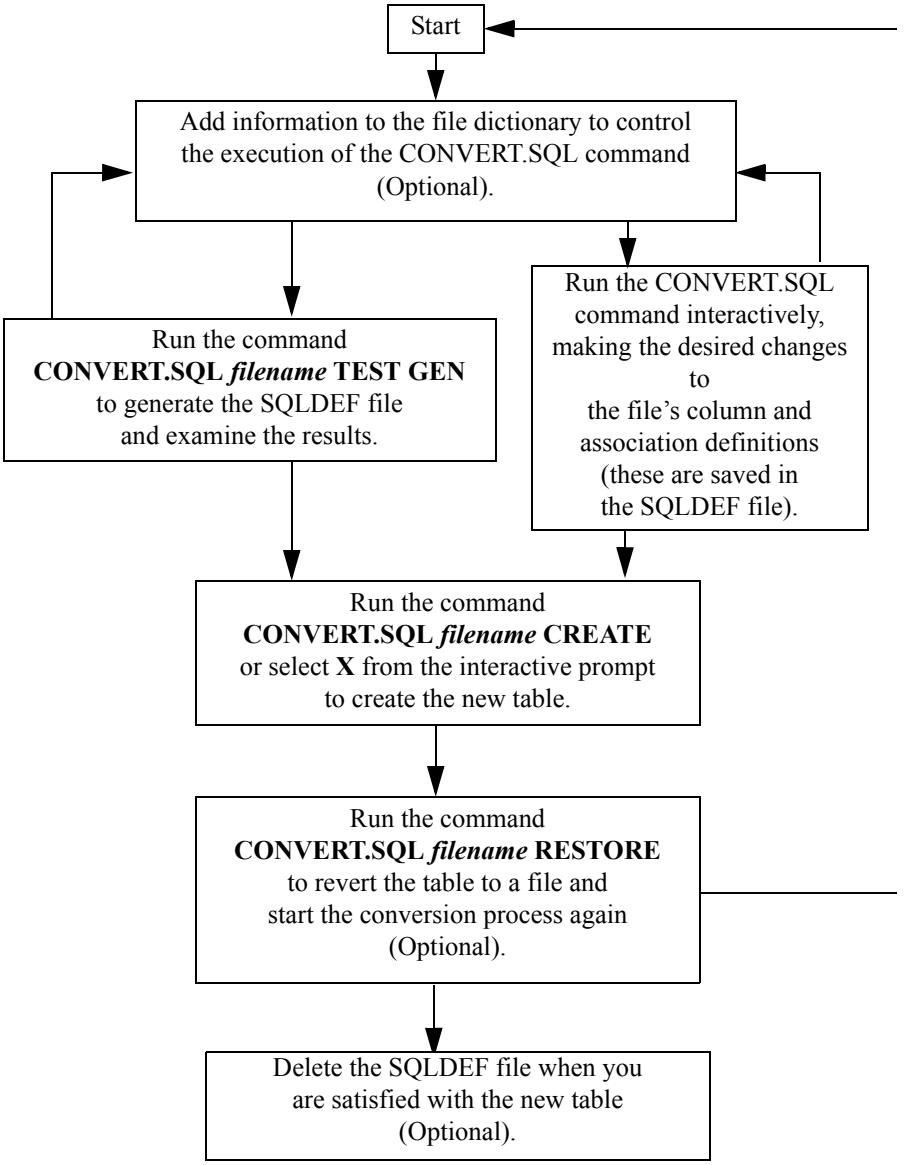
The following flowchart illustrates this process:



The next flowchart shows the process of running CONVERT.SQL in more generality. You can run the CONVERT.SQL operation interactively instead of using the TEST keyword. To run interactively, use this syntax:

CONVERT.SQL *filename*

In this mode, you can make your own changes to the generated column and association definitions by using the editing facility provided by the CONVERT.SQL command. With this facility, you can easily change the data type that was originally generated for a column to a different data type of your choice; or you can choose a different column definition than the one originally chosen by CONVERT.SQL, for a field location that is defined several times in the dictionary.


Start
Add information to the file dictionary to control the execution of the CONVERT.SQL command (Optional).
Run the command **CONVERT.SQL *filename* TEST GEN** to generate the SQLDEF file and examine the results.
Run the CONVERT.SQL command interactively, making the desired changes to the file's column and association definitions (these are saved in the SQLDEF file).
Run the command **CONVERT.SQL *filename* CREATE** or select **X** from the interactive prompt to create the new table.
Run the command **CONVERT.SQL *filename* RESTORE** to revert the table to a file and start the conversion process again (Optional).
Delete the SQLDEF file when you are satisfied with the new table (Optional).

CONVERT.SQL Example

Here is the dictionary of FILEB, which has D-descriptors, I-descriptors, a multipart key (using @KEY and @KEY_SEPARATOR), two definitions of field 1, an association AAA which includes an I-descriptor, and an @SELECT phrase:

```
                Type &
Field......... Field. Field........ Conversion.. Output Depth &
SQL
Name.......... Number Definition... Code........ Format Assoc..
DATA TYPE

@ID             D    0                           10L    S
KEY1            I      FIELD(@ID,"*"             10R    S
                       ,1)
KEY2            I      FIELD(@ID,"*"             10R    S
                       D2-,2)
@KEY_SEPARATOR  X      *
@KEY            PH     KEY1 KEY2
MULTI1          D    1                           10L    M
FIELD1          D    1                           10L    M AAA
FIELD2          D    2                           10R    M AAA
SUM             I      ADDS(FIELD1,              10R    M AAA
                       FIELD2)
AAA             PH     FIELD1 FIELD2
                       SUM
@SELECT         PH     KEY1 KEY2
                       FIELD1 FIELD2
                       SUM
```

The following command analyzes FILEB's dictionary and generates a new SQLDEF file, but doesn't convert the file to a table; it also shows the proposed CREATE TABLE statement:

```
>CONVERT.SQL FILEB TEST GEN SHOW
Analyzing 'FILEB' for conversion to SQL                  18 JUN
1997  17:43
Generating file 'FILEB_SQLDEF' .............
Table name: "FILEB"              (SQLDEF was generated  18 JUN
1997  17:43)
Columns:
 K01  "KEY1" INT FMT '10R'
 K02  "KEY2" DATE FMT '10R' CONV 'D2-'
  01  "FIELD1" VARCHAR MULTIVALUED FMT '10L'
  01B "MULTI1" VARCHAR MULTIVALUED FMT '10L'
  02  "FIELD2" INT MULTIVALUED FMT '10R'
Associations:
  01  "AAA" (01, 02)
CREATE EXISTING TABLE "FILEB"
```

```
("KEY1" INT NOT NULL FMT '10R',
 "KEY2" DATE NOT NULL FMT '10R' CONV 'D2-',
 PRIMARY KEY '*' ("KEY1", "KEY2"),
 "FIELD1" VARCHAR MULTIVALUED FMT '10L',
 "FIELD2" INT MULTIVALUED FMT '10R',
 ASSOC "AAA" ("FIELD1", "FIELD2"));
```

The column definitions and CREATE TABLE statement shown above illustrate that:

- The I-descriptors KEY1 and KEY2 show up as key-parts in the column definition and are used in the PRIMARY KEY definition in the CREATE TABLE statement.
- The conversion code D2- caused CONVERT.SQL to define KEY2's data type as DATE.
- FIELD1 is preferred over MULTI1 as the definition of field 1 because it was in the @SELECT phrase.
- Association AAA is defined as consisting of two data columns (the I-descriptor SUM is ignored as part of the table definition, but will be preserved in the association's phrase in the dictionary when a table is created).

Now suppose you want to change the definition of the table before creating it. The next command invokes CONVERT.SQL in interactive mode:

```
>CONVERT.SQL FILEB
Analyzing 'FILEB' for conversion to SQL                  19 JUN
1997  13:44
File 'FILEB_SQLDEF' exists. Do you wish to overwrite? [N] <Return>
Return accepts the existing SQLDEF file.
Table name: "FILEB"              (SQLDEF was generated  18 JUN
1997  17:43)
Columns:
 K01  "KEY1" INT FMT '10R'
 K02  "KEY2" DATE FMT '10R' CONV 'D2-'
  01  "FIELD1" VARCHAR MULTIVALUED FMT '10L'
  01B "MULTI1" VARCHAR MULTIVALUED FMT '10L'
  02  "FIELD2" INT MULTIVALUED FMT '10R'
Associations:
  01  "AAA" (01, 02)
Enter C..., D..., U..., R..., R, S, X, Q, or H for Help [R]: CK1 T
VARCHAR
```

This changes the data type of KEY1 to VARCHAR:

```
 K01  "KEY1" VARCHAR FMT '10R'
Enter C..., D..., U..., R..., R, S, X, Q, or H for Help [R]: DA1
```

This deletes association 1:

```
  Association deleted
Enter C..., D..., U..., R..., R, S, X, Q, or H for Help [R]: U1B
```

This chooses to use synonym B (MULTI) for column 1:

```
  01  "MULTI1" VARCHAR MULTIVALUED FMT '10L'
Enter C..., D..., U..., R..., R, S, X, Q, or H for Help [R]: S
```

This shows the generated CREATE TABLE statement, so you can see the effect of the interactive changes:

```
CREATE EXISTING TABLE "FILEB"
 ("KEY1" VARCHAR NOT NULL FMT '10R',
  "KEY2" DATE NOT NULL FMT '10R' CONV 'D2-',
  PRIMARY KEY '*' ("KEY1", "KEY2"),
  "MULTI1" VARCHAR MULTIVALUED FMT '10L',
  "FIELD2" INT MULTIVALUED FMT '10R');
Enter C..., D..., U..., R..., R, S, X, Q, or H for Help [R]:
X.SAVEDATA
```

This creates the table, saving its current data first:

```
Generating file 'FILEB_SQLSAVE'
Preparing to create table ......
Creating Table "FILEB"
Adding Column "KEY1"
Adding Column "KEY2"
Adding Column "MULTI1"
Adding Column "FIELD2"
```

The following updated dictionary display shows SQL data types and new phrases. Note that the @SELECT and AAA phrases still include the I-descriptor SUM. Also MULTI (preferred by you) replaces FIELD1 in the @SELECT phrase.

```
               Type &
Field......... Field. Field........ Conversion.. Output Depth & SQL
Name.......... Number Definition... Code........ Format Assoc.. DATA TYPE

@ID            D    0                           10L    S
FIELD2         D    2                           10R    M AAA   INTEGER
MULTI1         D    1                           10L    M       VARCHAR,254
@KEY_SEPARATOR X      *
@KEY           PH     KEY1 KEY2
FIELD1         D    1                           10L    M AAA
AAA            PH     FIELD1 FIELD2
                      SUM
@SELECT        PH     KEY1 KEY2
                      MULTI1 FIELD2
                      SUM
@              PH     ID.SUP KEY1
                      KEY2 MULTI1
                      FIELD2
@REVISE        PH     MULTI1 FIELD2
```

```
KEY1            I       FIELD(@ID,"*"               10R     S       VARCHAR,254
                        ,1)
KEY2            I       FIELD(@ID,"*" D2-           10R     S       DATE
                        ,2)
SUM             I       ADDS(FIELD1,F               10R     M AAA
                        IELD2)
```

The next command restores table FILEB to a file, restoring both its dictionary and its data file to their contents at the time the file was converted:

```
>CONVERT.SQL FILEB RESTOREDATA
Restoring Table FILEB to a file.
 Restoring DICT 'FILEB' (using FILEB_SQLDEF)
 Restoring DATA 'FILEB' (using FILEB_SQLSAVE)
 Deleting file 'FILEB_SQLSAVE'
File restored
```

Chapter

7

Ensuring Data Integrity

This chapter discusses *data integrity*, an important principle in relation to databases. As a DBA (database administrator) you play an important role in maintaining data integrity. Typically, a database is constructed, maintained, and used by many people. If there were no overall monitoring of the data, errors would quickly multiply.

Even with the relatively simple database used in this manual, any or all of the following errors could easily occur if no measures are taken to prevent them:

- Order entries in INVENTORY.T contain vendor IDs that have no match in the VENDORS.T table.
- A vendor ID (VENDOR_CODE) was changed in the VENDORS.T table, but the change was never reflected in the INVENTORY.T and EQUIPMENT.T tables.
- Nulls and zeros are found in primary key columns.
- The same engagement ID (LOCATION_CODE) and date appear several times in the ENGAGEMENTS.T table.
- A value of 1L500 rather than 11500 appears in the SEATS column of the LOCATIONS.T table.
- The PRICE value in a row in the INVENTORY.T table is less than the COST value in that same row.

Such errors and inaccuracies can be introduced any time you execute an INSERT, UPDATE, or DELETE statement against a table, but they are preventable. In some cases, detection and prevention are automatic (such as when a primary key column enforces unique values). In other cases they must be handled through the prudent use of certain UniVerse SQL facilities.

Data Integrity and UniVerse SQL

In UniVerse SQL there are four kinds of data integrity, as shown in the following table. The first three categories are covered on the following pages, and the last is dealt with in Chapter 9, "Transactions, Recovery, and Concurrent Access.".

Category	Description	Implementation
Entity integrity	Each value in a column must be unique.	UNIQUE, ROWUNIQUE, PRIMARY KEY
Semantic/domain integrity	Null values and empty strings are prohibited, and values must conform to certain rules.	NOT NULL, NOT EMPTY, SQL Data Types, CHECK
Referential integrity	Table-to-table, parent-child dependencies; a value in one column of one table must also exist as a value in some column in this or another table.	PRIMARY KEY, FOREIGN KEY Table Constraint, REFERENCES Privilege, ON DELETE, ON UPDATE
Consistency	An INSERT, UPDATE, or DELETE to this table also requires an INSERT, UPDATE, or DELETE to one or more other tables.	Transaction Processing

UniVerse SQL Data Integrity Constraints

You cannot apply a constraint to the data in a column retroactively. Such a constraint will purge any nonconforming values by adding a constraint with an ALTER TABLE statement. To ensure that all values conform to certain constraints, include those constraints at the time the table is created in the CREATE TABLE statement. Only table constraints (UNIQUE, CHECK, and FOREIGN KEY) can be added later, and such constraints affect only values entered after the change.

However, if you try to add a table constraint that is violated by data already in the table, you get an error message and the constraint is not added. Correct this by deleting or updating the rows causing the violation and then retry the ALTER TABLE statement.

Entity Integrity

For some columns it is important that every value be unique. This is always true for the primary key of a table, but it can apply to other columns as well. This concept of unique values is referred to as *entity integrity*, because duplicate values in such columns raise serious doubts about the accuracy of the data.

Unique Values and Primary Keys

Because each row in a table represents an individual entity in the real world, it makes sense for each row to be unique. Taking the Circus database as an example, each row in VENDORS.T represents an individual vendor source, each row in EQUIPMENT.T represents a specific piece of purchased equipment, each row in INVENTORY.T represents a specific stock item, and so forth.

The primary key of a table is one or more columns that together express this uniqueness. Looking at the Circus database, every table except one has a single column defined as the primary key of that table:

```
VENDOR_CODE. . . . PRIMARY KEY
LOCATION_CODE. . . PRIMARY KEY
ITEM_CODE. . . . . PRIMARY KEY
EQUIP_CODE . . . . PRIMARY KEY
BADGE_NO . . . . . PRIMARY KEY
ANIMAL_ID. . . . . PRIMARY KEY
CONC_NO. . . . . . PRIMARY KEY
ACT_NO . . . . . . PRIMARY KEY
```

A primary key often is a single column. However, in those tables with no single column of unique values, it can be made up of multiple columns. This is the case with ENGAGEMENTS.T, where the engagement ID (LOCATION_CODE) is not unique because a site may be booked more than once, on different dates. The primary key is defined as a combination of LOCATION_CODE and date (DATE).

```
LOCATION_CODE
"DATE"
.
.
.
CONSTRAINT PRIMARY KEY (LOCATION_CODE,"DATE")
```

If you allow a row to be entered that has the same primary key value as another row in the table, there would be no way to distinguish one from the other or to determine which was the "real" row for that entity (inventory item, location, equipment, act, and so on). Thus, SQL standards require that each primary key column (as well as any column with a uniqueness constraint) not contain any null values.

Every time you insert or update a row in a table, UniVerse SQL checks the uniqueness of its primary key (or @ID) value and rejects the insertion or update if the check fails.

Sometimes there are reasons for columns other than primary keys to contain unique values. In those cases, there are two levels of uniqueness:

- For single- or multivalued columns, uniqueness *across all rows* (UNIQUE)
- For multivalued columns, uniqueness only *within each row* (ROWUNIQUE)

Checking for Uniqueness (UNIQUE)

As an example of the first level with a singlevalued column, for every act's description to be unique, define the DESCRIPTION column as:

```
DESCRIPTION VARCHAR FMT '25T' NOT NULL UNIQUE
```

As an example of uniqueness with a multivalued column, you decide that no employee could be part of more than one act, and define OPERATOR in the ACTS.T table as:

```
OPERATOR INTEGER FMT '5L' NOT NULL UNIQUE MULTIVALUED
REFERENCES PERSONNEL
```

This definition prohibits the OPERATOR column from having duplicate values in the same row *or* in any other row in the table.

Checking for Unique Multivalues in Each Row (ROWUNIQUE)

In the case of multivalued columns, sometimes all you care about is that the multiple values in each row are unique. As an example, return to the ACTS.T table. This time an employee can be part of more than one act, but there is still no sense in having an employee ID appear more than once in an act, so define OPERATOR as ROWUNIQUE:

```
OPERATOR INTEGER FMT '5L' MULTIVALUED NOT NULL ROWUNIQUE
   REFERENCES PERSONNEL
```

UniVerse SQL syntax rules require that NOT NULL precede any ROWUNIQUE.

Consider applying ROWUNIQUE to other columns, such as all personnel, livestock, equipment, and inventory IDs found in the ACTS.T and CONCESSIONS.T tables:

```
ANIMAL_ID (in ACTS.T)
EQUIP_CODE (in ACTS.T, CONCESSIONS.T)
OPERATOR (in ACTS.T, CONCESSIONS.T)
ITEM_CODE (in CONCESSIONS.T)
```

These definitions mean that each animal, piece of equipment, employee, and inventory item can appear only once for each act or concession.

Semantic or Domain Integrity

One way to enforce data integrity is to apply certain tests to a value before it is accepted for entry into a column. There are three ways to do this:

- Testing that the value is real, and not a null value or an empty string
- Checking that the data is compatible with the data type of the column
- Applying one or more rules (CHECK) on the value

This is called *domain integrity* or *semantic integrity* and ensures that the value conforms to the characteristics defined for the column.

Testing for NOT NULL and NOT EMPTY

The simplest constraints are those requiring a column to contain "real" data values, that is, values that are neither null values nor empty strings.

If you look at the description of the Circus database, you see that many columns are described as NOT NULL. In fact, this constraint is *mandatory* for any column designated as UNIQUE or ROWUNIQUE. Primary key columns are implicitly defined as NOT NULL, so you need not explicitly define them as such.

A number of columns are defined as NOT NULL, such as:

```
DEP_NAME VARCHAR FMT '10T' MULTIVALUED NOT NULL ROWUNIQUE
VAC_TYPE CHAR(1) FMT '1L' MULTIVALUED NOT NULL ROWUNIQUE
"DATE" DATE FMT '10L' CONV 'D2/' NOT NULL
GATE_NUMBER INTEGER FMT '5R' MULTIVALUED NOT NULL ROWUNIQUE
```

You also can define a column with the NOT EMPTY keyword. The column must contain a nonempty string (a string of other than zero length) in order for the row to be accepted for insertion or updating. Because NOT NULL and NOT EMPTY have different effects, you can specify both constraints for a column.

Frequently, nulls and empty values are inserted in a column inadvertently because the person entering the data skips that column. Although this problem can be avoided by specifying a DEFAULT clause for the column, this approach is not always feasible. For example, there are no logical "default values" for name, vaccination type, or engagement date.

NOT NULL and NOT EMPTY are convenient for ensuring that *some* value is supplied. In UniVerse SQL, to check whether that value is valid, specify a data type and apply specific tests (rules) to the value.

Data Types and Domains

Unlike non-SQL UniVerse, which does not use data types, UniVerse SQL recognizes several data types. In SQL, each column is referred to as having a *domain*, a set of data values that are valid for that column. The data type of a column is one way of establishing that column's domain.

This section discusses what these data types are and how they control what values are accepted as valid.

If you have an SQL background, be aware that, in the context of UniVerse, certain SQL data types are treated as equivalent. Data types are summarized in the following table.

Data Types	Contents and Usage
NUMERIC and DECIMAL	Store decimal fixed-scale numbers, and are used for money amounts, percentages, and the other values that require precise fractional parts.
CHARACTER, VARCHAR, NCHAR, and NVARCHAR	Store any combination of numbers, letters, and special characters, and are used for names, addresses, descriptions, phone numbers, and so forth.
INTEGER and SMALLINT	Store whole decimal numbers, and are used for counts, quantities and the like.
DOUBLE PRECISION, REAL, and FLOAT	Store integer numbers, and are used for scientific values that can be calculated only approximately.
DATE and TIME	Store dates and times.
BIT and VARBIT	Store bit strings.

UniVerse SQL Data Types

For complete descriptions of these categories and data types, see the *UniVerse SQL Reference*.

Rules (CHECK)

Another way to establish a column's domain in UniVerse SQL is to define a rule for testing a value before it is accepted in that column. A rule (condition) is defined in a CHECK clause in the CREATE TABLE and ALTER TABLE statements and is entered as an SQL expression. For example:

CHECK (EST_LIFE BETWEEN 1 AND 50)

ensures that no animal is considered to have an estimated life span longer than 50 years (or less than 1 year).

A CHECK constraint can set either a column constraint (that is, part of the column definition) or a table constraint (which generally applies to more than one column in the table).

Although the Circus database was not defined CHECK constraints, here are a few instances where they might have been used to advantage.

Since the circus does not hire anyone under 18 and the mandatory retirement age is 70, set a column constraint on DOB in the PERSONNEL table to allow only values between 1924 and 1977 (using a current year of 1995):

```
DOB DATE CONSTRAINT AGE_CHK CHECK (DOB BETWEEN '1/1/24' AND
'1/1/77') FMT '10L' CONV 'D2/'
```

Because all shows are either early afternoon (2 p.m.) or late afternoon (5 p.m.) performances, specify a column constraint to restrict TIME to only those values. Note that UniVerse SQL times are like UniVerse times, with time literals expressed in military format and enclosed in single quotation marks:

```
"TIME" TIME CONSTRAINT TIME_CHK CHECK (TIME IN ('14:00',
'17:00')) FMT '10L' CONV 'MTH'
```

These two examples illustrate column constraints, because the condition referred only to the column itself. But what if you wanted to compare one column to another? Because two columns are involved, use a CHECK table constraint. One example of using a CHECK table constraint is to ensure that the cost of any entered inventory item (what you pay for it) always is less than its price (what you charge for it):

```
>CREATE TABLE INVENTORY.T (...CONSTRAINT COST_CHK
SQL+CHECK (COST < PRICE),...);
```

The CONSTRAINT *name* phrase can be included in the definition of a constraint. This is the name you use in the DROP clause of an ALTER TABLE statement to drop the constraint. Supplying a name is useful for documentation purposes. However, if you do not supply a name for a constraint, the system generates one for you. Double quoted column names are allowed within a CHECK constraint.

***Note**: You cannot use CURRENT_DATE and CURRENT_TIME in a CHECK constraint.*

CHECK Constraints and Multivalued Columns

When applying CHECK constraints to multivalued columns, multivalued logic is followed. Thus the use of the keyword EVERY in a CHECK constraint on a multivalued column requires that every value meet the criteria of the CHECK constraint when writing data into the file. Omission of the keyword EVERY implies intent of the keyword ANY: if any value in the multivalued columns meets the criteria of the CHECK constraint, data can be written.

The following example shows a CHECK table constraint involving multivalued columns. To ensure that every vaccination type (VAC_TYPE) is either R (rabies), P (parvo), or L (feline leukemia), and that every VAC_NEXT is greater than VAC_DATE, enter:

```
>CREATE TABLE LIVESTOCK.T (...
SQL+CONSTRAINT LV1 CHECK (EVERY VAC_TYPE IN ('R', 'P', 'L')
SQL+AND EVERY VAC_NEXT > VAC_DATE));
```

Referential Constraints

Another way to ensure the completeness and accuracy of your data is to set up dependencies between tables, using primary and foreign keys or the REFERENCES clause. This is known as *referential integrity.*

Certain relationships exist among the tables in the Circus database, as is the case in almost all databases. For example, each engagement has a location along with a list of rides and concessions; every vendor from whom you purchased inventory or equipment is represented in the VENDORS.T table; and all the acts, concessions, and rides use employees, equipment, and livestock.

Dependent, or Parent-Child, Relationships

These referential relationships are called *dependent,* or *parent-child, relationships*. Each VENDORS.T (parent) row has zero or more EQUIPMENT.T (child) rows with matching vendor IDs, and each EQUIPMENT.T (child) row has one VENDORS.T (parent) row with a matching VENDOR_CODE.

A referential constraint defines a dependent relationship between one column (the referencing, or child, column) and another column (the referenced, or parent, column). The referencing column becomes a foreign key. The referenced column is a primary key (or @ID if the table has no primary key) or other column containing unique values (which must be defined as UNIQUE and NOT NULL). Only values contained in the referenced column can be inserted into the referencing column (however, you can always insert null values into the referencing column).

Because of this relationship, a value about to be inserted into a child row can be verified by first comparing it to the parent table. That is, to verify that a vendor ID value about to be entered into the VENDOR_CODE column in the EQUIPMENT.T table is correct, it has to match one of the values in the VENDOR_CODE column of the VENDORS.T table.

Refer to the following table when using multivalued columns with referential constraints:

If the referencing column is...	The referenced column can be...
One singlevalued column	One single- or multivalued column
Two or more singlevalued columns (checked pair by pair)	Two or more singlevalued columns
One multivalued column (*every* value must exist somewhere in the referenced column)	One single- or multivalued column
Two or more multivalued columns (not allowed)	—

Referential Constraints

Referential Integrity

Referential integrity is important in the following situations:

- Before a new EQUIPMENT.T (child) row is inserted, referential integrity ensures that its VENDOR_CODE value has a match in the VENDORS.T table.
- Before an EQUIPMENT.T row is updated, referential integrity ensures that any change to its VENDOR_CODE value has a match in the VENDORS.T table.
- Indirectly, referential integrity also is affected if you modify the VENDORS.T table by deleting a vendor or updating the VENDOR_CODE of a vendor. Such changes would “orphan” any EQUIPMENT.T rows whose VENDOR_CODE column contained those deleted or old vendor IDs.

Scanning the database definitions, you see quite a few instances where referential integrity could be applied. Every location code (LOCATION_CODE) in the ENGAGEMENTS.T table should have a match in the LOCATION_CODE column of the LOCATIONS.T table, every vendor code in the VENDOR_CODE column of the INVENTORY.T table and the VENDOR_CODE column of the EQUIPMENT.T table should have a match in the VENDOR_CODE column of the VENDORS.T table, and so forth.

UniVerse SQL has two mechanisms for establishing referential integrity: primary and foreign keys, and the REFERENCES clause.

PRIMARY KEY Constraint

The PRIMARY KEY clause is, in itself, a constraint that imposes several restrictions on a column's content:

- Nulls are not allowed.
- The columns must be singlevalued.
- The values must be unique.

Each table can have only one primary key, but a primary key can comprise more than one column. In many cases, values in the primary key of one table can be used to maintain referential integrity for one or more columns in other tables by applying a FOREIGN KEY constraint.

FOREIGN KEY Constraint

The FOREIGN KEY table constraint defines a dependent relationship between one column (the *referencing* column) and another column (the *referenced* column). The syntax is as follows:

FOREIGN KEY (*referencing_columns*)
REFERENCES *tablename (referenced_columns*)
[**ON DELETE** *action*] [**ON UPDATE** *action*]

Implicit in the definition is the concept that any value entered in a referencing column must exist in the corresponding referenced column. The optional ON DELETE and ON UPDATE clauses are discussed later in this chapter.

If, as stated before, the values in VENDOR_CODE depend on the values in VENDOR_CODE of the VENDORS.T table, you could define VENDOR_CODE as a foreign key that references VENDORS.T.VENDOR_CODE:

```
>CREATE TABLE M... (...VENDOR_CODE INTEGER FMT '5L',
SQL+FOREIGN KEY (VENDOR_CODE)
SQL+REFERENCES VENDORS.T (VENDOR_CODE);
```

Now UniVerse SQL will match any value to be entered in VENDOR_CODE against the values in the VENDOR_CODE column of the VENDORS.T table before accepting it. If UniVerse SQL cannot find a match, it will reject the INSERT or UPDATE and return an error message. (Note that you could have omitted the actual name of the referenced column, VENDOR_CODE, because it is the primary key of the VENDORS.T table and the REFERENCES keyword refers to the primary key of the referenced table by default.)

As noted earlier, the Circus database offers many opportunities for establishing referential integrity. You could, of course, define those columns as foreign keys in the CREATE TABLE statements that originally created the Circus database. However, if you did not, use ALTER TABLE statements to add these foreign keys to the existing definitions. For example:

```
>ALTER TABLE ENGAGEMENTS.T
SQL+ADD CONSTRAINT EIDFK FOREIGN KEY (LOCATION_CODE)
SQL+REFERENCES LOCATIONS.T (LOCATION_CODE);
>ALTER TABLE INVENTORY.T
SQL+ADD CONSTRAINT IOVFK FOREIGN KEY (VENDOR_CODE)
SQL+REFERENCES VENDORS.T (VENDOR_CODE);
.
.
.
```

REFERENCES Clause

Another way to define a referential relationship between two columns is to use the column constraint REFERENCES to designate a specific column as a foreign key. It takes the same form as the REFERENCES keyword of the FOREIGN KEY clause and applies to the column being currently defined.

An example of REFERENCES is the CREATE TABLE statement for the ACTS.T table:

```
>CREATE TABLE ACTS.T (...
SQL+OPERATOR... REFERENCES PERSONNEL.T,
SQL+ANIMAL_ID... REFERENCES LIVESTOCK.T,
SQL+EQUIP_CODE... REFERENCES EQUIPMENT.T,...);
```

FOREIGN KEY Versus REFERENCES

In some situations, it is preferable to use the FOREIGN KEY clause rather than REFERENCES to define a referential relationship:

- You must use FOREIGN KEY if the key is multicolumn. Note that the key must be split up in the same way in both the parent table and the child table.
- If you are adding a foreign key constraint to an already existing table, you must use FOREIGN KEY, because you cannot add column constraints.

For example, if you are defining a new table that has a three-part foreign key, you could include one FOREIGN KEY clause in your CREATE TABLE statement:

```
FOREIGN KEY (FKEY1, FKEY2, FKEY3)
  REFERENCES TABLEA (TFKEY1, TFKEY2, TFKEY3)
```

If you are creating a table that has a single-column foreign key, it is simpler to define it with the REFERENCES keyword as part of that column's definition.

Referential Cycles

Special problems arise when table relationships form a referential cycle. An example of this is in the VENDORS.T and EQUIPMENT.T tables, where the VENDOR_CODE value in EQUIPMENT.T refers to VENDOR_CODE in VENDORS.T, and EQUIP_CODE in VENDORS.T refers to EQUIP_CODE in EQUIPMENT.T. Thus, a circular link flows in both directions between these tables: every vendor is related to one or more rows in EQUIPMENT.T, and every row in EQUIPMENT.T is related to a row in VENDORS.T.

A problem occurs when a new piece of equipment is purchased from a new vendor. For example, if both the VENDOR_CODE and EQUIP_CODE columns are defined with referential integrity and you try to insert a new row into EQUIPMENT.T first, with VENDOR_CODE containing the ID (152) for the new vendor:

```
>INSERT INTO EQUIPMENT.T VALUES (62, 232, 'Ted Schultz', 'D',
SQL+'Calliope', 75000, 20, 10, 220, '10/28/94');
UniVerse/SQL: REFERENCES Constraint Violation on column
VENDOR_CODE
Can't insert record <62>, contents <232...................>
```

UniVerse SQL will not find a value in the VENDOR_CODE column of VENDORS.T that matches the value you are trying to insert into the VENDOR_CODE column of EQUIPMENT.T because vendor 232 is not in the database.

If you reverse the order of these two statements and try the VENDORS.T insert first, you just encounter a different version of the same problem:

```
>INSERT INTO VENDORS.T VALUES (232, 'Showtime Music Co.', '2100
SQL+Plains Avenue', 'Milwaukee WI', 'USA', 'Net 90 Days',
SQL+'John Jenks', '414-555-1300', '414-555-1303',
SQL+62, 20, 30);
UniVerse/SQL: REFERENCES Constraint Violation on column EQUIP_CODE
Can't insert record <232>, contents <"Showtime.............">
```

This time UniVerse SQL will not find a value in the EQUIP_CODE column of EQUIPMENT.T that matches the value inserted into the EQUIP_CODE column of VENDORS.T because equipment item 62 is not in the database.

To get around this deadlock, be sure that when creating these tables, you define one of the foreign keys as accepting null values. Then execute the INSERT statement on that table first (with the value for the foreign key as NULL), and then on the second table. Then use an UPDATE statement on the first table to change the value of the foreign key from NULL to its real value. Continuing with the example, here's how you would make these two entries (assuming that there isn't a NOT NULL constraint on VENDOR_CODE):

```
>INSERT INTO EQUIPMENT.T VALUES (62, NULL, 'Ted Schultz',
SQL+'D', 'Calliope', 75000, 20, 10, 220, '10/28/94');
UniVerse/SQL: 1 record inserted.
>INSERT INTO VENDORS.T VALUES (232, 'Showtime Music Co.',
SQL+'2100 Plains Avenue', 'Milwaukee WI', 'USA',
SQL+'Net 90 Days', 'John Jenks', '414-555-1300',
SQL+'414-555-1303', 62, 20, 30);
UniVerse/SQL: 1 record inserted.
>UPDATE EQUIPMENT.T SET VENDOR_CODE = 232
SQL+WHERE EQUIP_CODE = 62;
UniVerse/SQL: 1 record updated.
```

ON DELETE and ON UPDATE Clauses

Until now, referential integrity implied that you could not delete or update a row in a referenced table if related rows exist in referencing tables. It implied, for example, that you could not delete a row from EQUIPMENT.T or update EQUIP_CODE to another value if EQUIP_CODE existed in ACTS.T.

But there is another way to maintain referential integrity, other than refusing to delete or update a referenced record. Specifying ON DELETE *action* or ON UPDATE *action* when defining the referential constraint accomplishes this. The CASCADE action in an ON UPDATE or ON DELETE clause causes columns in tables referencing the current (referenced) table column to be updated or deleted. Thus all values mirror one another. However, you cannot specify CASCADE in an ON UPDATE or ON DELETE clause if the initial referenced column is multivalued.

ON UPDATE specifies what action to take if a referential constraint violating update is attempted. Conversely, ON DELETE specifies what action to take if a referential constraint violating delete is attempted.

Add CASCADE (or another referential action such as SET NULL, SET DEFAULT, or NO ACTION) as part of the ON UPDATE or ON DELETE clause of a greater REFERENCES clause. For example, to specify a cascaded update or delete in the previous example of the CREATE TABLE statement for the ACTS.T table, enter:

```
>CREATE TABLE ACTS.T (...
SQL+OPERATOR... REFERENCES PERSONNEL.T,
SQL+ANIMAL_ID... REFERENCES LIVESTOCK.T ON UPDATE CASCADE,
SQL+EQUIP_CODE... REFERENCES EQUIPMENT.T,...);
```

In the previous example, if an ANIMAL_ID number is updated in LIVESTOCK.T, all occurrences of that ANIMAL_ID in ACTS.T are updated automatically to the same new value. If an animal is deleted from LIVESTOCK.T, all acts using that animal are deleted automatically.

If ON DELETE or ON UPDATE is omitted, or if NO ACTION is specified, any attempt to update or delete a referenced row will be deleted. If SET NULL or SET DEFAULT is specified, any such update (or delete) will cause values in the referencing rows to be rewritten as null or as the referencing column's default value, respectively.

Any change to a referencing table caused by CASCADE, SET NULL, or SET DEFAULT will be verified as valid with respect to the referencing table's integrity constraints. If the referencing table is also a referenced table, such a change could result in additional referential actions being performed on other tables.

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.

Referential Integrity and Multivalues

The result of ON UPDATE and ON DELETE clauses varies according to whether the values in a referenced or referencing column are single- or multivalued.

As discussed in "FOREIGN KEY Constraint" on page 15, updating rows in a referenced table also can affect rows in the referencing table. How such rows are affected is best illustrated in the following table. The tables use the following notation:

- *refgtab* is the name of the referencing table.
- *refgcol* is the name of the referencing column.
- *oldval* is the value being deleted or updated in a singlevalued referenced column.
- *newval* is the replacement value in an update.
- *oldval_1* through *oldval_n* are the values being deleted from a multivalued referenced column, or the now missing values in the case of an update to that column.
- *refgtab_refgassoc* represents the use of dynamic normalization to unnest a multivalued referencing column. If the column is unassociated, *refgassoc* is replaced by *refcol*.
- *refcol_1* through *refcol_m* are the column names of the *m* parts of a multipart referencing column set.
- *oldpart_1* through *oldpart_m* are the values in *m* parts of a multipart referenced column set that is being updated or deleted.
- *newpart_1* through *newpart_m* are the replacement values from the *m* parts of a multipart referenced column set that is being updated.

The following table shows what happens when the referencing table's REFERENCES clause includes the ON UPDATE clause.

Referenced Column	Referencing Column	Number of Parts	Action
Singlevalued	Singlevalued	1	UPDATE *refgtab* SET *refgcol* = { NULL| DEFAULT | *newval* } WHERE *refgcol* = *oldval*;
Multivalued	Singlevalued	1	UPDATE *refgtab* SET *refgcol* = { NULL | DEFAULT } WHERE *refgcol* IN (*oldval_1*, … , *oldval_n*);
Singlevalued	Multivalued	1	UPDATE *refgtab_refgassoc* SET *refgcol* = { NULL | DEFAULT | *newval* } WHERE *refgcol* = *oldval*;
Multivalued	Multivalued	1	UPDATE *refgtab_refgassoc* SET *refgcol* = { NULL | DEFAULT } WHERE *refgcol* IN (*oldval_1*, … , *oldval_n*);
Singlevalued	Singlevalued	*m* (>1)	UPDATE *refgtab* SET *refgcol_1* = {NULL | DEFAULT | *newpart_1*}, … , *refgcol_m* { NULL | DEFAULT | *newpart_m* } WHERE *refgpart_1* = *oldpart_1* AND … AND *refgpart_m* = *oldpart_m*;

Referential Integrity and Multivalues with ON UPDATE

The next table describes what happens when the referencing table's REFERENCES clause includes the ON DELETE clause.

Referenced Column	Referencing Column	Number of Parts	Action
Singlevalued	Singlevalued	1	DELETE FROM *refgtab* WHERE *refgcol = oldval*;
Multivalued	Singlevalued	1	DELETE FROM *refgtab* WHERE *refgcol* IN (*oldval_1*, … , *oldval_n*);
Singlevalued	Multivalued	1	DELETE FROM *refgtab_refgassoc* WHERE *refgcol = oldval*;
Multivalued	Multivalued	1	DELETE FROM *refgtab_refgassoc* WHERE *refgcol* IN (*oldval_1*, … , *oldval_n*);
Singlevalued	Singlevalued	*m* (>1)	DELETE FROM *refgtab* WHERE *refgcol_1 = oldpart_1* AND … AND *refgcol_m = oldpart_m;*

Referential Integrity and Multivalues with ON DELETE

Removing Integrity Constraints

The only constraints you can remove from a table definition are the table constraints UNIQUE, CHECK, and FOREIGN KEY. You cannot remove a PRIMARY KEY constraint.

To remove a constraint, use the ALTER TABLE statement with a DROP CONSTRAINT clause. DROP CONSTRAINT specifies the name of the constraint (when creating or altering a table, you can assign a name to a constraint; if you do not, the system assigns a default name).[1] For example, if you added a FOREIGN KEY constraint to ACTS.T and called it AOFK, and you now want to remove it, issue the following ALTER TABLE statement:

```
>ALTER TABLE ACTS.T DROP CONSTRAINT AOFK;
Dropping Constraint AOFK
```

When used to drop a UNIQUE constraint, the DROP CONSTRAINT clause has two options: RESTRICT and CASCADE. RESTRICT is the default, and prohibits the removal of a UNIQUE constraint if the column is referenced by a foreign key in another table. CASCADE has the opposite effect, and it not only allows removal of the UNIQUE constraint in such instances but also removes the referential constraint from the referencing (foreign key) columns.

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.

Chapter

8

Maintaining Database Security

The need for data security is obvious, especially when you consider how easy it is to access a database through interactive SQL. Without security, anyone at a terminal can probe, change, and even destroy your data at will. However, establishing blanket security on everything is not the answer; a database that only you can access is not very useful. Rather, security should be established at various levels. This is because the security needs of a particularly large production database are complex and differ at each level. For example:

- A given schema accessed by some users but not at all by others.
- Within a schema, each table accessed by some users but not others.
- Within a given table, access to data restricted on a column-by-column basis. Access also can be restricted according to function—users can be granted SELECT Privilege, DELETE Privilege, INSERT Privilege, UPDATE Privilege, or ALTER Privilege on a table, or any combination thereof.

With the additional information provided by the SQL catalog and the SICA (security and integrity constraints area) of an SQL table, UniVerse SQL offers more levels of data security and more options within those levels than UniVerse without SQL. Just as UniVerse SQL incorporates more extensive data integrity mechanisms, so does it incorporate greater database security, and for many of the same reasons. The SQL catalog contains extensive information about the database tables and users, and the SICA region contains in-depth information about the table, its columns, constraints, and permissions.

Controlling Access to Your Database

To determine what types of security you need for a particular database, ask yourself the following questions:

- *Who* should have access?
- *To what* should they have access?
- *In what manner* should they have access?

This introduces the three major elements of database security:

- Users
- Objects
- Privileges

Users

Who should have access?

Technically, a *user* is a name defined with an operating system add command. All users who have access to the database have user IDs: unique names that identify them to the system. Because every process carried out by the system is on behalf of some user, it is this user ID that determines what database objects are accessible and what SQL functions can be performed on them. In addition to a user ID, each user also has a password for greater security, and both must be entered before the user is authenticated to the system.

All registered UniVerse SQL users have CONNECT Privilege and are defined in the UV_USERS table of the SQL catalog.

In some installations, groups of users have similar needs and consequently are assigned similar privileges. In the Circus database, everyone in the Concessions Manager's office might be granted similar privileges on the CONCESSIONS.T and INVENTORY.T tables, and everyone in the Procurement department might be given the same privileges on the INVENTORY.T and EQUIPMENT.T tables. However, only the Human Resources department and a few high-level managers might be granted privileges on the PERSONNEL.T table because of the sensitive data that it contains.

In some cases you might want to grant privileges to everyone; PUBLIC is a group that includes *all* UniVerse users, whether or not they are defined in the SQL catalog.

Database Objects

To what objects should each user be granted access?

An *object* can be a schema, a table in a schema, or one or more columns or rows in a table. Access to objects is controlled hierarchically; you cannot have access to a table unless you first have access to the schema in which it resides, and you cannot have access to a particular column or row of a table unless you first have access to the table itself.

Access to tables is gained either by being the owner (creator) of the table or by being granted privileges to that table by someone who has those privileges and the authority to grant them.

Privileges

In what manner should users have access?

The answer to this question is determined by *privileges*, a set of permitted actions that a user can perform on an object. For example, some users might have only SELECT privilege on the ENGAGEMENTS.T table (that is, they can execute only SELECTs against that table, but not an INSERT, DELETE, UPDATE, or ALTER), while other users may have SELECT, INSERT, DELETE, and UPDATE privileges on the same table.

Privileges consist of database privileges and table privileges. Database privileges are grouped at three levels:

- CONNECT is the lowest level and registers a user as a UniVerse SQL user, with certain limited privileges (mainly restricted to the tables or views they have created).
- RESOURCE Privilege includes all CONNECT capabilities, and adds the ability to create schemas.

- DBA Privilege (database administrator), the highest level of database privilege, allows one to do everything (much like a superuser), incorporating all RESOURCE privilege plus the power to create schemas and tables for other users, grant database privileges, grant table privileges on any table and view to any user, revoke the database or table privileges of any user, and access all tables.

The database privilege state of each user is recorded in the UV_USERS table of the SQL catalog. Just being listed in the UV_USERS table implies that the user has CONNECT privileges. In addition, two bytes, one for RESOURCE and one for DBA, indicate whether the user has been granted privileges for either or both of those levels.

Table privileges (applicable to views as well as tables) are grouped by process: SELECT, INSERT, UPDATE, DELETE, REFERENCES, ALTER, and ALL PRIVILEGES. A user obtains any of these privileges for a table in two ways:

- By creating a table, a user becomes its owner and has full privileges to it automatically. Similarly, if a user creates a view for a table, he or she is the owner of that view and has full privileges to the view, but not necessarily to the table.
- By being granted specific privileges to the table. Privileges can be granted by the table's owner, by someone to whom the same privileges have been granted previously, or by someone with DBA privileges.

SQL security on tables and views is enforced for all attempts at access, whether by SQL statements, UniVerse commands and utilities, or UniVerse BASIC programs. Note that a UniVerse BASIC program can write into a table only if the user has both INSERT and UPDATE privileges. Views are discussed in the *UniVerse SQL User Guide*.

Granting Privileges

Except for obtaining certain privileges automatically, either because of one's database privilege level or because one creates (and therefore owns) a table or view, the only way to obtain privileges is through the GRANT statement.

Granting Database Privileges

When UniVerse SQL is first installed, only one user is registered to use SQL. That user is either *uvsql*, *root*, or *uvadm*, depending on how UniVerse was installed. That user is also granted DBA authority. Only a DBA can grant or revoke database privileges and create schemas for other users.

The GRANT statement syntax for granting database privileges is:

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

users is a comma-separated list of user IDs.

You cannot grant RESOURCE or DBA privilege to any user who does not already have CONNECT privileges. If you try to grant a privilege to a user who already has that privilege, the grant request is ignored.

CONNECT

CONNECT is the lowest level of database privilege and is the minimum authority assigned to anyone registered as a UniVerse SQL user. To register a new user at this level, the DBA uses the following syntax:

GRANT CONNECT TO *user***;**

Users granted CONNECT authority can:

- Create tables (and become their owners)
- Alter and drop any tables they own
- Grant and revoke privileges on tables they own and tables for which they have the GRANT OPTION
- Use the SELECT, INSERT, UPDATE, and DELETE statements on tables to which they have access

- Create and drop views on tables to which they have access

For example, to register three new users (with IDs of *sam*, *joyce*, and *john*), and grant them CONNECT privilege, enter:

```
>GRANT CONNECT TO sam, joyce, john;
Granting privilege(s).
```

RESOURCE

To register these new users with a higher level of database authority, allowing them all aspects of CONNECT authority as well as the ability to create their own schemas, grant RESOURCE privilege. For example, enter:

```
>GRANT RESOURCE TO sam, joyce, john;
Granting privilege(s).
```

Note that you *must* grant CONNECT privilege before granting RESOURCE privilege.

DBA

A DBA has the most power and can grant any privilege to (or revoke the privilege of) any user, create schemas (and tables and views within the CREATE SCHEMA statement) for *other* users (users with CONNECT and RESOURCE levels can create database objects only for themselves), and access any table or view, regardless of who owns it. DBA authority includes both the CONNECT and the RESOURCE privileges.

Although only one user is designated DBA when UniVerse SQL is installed, that user can create other DBAs at any time. To grant DBA (and therefore RESOURCE and CONNECT) privilege to two other users, *daveb* and *wendyj*, enter:

```
>GRANT DBA TO daveb, wendyj;
Granting privilege(s).
```

Granting Table Privileges

Table (and view) privileges, allowing users to execute ALTER, DELETE, INSERT, REFERENCES, SELECT, and UPDATE statements, can be granted by:

- A DBA

- Anyone who owns a table or view
- Anyone who has been granted those privileges with a WITH GRANT OPTION by someone else

The syntax for the GRANT statement that grants table privileges is:

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

When you create a table, you are the owner of that table and the only user with privileges on it (except for any DBAs). As such, you can grant any or all of those privileges to others. If you are not the creator/owner of a table, you have only those privileges that others have granted to you.

The next sections discuss the various table privileges and the clauses that designate the recipients of the grant and determine whether those recipients can pass it on.

Letting Other Users Change a Table (ALTER Privilege)

Granting the ALTER Privilege allows a user to change the structure of a table. This includes adding new columns, adding or removing table constraints or associations, and setting or removing default specifications using an ALTER statement.

For example, to grant ALTER privilege on the EQUIPMENT.T table to *sheilaf* and *bonnieb*, enter:

>**GRANT ALTER ON EQUIPMENT.T TO shielaf, bonnieb;**
Granting privilege(s).

Letting Other Users Delete Rows from a Table (DELETE Privilege)

Granting the DELETE Privilege allows a user to remove rows from a table using a DELETE statement. You can grant DELETE privilege only on the entire table.

The following statement grants DELETE privilege on the CONCESSIONS.T table to *stevew*:

>**GRANT DELETE ON CONCESSIONS.T TO stevew;**
Granting privilege(s).

Letting Other Users Add Rows to a Table (INSERT Privilege)

Granting the INSERT Privilege allows a user to add new rows to a table using an INSERT statement.

To grant INSERT privilege on the VENDORS.T table to *barbaram*, enter:

```
>GRANT INSERT ON VENDORS.T TO barbaram;
Granting privilege(s).
```

Letting Other Users Define Foreign Keys (REFERENCES Privilege)

Granting the REFERENCES Privilege allows a user to place referential constraints (define foreign keys) on columns. As with UPDATE, you can grant the REFERENCES privilege to all the columns of the table, or you can restrict permission to selected columns. If you grant the REFERENCES privilege to an entire table, the other user can create his or her own tables and include a referential constraint pointing to this table. If you grant REFERENCES to only specific columns, the other user can define referential constraints that point to only the specified columns in this table.

To grant the REFERENCES privilege on all columns of the PERSONNEL.T table to *chuckt*, enter:

```
>GRANT REFERENCES ON PERSONNEL.T TO chuckt;
Granting privilege(s).
```

To grant the REFERENCES privilege on only the BADGE_NO column, enter:

```
>GRANT REFERENCES ON PERSONNEL.T (BADGE_NO) TO
chuckt;
Granting privilege(s).
```

Letting Other Users Read from a Table (SELECT Privilege)

Granting the SELECT Privilege grants “look but don’t touch” permission, allowing a user to read the data in a table using, for example, a SELECT statement or BASIC program, but not to insert, delete, or update it.

You can grant SELECT privilege only on the entire table. If you want to restrict the privilege to particular columns, define a view on the table and grant SELECT privilege on the view.

To grant read-only privileges on the ENGAGEMENTS.T table to *paulc*, enter:

```
>GRANT SELECT ON ENGAGEMENTS.T TO paulc;
Granting privilege(s).
```

Letting Other Users Update Existing Rows (UPDATE Privilege)

Granting the UPDATE Privilege allows a user to update existing rows in a table, using an UPDATE statement. As with REFERENCES, you can grant UPDATE privilege to all the columns of the table, or restrict permission to selected columns.

To grant UPDATE privilege to *jackd* for any column of the INVENTORY.T table, enter:

```
>GRANT UPDATE ON INVENTORY.T TO jackd;
Granting privilege(s).
```

Because you did not include WITH GRANT OPTION, *jackd* will not be able to assign this privilege to anyone else. Note that when you grant UPDATE privilege on all the columns of a table, it applies not only to existing columns but to any columns that may be added later.

If *jackd* is responsible for setting prices, and you want to grant him permission to look at all columns in the INVENTORY.T table but be able to update only the PRICE column, issue the following grants:

```
>GRANT SELECT ON INVENTORY.T TO jackd;
Granting privilege(s).
>GRANT UPDATE (PRICE) ON INVENTORY.T TO jackd;
Granting privilege(s).
```

If *sandyh* is a stock clerk who needs to see the entire INVENTORY.T table but should be allowed to update only the quantity on hand (QOH), grant SELECT and UPDATE privileges as follows:

```
>GRANT SELECT ON INVENTORY.T TO sandyh;
Granting privilege(s).
>GRANT UPDATE (QOH) ON INVENTORY.T TO sandyh;
Granting privilege(s).
```

Granting Multiple Privileges

You can specify more than one table privilege in a GRANT statement. To grant SELECT, INSERT, and DELETE privileges on the LIVESTOCK.T table to *lindah*, enter:

```
>GRANT SELECT, INSERT, DELETE ON LIVESTOCK.T TO lindah;
Granting privilege(s).
```

To grant all of the grantable privileges you possess on a table, specify ALL PRIVILEGES as *table.privileges*. To grant all privileges on the ENGAGEMENTS.T table to *davidb*, enter:

```
>GRANT ALL PRIVILEGES ON ENGAGEMENTS.T TO davidb;
Granting privilege(s).
```

Specifying the Recipient of the GRANT (TO Clause)

The TO *users*|PUBLIC clause indicates the users to whom you are granting permission.

If you specify TO *users*, those users named must be defined in the SQL catalog. A grant can be to one or to multiple users; in the latter case, separate the users by commas as shown:

```
>GRANT SELECT ON ENGAGEMENTS.T TO celiaw, tedj, billg, joyced;
Granting privilege(s).
```

If you specify TO PUBLIC instead, you are granting the privileges to all UniVerse users, whether or not they are in the catalog. The following statement grants SELECT privilege on the LOCATIONS.T table to all UniVerse users:

```
>GRANT SELECT ON LOCATIONS.T TO PUBLIC;
Granting privilege(s).
```

Be aware that PUBLIC grants permission to current *and* future SQL users. One advantage of using PUBLIC is that you will not have to explicitly grant privileges to new users as they are added. However, the disadvantage is that you no longer have control over the grant (unless you revoke it).

Passing on the Privileges (WITH GRANT OPTION Clause)

Including WITH GRANT OPTION allows those users to whom you are granting privileges to pass them on to others. If you do not include WITH GRANT OPTION, they can take advantage of the privileges themselves, but they cannot pass them on.

For example, to grant all privileges on the PERSONNEL.T table to *susanc* and allow her to pass on those privileges to others, enter the following statement:

```
>GRANT ALL PRIVILEGES ON PERSONNEL.T TO susanc
SQL+WITH GRANT OPTION;
Granting privilege(s).
```

Now *susanc* has the option of passing along any or all of the privileges she has been granted to other users. So she decides to give *bobm* SELECT privilege to allow him to view the table and enters:

```
>GRANT SELECT ON PERSONNEL.T TO bobm;
Granting privilege(s).
```

Because the preceding GRANT statement does *not* include the WITH GRANT OPTION, the permissions chain ends here; *bobm* cannot pass on his SELECT privilege to someone else.

Note that the WITH GRANT OPTION applies to all the privileges listed in the statement. To allow the recipient to pass on only some of the privileges you are granting to him or her, issue two separate GRANT statements as follows:

```
>GRANT SELECT, INSERT ON PERSONNEL.T TO bobm WITH
GRANT OPTION;
Granting privilege(s).
>GRANT ALTER, UPDATE, DELETE ON PERSONNEL.T TO
bobm;
Granting privilege(s).
```

Here, *bobm* can pass on only the SELECT and INSERT privileges to other users.

Revoking Privileges

Any privileges you grant, you can also take back using the REVOKE statement. The syntax for REVOKE parallels that of the GRANT statement, specifying a set of privileges, the object to which they apply, and the users.

Revoking Database Privileges

Only a user with DBA privilege can revoke database privileges. If you revoke a user's DBA privilege, the user still retains RESOURCE and CONNECT privileges. If you revoke a user's RESOURCE privilege, the user still has the CONNECT privilege.

The syntax for revoking database privileges is:

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

For example, to revoke RESOURCE privilege from *jamesd*, enter:

>**REVOKE RESOURCE FROM jamesd;**

You must revoke database privileges from the top down. For example, you cannot directly revoke the CONNECT privilege of a user who has DBA or RESOURCE privilege. Attempting to do so will return an error. Instead, if a user has RESOURCE privilege, first revoke the user's RESOURCE privilege (which leaves the user with the CONNECT privilege), and then revoke the CONNECT privilege. If you try to revoke a privilege that the user does not have, the request is ignored.

If you revoke the CONNECT privilege from a user, any schemas or tables owned by that user will have their ownership changed to *uvsql*, *root,* or *uvadm*, (whichever owns the CATALOG schema).

Revoking Table Privileges

The syntax for revoking table privileges is:

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

As the second syntax indicates, you can remove just the WITH GRANT OPTION on a privilege, or you can revoke the privilege itself. Revoking a privilege also revokes any GRANT OPTION on that privilege.

Because you already are familiar with the choices in the GRANT statement, you should be able to follow these REVOKE statements (the privileges revoked are those granted by some of the previous examples of the GRANT statement):

```
>REVOKE RESOURCE FROM sam;
Revoking privilege(s).
>REVOKE ALTER ON EQUIPMENT.T FROM shielaf, bonnieb;
Revoking privilege(s).
>REVOKE SELECT, UPDATE ON INVENTORY.T FROM jackd;
Revoking privilege(s).
>REVOKE SELECT ON LOCATIONS.T FROM PUBLIC;
Revoking privilege(s).
>REVOKE UPDATE (QOH) ON INVENTORY.T FROM sandyh;
Revoking privilege(s).
```

If you have the power to grant a privilege, you can revoke that privilege, no matter how a user obtained the privilege, whether from you or from someone else.

REVOKE and WITH GRANT OPTION

Because REVOKE has no CASCADE option, even when you revoke a user's previously issued GRANT OPTION, any privileges that this user passed on to other users as a result remain unaffected.

For example, earlier you granted ALL PRIVILEGES WITH GRANT OPTION on the PERSONNEL.T table to *susanc*, and she in turn passed on the SELECT privilege to *bobm*. Later, you issue a REVOKE statement, taking back the GRANT OPTION:

```
>REVOKE GRANT OPTION FOR ALL PRIVILEGES ON
PERSONNEL.T
SQL+FROM susanc;
Revoking privilege(s).
```

This revokes *susanc*'s GRANT OPTION privilege, but has no effect on *bobm*, who can continue retrieving information from the PERSONNEL.T table. The same would hold true if you revoked *susanc*'s SELECT privilege itself: *bobm* would still retain the privilege.

REVOKE and Overlapping GRANTs

Sometimes a user will be granted the same privilege by more than one user, creating a situation known as an overlapping GRANT.

If you and another user grant a third user the same privilege, and you revoke yours, the other (duplicate) one is also revoked.

Chapter

9

Transactions, Recovery, and Concurrent Access

While databases are in use, systems can fail, errors and discrepancies can occur, and not all operations necessarily run to a successful completion. If whole databases are lost or destroyed, they must be recreated. Tables are accessed by many users, who may be inserting new rows, or deleting or modifying existing rows in the same table at the same time. Managing these situations is the DBA's responsibility and involves *transaction processing*, *transaction logging and recovery*, and *concurrent access*.

Transaction Processing

A *transaction* is a series of statements that are treated as a single event. Any changes made to the database by a transaction are guaranteed either to go to completion or to have no effect at all. Transactions and transactional processing are explained fully in *UniVerse BASIC*.

An example of a familiar transaction occurs at a bank ATM (automatic teller machine, or cash machine): transferring funds from your savings account to your checking account. If the system crashed just after deducting the money from savings but before depositing it in checking, what would be the state of your account?

There are two possibilities: either the system remembers to deposit the money in your checking account when service is restored, or it reverses (rolls back) the deduction from your savings and leaves everything as it was before the transaction began.

If someone were watching this at a terminal, he or she would know if each activity proceeded to completion and could take the necessary measures if it did not (such as manually retyping the “deposit to checking” request). But when the statements executed are part of a program, there is no way to directly monitor what is happening or to take steps to correct any omissions. Therefore, a transaction needs to be coded for all possible outcomes.

Transaction processing is designed for situations like the ATM transaction described here and introduces the subject of programmatic SQL.

Transaction Processing in Programs

To illustrate the principles of SQL usage, this manual uses *interactive* SQL—SQL statements typed at a terminal that return an immediate response. Interactive SQL statements are issued as UniVerse commands and deliver their output to a terminal screen or a printer. There is another kind of SQL—*programmatic* SQL.

Programmatic SQL is a set of SQL statements that can be embedded within and executed by a C-language program (using the UniVerse Call Interface, or UCI) or a UniVerse BASIC program (using the UniVerse BASIC SQL Client Interface, or BCI). Programmatic SQL statements are part of the program code and deliver their output to data variables defined within the program.

Generally, the SQL statements used in BASIC and C programs are the same statements that you can issue interactively. However, there are some differences at the detail level between programmatic SQL and interactive SQL. More information about programmatic SQL is in *UniVerse SQL Reference*. For information about UCI and the BASIC SQL Client Interface, see *UCI Developer's Guide* and *UniVerse BASIC SQL Client Interface Guide*, respectively.

Procedurally, a transaction is a series of statements that begin when a BEGIN TRANSACTION statement is executed and ends when either a ROLLBACK or a COMMIT statement is executed. An END TRANSACTION statement marks the point where processing continues after the transaction ends.

Within the transaction sequence, there must be either a COMMIT statement or a ROLLBACK statement. COMMIT applies all changes made during the transaction if everything proceeded normally. ROLLBACK cancels all changes made during the transaction if some error occurred. For example:

```
BEGIN TRANSACTION
   READU data1 FROM file1, record1 ELSE ROLLBACK
   READU data2 FROM file2, record2 ELSE ROLLBACK
      .
   [process data]
      .
   WRITE new.data1 ON file1, record1 ELSE ROLLBACK
   WRITE new.data2 ON file2, record2 ELSE ROLLBACK
   COMMIT WORK
END TRANSACTION
```

As an application performs writes or deletes on tables, the actual data is stored in a cache (temporary storage), not in the table.While a transaction sequence is being executed, an application read is satisfied from the cache if such an entry exists. Data is not written to disk or available to other users until the COMMIT statement is executed, at which time all data changes are written to their respective tables.

Locking is not automatic, but any locks that are explicitly set during the transaction are not released until COMMIT or ROLLBACK is executed, at which time all locks acquired during the transaction are released.

Transaction Processing and UniVerse SQL

Within UniVerse, transaction processing in interactive SQL is different from transaction processing in programmatic SQL. UniVerse SQL is designed to operate within a transactional environment. For that reason, each individual interactive UniVerse SQL statement is treated as a transaction. Each time you enter an SQL statement, it begins a new transaction. Until that transaction finishes, the results are not available to you or other users, nor are they permanently committed to disk. To terminate the transaction, an implied COMMIT (called autocommit) is issued by the system at the completion of an SQL statement, making each such interactive statement an independent transaction.

In interactive SQL you cannot group multiple SQL statements into a single transaction because you cannot enter BEGIN TRANSACTION, COMMIT, ROLLBACK, or END TRANSACTION statements interactively.

In programmatic SQL a transaction is one or more SQL or non-SQL statements bounded by BEGIN TRANSACTION, COMMIT, ROLLBACK, and END TRANSACTION statements. In addition, transactions can be nested, that is, a program can initiate a new transaction while another transaction is still active.

Consequently, transaction processing is relevant only with respect to either programmatic SQL transactions or non-SQL transactions.

As an example of a transaction rollback, assume that the circus bought out a smaller circus and is merging those personnel records with their own, treating their EMP_NO as the BADGE_NO primary key. The transaction proceeds until an EMP_NO is encountered that duplicates a BADGE_NO in your PERSONNEL.T table and causes a “Duplicate Primary Key” error. The entire INSERT operation is rolled back automatically, and no new employees are added.

Database Recovery

A number of events can disrupt processing and corrupt or even destroy your database. These include media failures (such as disk crashes), system failures, and power outages. In such an event, you need some way to recover your database and restore it to its most recent state. Several components work together to enable a database recovery:

Component	Description
File backup	Copying all files in an account (or, optionally, each file separately) periodically.
Transaction logging	Copying every file update to a log file.
Media recovery	Restoring the database to a usable state if the data becomes corrupted by a media failure (a damaged disk, for example). This is done by taking the backup copy and rolling forward all completed transactions (from the transaction log) that were executed from the time the backup was made to the point where the media failure occurred.
Warmstart recovery	Handling certain file structuring and other system-level details that may be left unfinished when a system failure (such as a power outage or system crash) occurs.

Components for Database Recovery

Most of these actions, such as backing up files, setting up transaction logging, identifying the files that should be activated for logging, and recovering media, are the responsibility of the system administrator, not the individual users.

File Backup

Periodically copying all disk files onto an offline storage medium, such as magnetic tape or a second disk, is an accepted computing practice. If files are lost or destroyed, they can be restored from these backup copies.

Each installation chooses its own strategy for implementing file backup and restoration. You can use the **Backup** and **Restore** options of UniVerse Admin, the UniVerse *uvbackup* and *uvrestore* commands, or the appropriate backup and restore commands for the operating system. When you use the UniVerse backup facilities, you can choose to perform either a full backup (making backup copies of all files), or an incremental backup (copying only those files that are new or have changed since the last backup).

The procedures for backing up and restoring files in UniVerse are covered in *Administering UniVerse*.

Transaction Logging

File backup is an important step in protecting your database, but what happens if you do a backup on Monday, and suddenly on Thursday your disk crashes? Does that mean you have lost three days of work? If you have been using transaction logging, the answer is no.

Transaction logging is the first part of the two-part UniVerse database recovery process. The second part is media recovery (covered later in this chapter) and warm-start recovery. Basically, transaction logging is the recording of each file update applied to the database.

The variables for transaction logging affect how transactions are both executed and logged. Transaction logging may be set up in one of two modes, activated or not, and each table or file may be designated recoverable or nonrecoverable.

Setting Up Transaction Logging

Unless you activated the transaction logging system, recoverable files will not be logged. Besides creating a log directory and creating several log files in that directory, you also determine whether to run logging in archive mode (primarily designed for media recovery) or checkpoint mode (primarily designed for warmstart recovery), or both.

Once you set up transaction logging, the next task is to designate which of the tables in the database are to be made recoverable.

Making Tables Recoverable

It is up to you to decide which tables and files are to be made recoverable. Then the user can implement transaction logging either table-by-table or by account. Once a user activates a table for logging, the logging of all file updates to that table is automatic. But updates to nonrecoverable tables are not written to the log file, and therefore such tables are not recoverable.

Many considerations determine whether a table should be recoverable: the importance of the data, whether the table can be recreated quickly and economically, the cost (in terms of system performance) of making the table recoverable, how frequently the table is backed up, and so on. Tables and files are made recoverable through the transaction logging menus, as described in *UniVerse Transaction Logging and Recovery*. For example, assuming that you already set up transaction logging (as described in the previous section), designate the ENGAGEMENTS.T and LOCATIONS.T tables in the Circus account as recoverable, or a user could make all the tables in the account recoverable.

Setting the Status of the Transaction Logging Subsystem

You can enable, disable, or suspend the transaction logging subsystem itself through the transaction logging menus (or through the UniVerse ENABLE.RECOVERY, SHUTDOWN.RECOVERY, and SUSPEND.RECOVERY commands). For more information about these menus and commands, see *Administering UniVerse* and the *UniVerse User Reference*.

Media Recovery

When a disk crash or other media failure occurs, you need to use recovery procedures to restore your database. The two approaches to doing this are media recovery and file recovery, both detailed in *UniVerse Transaction Logging and Recovery*. Media recovery, the more complex of the two, restores multiple files, whereas file recovery rolls forward updates on a specific file.

Warmstart Recovery

If the system crashes, or UniVerse fails or is shut down in an uncontrolled manner, files on disk may be left in an inconsistent state. If such inconsistencies are detected when UniVerse is brought back up again (with logging enabled in checkpointing mode), a process called warmstart recovery is initialized automatically.

Warmstart recovery uses information from the log files to restore the structure and integrity of the database. First, each file is brought to a structurally consistent state (correcting such problems as invalid overflow block pointers and partially updated index files). Then the database is brought to a logically consistent state, in which each transaction has been either fully committed to disk or rolled back. Warmstart recovery is discussed in greater detail in *UniVerse Transaction Logging and Recovery.*

Concurrent Access

If every user had exclusive access to a database, and transactions against that database were executed serially, results would be totally predictable. However, when multiple transactions are run concurrently against a database, reading from and writing to the same data rows, conflicts can produce unpredictable and erroneous results.

The problems that can arise from concurrent access of a database can be grouped into four classic categories:

- Lost updates. Both transactions A and B attempt to update the same row. Because each transaction reads and updates its own "copy" of the row, the last transaction to write an update to that row overwrites the update of the first transaction.
- Dirty reads. Transaction A updates a row, and transaction B reads the updated row and proceeds to modify it. Meanwhile transaction A aborts (thus restoring the row to its original state), and then transaction B writes back its update (which is based on values no longer valid).
- Nonrepeatable reads. Transaction A reads a row, then transaction B updates that row. Later, transaction A rereads the same row and sees different values than it saw initially.
- Phantom discrepancies. Transaction A determines the number of rows that meet certain selection criteria (say the COUNT of cars in stock that are red), then transaction B adds another red car into inventory. Later, transaction A requests the SUM of costs of all red cars and divides that sum by the original count. The result is an artificially high average cost for red cars, because the set of rows meeting the selection criteria changed during the transaction.

All of these situations can be resolved to some degree by the use of locks and isolation levels.

You can explicitly set locks and isolation levels only in a UniVerse BASIC program. Even though there are no explicit SQL statements for setting locks and isolation levels, you should have a general understanding of how locks and isolation levels work because their activation by UniVerse BASIC programs in a session can affect how your SQL statements operate. Also, each SQL transaction automatically does the appropriate locking, as explained in "Isolation Levels" on page 14.

The ultimate purpose of locks and isolation levels is to provide *serializability*, which ensures that the output of any set of concurrent transactions is the same as that produced by running the individual transactions serially in some specific order, with each of the transactions having exclusive use of the system during its execution.

Locks

UniVerse record and file locks control access to records and files among concurrent user processes.

Lock compatibility determines what a user's process can access while other processes have locks on records or files. Record locks allow more compatibility because they coexist with other record locks, thus allowing more transactions to take place concurrently. However, these "finer-grained" locks provide a lower isolation level. File locks enforce a high isolation level and more concurrency control, but less compatibility. For information about transaction processing and isolation levels, see *UniVerse BASIC*.

UniVerse supports the following locks (in order of increasing strength):

- Shared record lock
- Update record lock
- Shared file lock
- Intent file lock
- Exclusive file lock

Whenever you use an SQL statement to access a table, you may find that a lock has been placed on that table or on one or more rows in that table. If this happens, your statement must delay execution until the lock is released. A UniVerse BASIC program can impose several kinds of lock:

Type of Lock	Description
FILELOCK (FS, IX, FX)	Locks the entire table. A file lock can be a shared lock (which makes the table read-only to other users), an intent lock (which reserves the right to set an exclusive lock), or an exclusive lock (which excludes all other users from accessing the table in any manner).
READL (RL), READVL	Sets a shared lock on a row, letting other programs or users read but not update the row.
READU (RU), READVU	Sets an exclusive update lock on a row, preventing other programs or users from updating or locking the row.

BASIC Program Locks

For a full discussion of file and record locks, see *UniVerse BASIC*.

Transactions and Locks

Locks acquired before a transaction exists or outside an active transaction are inherited by the active transaction. Locks acquired or promoted within a transaction are not released. Instead they adhere to the following behavior:

- Locks acquired or promoted within a nested transaction are adopted by the parent transaction when the nested transaction commits.
- Locks acquired within a nested transaction are released when the nested transaction rolls back.
- Locks promoted within a nested transaction are demoted to the level they were before the start of that transaction when the nested transaction rolls back.
- All locks acquired, promoted, or adopted from nested transactions are released when the top-level transaction commits or rolls back.

Managing Deadlocks

Deadlocks occur when one of several processes acquiring locks incrementally tries to acquire a lock that another process owns, and the existing lock is incompatible with the requested lock.

You can configure UniVerse to automatically identify and resolve deadlocks as they occur, or you can manually fix a deadlock by selecting and aborting one of the deadlocked user processes. For more information about managing deadlocks, see *Administering UniVerse*.

Isolation Levels

Isolation levels separate one concurrent transaction from another so that their actions do not affect one another. Although the SQL standard recognizes only four isolation levels, UniVerse provides a fifth, adding isolation level 0 for backward compatibility. The default isolation level is 1.

UniVerse BASIC provides two ways to set isolation levels: the SET TRANSACTION ISOLATION LEVEL statement and the ISOLATION LEVEL clause of the BEGIN TRANSACTION statement. For more information about setting isolation levels, see *UniVerse BASIC*.

You can also use the SET.SQL command at the UniVerse prompt to set isolation levels.

To ensure successful operation, most isolation levels require that a program obtain a certain minimal lock on a table. In UniVerse SQL the SQL statements obtain the appropriate lock automatically, as shown in the following table:

When you use...	At this isolation level...	UniVerse SQL obtains this lock for you automatically...
SELECT	NO.ISOLATION (0)	None
	READ.UNCOMMITTED (1)	None
	READ.COMMITTED (2)	Shared record lock (RL)
	REPEATABLE.READ (3)	Shared record lock (RL)
	SERIALIZABLE (4)	Shared file lock (FS)

Locks Obtained by SQL Statements

When you use...	At this isolation level...	UniVerse SQL obtains this lock for you automatically...
DELETE, INSERT, or UPDATE	NO.ISOLATION (0)	Update record lock (RU)
	READ.UNCOMMITTED (1)	Update record lock (RU)
	READ.COMMITTED (2)	Update record lock (RU)
	REPEATABLE.READ (3)	Update record lock (RU)
	SERIALIZABLE (4)	Update record lock (RU) and intent file lock (IX)

Locks Obtained by SQL Statements (Continued)

Note: In the case of record locks, if an SQL statement needs more than MAXRLOCK (a system-configurable parameter) record locks, UniVerse obtains a file lock instead of adding more record locks.

For more information about locks and isolation levels and how to set them, see *UniVerse BASIC*.

Chapter

10

Transferring Tables Across Schemas

You can transfer SQL tables from one schema to another on the same system or from one system to another using the format conversion utility. The format conversion utility converts UniVerse database files, SQL tables, and UniVerse BASIC object code from one machine's storage format to another.

You can use the *format.conv* command at the UNIX shell prompt, or the FORMAT.CONV command in UniVerse.

You can use these commands to do the following:

- Prepare SQL tables for transfer to another schema on either the same or a different system
- Convert the storage format of SQL tables and reconstitute them in the schema to which they have been transferred

Moving SQL Tables

Use the format conversion utility when you want to move UniVerse SQL tables from one schema to another. The schemas can be on the same or on different machines. Converting tables is a bit more complex than converting regular UniVerse files. The process consists of the following steps:

1. Prepare the tables for transfer on the source machine, using the *–export* option of the format conversion utility.
2. Physically transfer the tables to an existing schema on the same or a different machine.
3. Evaluate the transferred data to prevent conflicts with existing data in the new schema.
4. Import the tables on the target machine using the *–import* option of the format conversion utility.
5. If desired, delete the tables from the source machine.

Preparing to Export SQL Tables

The format conversion utility with the *–export* option generates a set of files containing commands to be used to convert and reconstitute tables after you transfer them to another schema on the same or a different machine. It analyzes the SICAs of specified tables and generates a set of data definition language (DDL) commands that are used later to reconstitute the tables in the target schema. These commands are stored in a set of four text files whose default names are:

- EXPORTEDDDL.EXPORT
- EXPORTEDDDL.IMPORT
- EXPORTEDDDL.DROP
- EXPORTEDDDL.UNDO

To generate these files, use the following command at the UNIX shell prompt:

format.conv –export [**–name** *name*] [**–silent**] *tablepaths*

Or use the following command at the UniVerse prompt:

FORMAT.CONV –export [**–name** *name*] [**–silent**] *tablepaths*

You must be the owner of the specified tables or a DBA to use the *–export* option.

Use the *–name* option to specify a prefix other than EXPORTEDDDL for the names of the four text files. Use the *–silent* option to suppress system messages generated by the format conversion utility.

Use *tablepaths* to specify one or more tables to be exported. Specify each table by its relative or absolute path. Separate multiple table paths with spaces.

Specify tables with referential constraints first, before you specify the tables they reference. This ensures that the commands in the *name*.DROP and *name*.UNDO files are in the proper sequence.

Conversion File Formats

The *name*.EXPORT file looks something like this:

```
DISPLAY name.EXPORT created to hold DDL  09:19:51 04 APR 1997
* List of files that should be transferred.
* /usr/joan/uv/name.*
* /usr/joan/uv/table1
* /usr/joan/uv/D_table1
* /usr/joan/uv/table2
* /usr/joan/uv/D_table2
```

The format of the *name*.IMPORT file is as follows:

```
DISPLAY name.IMPORT created to hold DDL  09:19:51 04 APR 1997
* Command file to change copied files to Base Tables.
DISPLAY IMPORT/CREATE BASE TABLE section
* create VOC 'F' entry
INSERT INTO VOC (@ID, F1, F2, F3) VALUES ('table1', 'F Generated
by
   FORMAT.CONV -export utility', 'table1','D_table1');
* remove 'I' descriptor(s) for UNIQUE and INDEX names from DICT
DELETE DICT table1 UVCON_4
* import Base Table
FORMAT.CONV "D_table1"
FORMAT.CONV -convert "table1"
* raise to Base Table [DYNAMIC, SEQ NUM, MINIMUM MODULUS 2,GROUP
SIZE 2,
   SPLIT LOAD 75, MERGE LOAD 40,]
CREATE EXISTING TABLE "table1" . . .
* create INDEX(es)
* create TRIGGER(s)
* grant PERMISSION(s)
GRANT SELECT, INSERT, UPDATE on "table1" TO PUBLIC
GRANT SELECT, UPDATE("COL5") ON "table1" TO PUBLIC WITH GRANT
OPTION;
```

You can execute the FORMAT.CONV utility multiple times, using the same value of the *–name* option, to export a number of tables in the same group. Each execution of the FORMAT.CONV utility appends more information to the end of the four text files. For example, to create just one set of text files, including export information for tables TABLE1 through TABLE5, you could enter the following commands:

```
>FORMAT.CONV -export -name AA TABLE1 TABLE2
>FORMAT.CONV -export -name AA TABLE3
>FORMAT.CONV -export -name AA /usr/joe/TABLE4 /usr/joe/TABLE5
```

These commands create one set of text files, called AA.EXPORT, AA.IMPORT, AA.DROP and AA.UNDO, which include export information for tables TABLE1 through TABLE5.

After transferring this set of tables and text files to the target schema or system, you should delete the four text files, so that you can use the same –*name* value for another export session.

Physically Transferring Exported SQL Tables

When *format.conv –export* has finished, copy the following tables and files to either the export media or a target schema:

- All exported tables
- All exported table dictionaries
- Any trigger BASIC programs associated with the tables
- The four text files:
  - *name*.EXPORT
  - *name*.IMPORT
  - *name*.DROP
  - *name*.UNDO

***Note**: Be sure to copy the tables. Do not use operating system commands to move or delete them. To delete the tables after you copy them, use the format conversion utility with the –drop option.*

Use the list of tables in *name*.EXPORT file as a checklist of all the tables you need to transfer. For type 30 tables, be sure to transfer the directory and all subordinate files and directories.

If the target schema (the one to which you are transferring your tables and files) is new, transfer the tables directly into the schema. If the target schema already exists and contains tables, you should transfer your tables and files to a temporary directory. This lets you verify that nothing in the transferred tables or files conflicts with anything in the target schema.

Resolving Conflicts in the New Schema

Once you transfer your tables and files to the temporary directory, you can edit the *name*.IMPORT file to resolve any conflicts that may occur. For example, the names of the VOC file pointers to be created should not already exist in the schema's VOC file, nor should the names of the tables to be reconstituted be the same as any existing tables in the target schema.

The following cases require some kind of administrative intervention:

- If user names from the source machine do not exist on the target machine, edit the user names specified in the `* grant PERMISSION(s)` subsection under each base table.
- You may not want to create certain indexes, or you may want to activate or deactivate certain indexes. Edit the `* create INDEX(es)` subsection under each base table.
- Some trigger names may conflict with other program names on the target machine. Edit the `* create TRIGGER(s)` subsection under each base table.
- If you are transferring tables to a schema on the same machine, remove the CATALOG command from the `* create TRIGGER(s)` subsection.
- If any foreign key clauses reference unexported or nonexistent tables, remove the ALTER TABLE … ADD FOREIGN KEY command from the FOREIGN KEY subsection.
- If a table has a referential constraint to another table's constraint with multiple columns, you must include the names of the columns in the foreign key section of the import file.
- If any table names conflict with the names of existing tables, edit the names of the tables and their dictionaries in the *name*.IMPORT file to make them unique.

Importing Transferred SQL Tables

After you finish editing the *name*.IMPORT file, import the transferred tables using the following command at the UNIX shell prompt:

format.conv –import [**–name** *name*]

Or use the following command at the UniVerse prompt:

FORMAT.CONV –import [**–name** *name*]

These commands execute the set of commands in the *name*.IMPORT file, including byte-order conversion, if necessary. The user who executes these commands becomes the owner of the imported tables.

If you are importing several tables, you may want to capture the screen output and save it to a file. In UniVerse you can use the COMO command; in UNIX, pipe both standard output and standard error to the *tee* command.

Errors in Importing

If something goes wrong during the import conversion process, you can either use specific commands to correct the problem, or use the format conversion utility with the *–undo* option:

format.conv –undo [**–name** *name*]

The *–undo* option executes the set of commands in the *name*.UNDO file. These commands turn imported SQL tables back into UniVerse files, removing their SICAs and any SQL indexes. The format of the *name*.UNDO file is as follows:

```
DISPLAY name.UNDO created to hold DDL  09:19:51 04 APR 1997
* Execute this file to UNDO the effect of the import command file.
CREATE EXISTING TABLE "table1" RESTORE;
DELETE FROM VOC "table1";
CREATE EXISTING TABLE "table2" RESTORE;
DELETE FROM VOC "table2";
```

Deleting Exported Tables from the Old Schema

If you want to delete the exported tables from the source schema after transferring them to their destination, use the format conversion utility with the *–drop* option:

format.conv –drop [**–name** *name*]

The *–drop* option executes the commands contained in the *name*.DROP file. The format of this file is as follows:

```
DISPLAY name.DROP created to hold DDL  09:19:51 04 APR 1997
* Execute this file to DROP all exported Base Tables.
DROP TABLE "table1" CASCADE;
DROP TABLE "table2" CASCADE;
```

Chapter

11

Creating an XML Document with UniVerse SQL

XML for IBM UniVerse

The Extensible Markup Language (XML) is a markup language used to define, validate, and share document formats. It enables you to tailor document formats to specifications unique to your application by defining your own elements, tags, and attributes.

***Note**: XML describes how a document is structured, not how a document is displayed.*

XML was developed by the World Wide Web Consortium (W3C), who describe XML as:

> The Extensible Markup Language (XML) is the universal format for structured documents and data on the Web.

An XML document consists of a set of tags that describe the structure of data. Unlike HTML, you can write your own tags. You can use XML to describe any type of data so that it is cross-platform and machine independent.

For detailed information about XML, see the W3C website at http://www.w3.org/TR/REC-xml.

Document Type Definitions

You must define the rules of the structure of your XML document. These rules may be part of the XML document, and are called the Document Type Definition, or DTD. The DTD provides a list of elements, tags, attributes, and entities contained in the document, and describes their relationship to each other.

A DTD can be external or internal.

- External DTD — An external DTD is a separate document from the XML document, residing outside of your XML document. External DTDs can be applied to many different XML documents. If you need to change the DTD, you can make the change once, and all referencing XML documents are updated automatically.
- Internal DTD — An internal DTD resides in the XML document as part of the header of the document, and applies only to that XML document.

You can combine external DTDs with internal DTDs in an XML document.

The Document Object Model (DOM)

The Document Object Model (DOM) is a platform- and language-independent interface that enables programs and scripts to dynamically access and update the content, structure, and style of documents. A DOM is a formal way to describe an XML document to another application or programming language. You can describe the XML document as a tree, with nodes representing elements, attributes, entities, an text.

Well-Formed and Valid XML Documents

An XML document is either well-formed or valid:

- Well-formed XML documents must follow XML rules. All XML documents must be well-formed.
- Valid XML documents are both well-formed, and follow the rules of a specific DTD. Not all XML documents must be valid.

For optimum exchange of data, you should try to ensure that your XML documents are valid.

Creating an XML Document from RetrieVe

You can create an XML document from UniVerse files through RetrieVe. To create an XML document through RetrieVe, complete the following steps:

1. Analyze the DTD associated with the application to which you are sending the XML file. Determine which of your dictionary attributes correspond to the DTD elements.
2. Create an XML mapping file, if necessary.
3. List the appropriate fields using the LIST command.

Create the &XML& File

UniVerse stores XML mapping files in the &XML& directory file. To create this file, enter the following command:

```
CREATE.FILE &XML& 19
```

Mapping Modes

UniVerse supports four modes for mapping data to XML files. These modes are:

- Attribute-centric
- Element-centric
- Mixed
- Match-Element

Attribute-centric Mode

In the attribute-centric mode, which is the default mode, each record displayed in the query statement becomes an XML element. The following rules apply to the record fields:

- Each singlevalued field becomes an attribute within the element.
- Each multivalued or multi-subvalued field becomes a sub-element of the record element. The name of the sub-element is *association_name*-MV.

- Within a sub-element, each multivalued field becomes an attribute of the sub-element.
  - Associated multi-subvalued fields become another nested sub-element of the sub-element. The name of this nested sub-element is *association_name*-MS.
  - If there are no associated multi-subvalued fields, the sub-element name is field_name-MV/MS.

This is the default mapping scheme. You can change the default by defining maps in the &XML& file.

Element-centric Mode

In the element-centric mode, as in the attribute-centric mode, each record becomes an XML element. The following rules apply:

- Each singlevalued field becomes a simple sub-element of the element, containing no nested sub-elements. The value of the field becomes the value of the sub-element.
- Each association whose multivalued and multi-subvalued fields are included in the query statement form a complex sub-element. In the sub-element, each multivalued field belonging to the association becomes a sub-element that may contain multi-subvalued sub-elements. There are two ways to display empty values in multivalued fields belonging to an association. For detailed information, see Displaying Empty Values in Multivalued Fields in An Assocation.
- By default, UniVerse converts text marks to an empty string.

Specify that you want to use element-centric mapping by using the ELEMENTS keyword in the RetrieVe statement.

Displaying Empty Values in Multivalued Fields in An Assocation

How UniVerse displays empty values in multivalued fields belonging to an association is dependent on the setting of the Matchelement field in the U2XMLOUT.map file, located in the $UVHOME/&XML& directory.

If Matchelement is set to 1 (the default), matching values or subvalues belonging to the same association display as empty elements for matching pairs.

Consider the following example:

```
LIST XSTUDENT NAME CGA 02:22:19pm  14 Sep 2006  PAGE    1
XSTUDENT..    Name.........    Semester..    Courser NO    GRADE

521814564     Smith            FA93          CS130         A
                                             CS100         B
                                             PY100         B
                                SP94          CS131         B
                                                           B
                                             PE220         A
```

Notice that the second value of the COURSE NO field is empty, but the associated value for GRADE is not. When Matchelement is set to 1, the second value for COURSE NO displays as an empty value in the XML document, as shown in the following example:

>LIST XSTUDENT LNAME CGA TOXML ELEMENTS

```
<?xml version="1.0" encoding="UTF-8"?>
<ROOT>
<XSTUDENT>
  <_ID>521814564</_ID>
  <LNAME>Smith</LNAME>
  <CGA-MV>
    <SEMESTER>FA93</SEMESTER>
    <COURSE_NBR>CS130</COURSE_NBR>
    <COURSE_GRD>A</COURSE_GRD>
    <SEMESTER/>
    <COURSE_NBR>CS100</COURSE_NBR>
    <COURSE_GRD>B</COURSE_GRD>
    <SEMESTER/>
    <COURSE_NBR>PY100</COURSE_NBR>
    <COURSE_GRD>B</COURSE_GRD>
  </CGA-MV>
  <CGA-MV>
    <SEMESTER>SP94</SEMESTER>
    <COURSE_NBR>CS131</COURSE_NBR>
    <COURSE_GRD>B</COURSE_GRD>
<SEMESTER/>
    <COURSE_NBR/>                          <-- Empty Value
    <COURSE_GRD>B</COURSE_GRD>
    <SEMESTER/>
    <COURSE_NBR>PE220</COURSE_NBR>
    <COURSE_GRD>A</COURSE_GRD>
  </CGA-MV>
</XSTUDENT>
</ROOT>
>
```

This is the default behavior.

When Matchelement is set to 0, the second value for COURSE NO is ignored in the XML document, as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<ROOT>
<XSTUDENT>
  <_ID>521814564</_ID>
  <LNAME>Smith</LNAME>
  <CGA-MV>
    <SEMESTER>FA93</SEMESTER>
    <COURSE_NBR>CS130</COURSE_NBR>
    <COURSE_GRD>A</COURSE_GRD>
    <COURSE_NBR>CS100</COURSE_NBR>
    <COURSE_GRD>B</COURSE_GRD>
    <COURSE_NBR>PY100</COURSE_NBR>
    <COURSE_GRD>B</COURSE_GRD>
  </CGA-MV>
  <CGA-MV>
    <SEMESTER>SP94</SEMESTER>
    <COURSE_NBR>CS131</COURSE_NBR>
    <COURSE_GRD>B</COURSE_GRD>                    <-- Missing Value
    <COURSE_GRD>B</COURSE_GRD>
    <COURSE_NBR>PE220</COURSE_NBR>
    <COURSE_GRD>A</COURSE_GRD>
  </CGA-MV>
</XSTUDENT>
</ROOT>
```

Mixed Mode

In the mixed-mode, you create your own map file, where you specify which fields are treated as attribute-centric and which fields are treated as element-centric.

Field-level mapping overrides the mode you specify in the RetrieVe.

Creating a Mapping File

You can create the U2XMLOUT.map file in $UVHOME/&XML& to define commonly used global settings for creating XML documents. UniVerse reads and processes this mapping file each time UniVerse is started. For example, if you normally create element-centric output, and display empty elements for missing values or subvalues belonging to the same association, you can define these settings in the U2XMLOUT.map file, as shown in the following example:

```
<U2
  matchelement = "1"
  treated-as = "element"
/>
```

Defining these settings in the mapping file eliminates the need to specify them in each RetrieVe statement.

UniVerse processes XML options as follows:

1. Reads options defined in the U2XMLOUT.map file when UniVerse starts.
2. Reads any options defined in a mapping file. This mapping file resides in the &XML& directory in the current account, and is specified in the RetrieVe statement, as shown in the following example:

```
LIST STUDENT SEMESTER TOXML XMLMAPPING mystudent.map
```

3. Processes any options you specify in the RetrieVe statement.

Options you specify in the RetrieVe statement override options defined in the mapping file. Options defined in the mapping file override options defined in the U2XMLOUT.map file.

Warning: The attribute type definition for a multivalued or multi-subvalued field in a UniVerse dictionary record is "M." If you know that a field is multi-subvalued, you must define it as such in a mapping file, or the XML document you create may not be accurate.

A mapping file has the following format:

```
<?XML version="1.0"?>
<U2xml-mapping xmlns:U2xml="http://www.informix.com/U2-xml">
<!--there can be multiple <U2xml:mapping> elements -->
  <U2xml :mapping file="file_name"
                    field="dictionary_display_name"
                    map-to="name_in_xml_doc"
                    namespace="namespace_name"
                    type="MV" | "MS"
                    hastm="yes" | "1"
                    treated-as="attribute" | "element"
                    root="root_element_name"
                    record="record_element_name"
                    association-mv="mv_level_assoc_name"
                    association-ms="ms_level_assoc_name"
                    matchelement="0"|"1"
                    format = "format -pattern"..
                    conversion = "conversion code"
                    encode="encoding characters"
/>
```

...

</U2xml-mapping>

The following table describes each field of the mapping file.

Field	Description
XML version	The XML version number.
U2xml-mapping	The element name of a mapping rule.
file	The name of the UniVerse data file from which you are creating the XML document.
field	The display name of the dictionary record from which you are creating an XML element.
map-to	The name of the field to create in the XML document.
namespace	The name of the namespace. A namespace is a unique identifier that links an XML markup element to a specific DTD. They indicate to the processing application which DTD you are using.
type	The type of attribute. Value can be MV (multivalued), MS (multi-subvalued), TM (text mark), or any other mark contained in your data. The default is S (singlevalued).
hastm	Specifies whether or not to treat a text mark as another level of an element. A value of yes treats a text mark as another level of an element, while a value of 1 does not. This field applies only to element-centric mode.
treated-as	Specifies how to treat the XML element. Value is either attribute or element.
root	Root element name.
record	The name of the resulting XML record element.
association-mv	The name of the resulting XML multivalued association.
association-ms	The name of the resulting XML multi-subvalued association.

Mapping File Fields

Field	Description
format	Specifies the format to use in the output. See "Formatting Considerations" for more information.
conversion	Specifies the conversion code to use in the output. See "Conversion Code Considerations" for more information about how UniVerse XML processes conversion codes.
encode	Encoding characters in hexadecimal format, separated by spaces. See "Mapping File Encoding" for more information about how UniVerse XML handles encoding.

Mapping File Fields (Continued)

Conversion Code Considerations

UniVerse XML follows the following rules when extracting data from database files:

- If the dictionary record of a field you are extracting contains a conversion code, UniVerse XML uses that conversion code when extracting data from database files.
- If you specify a conversion code in the mapping file, the conversion code in the mapping file overrides the conversion code specified in the dictionary record.
- If you specify a conversion code using the CONV keyword during the execution of a RetrieVe statement, that conversion code overrides both the conversion code specified in the mapping file, and the conversion code specified in the dictionary record.

Formatting Considerations

UniVerse XML does not generally apply the dictionary format pattern to the extracted data. To specify a format, define it in the mapping file. If you specify a format using the FMT keyword in a RetrieVe statement, that format will override the format defined in the mapping file.

Mapping File Encoding

For special character encountered in data, UniVerse XML uses the default XML entities to encode the data. For example, ‘<‘ becomes <, ‘>’ becomes >, ‘&’ becomes &, and “ becomes ". However, UniVerse XML does not convert ‘ to ', unless you specify it in attribute encode. (<, >, &, ', and " are all built-in entities for the XML parser).

Use the encode field in the mapping file to add flexibility to the output. You can define special characters to encode in hexadecimal form. UniVerse encodes these special characters to &#x##;. For example, if you want the character ‘{‘ to be encoded for field FIELD1, specify the following encode value in the mapping file for FIELD1:

```
encode=”0x7B”
```

In this case, UniVerse XML will convert ‘{‘ found in the data of FIELD1 to &#x7B.

You can also use this type of encoding for any nonprintable character. If you need to define more than one character for a field, add a space between the hexadecimal definitions. For example, if you want to encode both ‘{‘ and ‘}’, the encode value in the mapping file should look like the following example:

```
encode=”0x7B 0x7D”
```

The following example illustrates a mapping file for the STUDENT file.

```
<?XML version="1.0"?>
<U2xml-mapping xmlns:U2xml="http://www.informix.com/U2-xml">
<! -- this is a comment -->
<U2xml:mapping record="STUDENT_rec"
   root = "SCHOOL"
   xmlns:IBM="http://www.IBM.com"
   namespace="IBM"
/>
<U2XML:mapping file="STUDENT"
      field = "SEMESTER"
      type="MV"
   treated-as="element"
/>
<U2XML:mapping file="STUDENT"
      field = "COURSE_NBR"
      type="MS"
   treated-as=element"
/>
<U2XML:mapping file="STUDENT"
      field = "COURSE_GRD"
      type="MS"
   treated-as="element"
/>
<U2XML:mapping file="STUDENT"
      field = "COURSE_NAME"
      type="MS"
   treated-as="element"
/>
</U2xml-mapping>
```

Notice that the SEMESTER, COURSE_NBR, COURSE_GRD, and COURSE_NAME fields are to be treated as elements. When you create the XML document, these fields will produce element-centric XML data. Any other fields listed in the query statement will produce attribute-centric XML data, since attribute-centric is the default mode.

Additionally, COURSE_NBR, COURSE_GRD, and COURSE_NAME are defined as multi-subvalued fields. If they were not, UniVerse would create the XML data as if they were multivalued attributes.

How Data is Mapped

Regardless of the mapping mode you choose, the outer-most element in the XML document is created as <ROOT>, by default. The name of each record element defaults to <*file_name*>.

You can change these mapping defaults in the mapping file, as shown in the following example:

```
<U2xml:mapping root="root_name"
               record="record_name"/>
```

Mapping Example

The following examples illustrate creation of XML documents. These examples use the STUDENT file, which contains the following fields:

```
>LIST DICT STUDENT

DICT STUDENT    12:50:05pm  10 Oct 2001  Page    1

              Type &
Field......... Field. Field........ Conversion.. Column......... Output
Depth &
Name.......... Number Definition... Code........ Heading........ Format
Assoc..

@ID             D    0                          STUDENT         10L    S
ID              D    0                          STUDENT         12R### S
                                                                -##-##
                                                                ##
LNAME           D    1                          Last Name       15T    S
FNAME           D    2                          First Name      10L    S
MAJOR           D    3                          Major           4L     S
MINOR           D    4                          Minor           4L     S
ADVISOR         D    5                          Advisor         8L     S
SEMESTER        D    6                          Term            4L     M
CGA
COURSE_NBR      D    7                          Crs #           5L     M
CGA
COURSE_GRD      D    8                          GD              3L     M
CGA
TEACHER         I      TRANS('COURSE            Teacher         10L    M
CGA
                       S',COURSE_NBR
                       ,'TEACHER','X
                       ')
COURSE_NAME     I      TRANS('COURSE            Course Name     25L    M
CGA
Press any key to continue...

DICT STUDENT    12:50:07pm  10 Oct 2001  Page    2

              Type &
Field......... Field. Field........ Conversion.. Column......... Output
Depth &
Name.......... Number Definition... Code........ Heading........ Format
Assoc..

                       S',COURSE_NBR
                       ,'NAME','X')
GPA1            I      SUBR('GPA1',C MD3        GPA             5R     S
                       OURSE_HOURS,C
                       OURSE_GRD)
COURSE_HOURS    I      TRANS('COURSE            Hours           5R     M
CGA
                       S',COURSE_NBR
                       ,CREDITS,'X')
CGA             PH     SEMESTER
                       COURSE_NBR
                       COURSE_NAME
```

```
                    COURSE_GRD
                    COURSE_HOURS
                    TEACHER
@ORIGINAL      S  @ID                                                    M
@SYNONYM       S   ID                                                    M

Press any key to continue...

DICT STUDENT    12:50:08pm  10 Oct 2001  Page    3

               Type &
Field......... Field. Field........ Conversion.. Column......... Output
Depth &
Name.......... Number Definition... Code........ Heading........ Format
Assoc..

17 records listed.
>
```

Creating an XML Document

To create an XML document using RetrieVe, use the LIST command.

> LIST [DICT | USING [DICT] *dictname*] *filename* ... [TOXML [ELEMENTS] [WITHDTD] [XMLMAPPING ***mapping_file***]]...

The following table describes each parameter of the syntax.

Parameter	Description
DICT	Lists records in the file dictionary of *filename*. If you do not specify DICT, records in the data file are listed.
USING [DICT] *dictname*	If DICT is not specified, uses *dictname* as the dictionary of *filename*. If DICT is specified, the dictionary of *dictname* is used as the dictionary of *filename*.
filename	The file whose records you want to list. You can specify *filename* anywhere in the sentence. LIST uses the first word in the sentence that has a file descriptor in the VOC file as the filename.
TOXML	Outputs LIST results in XML format.

LIST Parameters

Parameter	Description
ELEMENTS	Outputs results in element-centric format. I
WITHDTD	Output produces a DTD corresponding to the query.
XMLMAPPING *mapping_file*	Specifies a mapping file containing transformation rules for display. This file must exist in the &XML& file.

LIST Parameters (Continued)

For detailed information about the LIST command, see the *RetrieVe Users Guide*.

Examples

Note: The examples in this section use the STUDENT.F and COURSES files. To create these files, execute the MAKE.DEMO.FILES from the TCL prompt.

Creating an Attribute-centric XML Document

This example lists fields from the STUDENT file, using the TOXML keyword, to create an XML document. By default, UniVerse uses the attribute-centric mapping mode.

***Note**: UniVerse does not store the XML document unless you execute the COMO ON statement prior to executing the RetrieVe statement. If you execute COMO ON, the XML document is stored in the &COMO& file. You can also direct the output to the &HOLD& file using SETPTR, or the printer using LPTR.*

```
>LIST STUDENT.F SEMESTER COURSE_NBR COURSE_GRD COURSE_NAME TOXML
Validate XML name changed display name from '@ID' to '_ID'



<?xml version="1.0"?>
<ROOT>
<STUDENT.F_record _ID = "424325656">
  <CGA-MV SEMESTER = "SP94" COURSE_NBR = "PY100" COURSE_GRD = "C"
COURSE_NAME =
"Introduction to Psychology" COURSE_NBR = "PE100" COURSE_GRD = "C"
COURSE_NAME =
 "Golf - I"/>
</STUDENT.F_record>
<STUDENT.F_record _ID = "521814564">
  <CGA-MV SEMESTER = "FA93" COURSE_NBR = "CS130" COURSE_GRD = "A"
COURSE_NAME =
"Intro to Operating Systems" COURSE_NBR = "CS100" COURSE_GRD = "B"
COURSE_NAME =
 "Intro to Computer Science" COURSE_NBR = "PY100" COURSE_GRD = "B"
COURSE_NAME =
 "Introduction to Psychology"/>
  <CGA-MV SEMESTER = "SP94" COURSE_NBR = "CS131" COURSE_GRD = "B"
COURSE_NAME =
"Intro to Operating Systems" COURSE_NBR = "CS101" COURSE_GRD = "B"
COURSE_NAME =
 "Intro to Computer Science" COURSE_NBR = "PE220" COURSE_GRD = "A"
COURSE_NAME =
 "Racquetball"/>
</STUDENT.F_record>
<STUDENT.F_record _ID = "978766676">
  <CGA-MV SEMESTER = "FA93" COURSE_NBR = "FA120" COURSE_GRD = "A"
COURSE_NAME =
"Finger Painting" COURSE_NBR = "FA230" COURSE_GRD = "C"
COURSE_NAME = "Photograp
hy Principals" COURSE_NBR = "HY101" COURSE_GRD = "C" COURSE_NAME =
"Western Civi
lization"/>
  <CGA-MV SEMESTER = "SP94" COURSE_NBR = "FA121" COURSE_GRD = "A"
COURSE_NAME =
"Watercorlors" COURSE_NBR = "FA231" COURSE_GRD = "B" COURSE_NAME =
"Photography
Practicum" COURSE_NBR = "HY102" COURSE_GRD = "I" COURSE_NAME =
"Western Civiliza
tion - 1500 to 1945"/>
</STUDENT.F_record>
<STUDENT.F_record _ID = "221345665">
  <CGA-MV SEMESTER = "FA93" COURSE_NBR = "EG110" COURSE_GRD = "C"
```

```
COURSE_NAME =
"Engineering Principles" COURSE_NBR = "MA220" COURSE_GRD = "B"
COURSE_NAME = "Ca
lculus- I" COURSE_NBR = "PY100" COURSE_GRD = "B" COURSE_NAME =
"Introduction to
Psychology"/>
  <CGA-MV SEMESTER = "SP94" COURSE_NBR = "EG140" COURSE_GRD = "B"
COURSE_NAME =
"Fluid Mechanics" COURSE_NBR = "EG240" COURSE_GRD = "B"
COURSE_NAME = "Circut Th
eory" COURSE_NBR = "MA221" COURSE_GRD = "B" COURSE_NAME =
"Calculus - II"/>
</STUDENT.F_record>
<STUDENT.F_record _ID = "291222021">
  <CGA-MV SEMESTER = "SP94" COURSE_NBR = "FA100" COURSE_GRD = "B"
COURSE_NAME =
"Visual Thinking"/>
</STUDENT.F_record>
<STUDENT.F_record _ID = "414446545">
  <CGA-MV SEMESTER = "FA93" COURSE_NBR = "CS104" COURSE_GRD = "D"
COURSE_NAME =
"Database Design" COURSE_NBR = "MA101" COURSE_GRD = "C"
COURSE_NAME = "Math Prin
cipals" COURSE_NBR = "FA100" COURSE_GRD = "C" COURSE_NAME =
"Visual Thinking"/>
  <CGA-MV SEMESTER = "SP94" COURSE_NBR = "CS105" COURSE_GRD = "B"
COURSE_NAME =
"Database Design" COURSE_NBR = "MA102" COURSE_GRD = "C"
COURSE_NAME = "Algebra"
COURSE_NBR = "PY100" COURSE_GRD = "C" COURSE_NAME = "Introduction
to Psychology"
/>
</STUDENT.F_record>
</ROOT>
>
```

Creating an Element-centric XML Document

To create an element-centric XML document, use the ELEMENTS keyword.

```
>LIST STUDENT.F SEMESTER COURSE_NBR COURSE_GRD COURSE_NAME TOXML
ELEMENTS
Validate XML name changed display name from '@ID' to '_ID'



<?xml version="1.0"?>
<ROOT>
<STUDENT.F_record _ID = "424325656">
  <CGA-MV SEMESTER = "SP94" COURSE_NBR = "PY100" COURSE_GRD = "C"
COURSE_NAME =
"Introduction to Psychology" COURSE_NBR = "PE100" COURSE_GRD = "C"
COURSE_NAME =
 "Golf - I"/>
</STUDENT.F_record>
<STUDENT.F_record _ID = "521814564">
  <CGA-MV SEMESTER = "FA93" COURSE_NBR = "CS130" COURSE_GRD = "A"
COURSE_NAME =
"Intro to Operating Systems" COURSE_NBR = "CS100" COURSE_GRD = "B"
COURSE_NAME =
 "Intro to Computer Science" COURSE_NBR = "PY100" COURSE_GRD = "B"
COURSE_NAME =
 "Introduction to Psychology"/>
  <CGA-MV SEMESTER = "SP94" COURSE_NBR = "CS131" COURSE_GRD = "B"
COURSE_NAME =
"Intro to Operating Systems" COURSE_NBR = "CS101" COURSE_GRD = "B"
COURSE_NAME =
 "Intro to Computer Science" COURSE_NBR = "PE220" COURSE_GRD = "A"
COURSE_NAME =
 "Racquetball"/>
</STUDENT.F_record>
<STUDENT.F_record _ID = "978766676">
  <CGA-MV SEMESTER = "FA93" COURSE_NBR = "FA120" COURSE_GRD = "A"
COURSE_NAME =
"Finger Painting" COURSE_NBR = "FA230" COURSE_GRD = "C"
COURSE_NAME = "Photograp
hy Principals" COURSE_NBR = "HY101" COURSE_GRD = "C" COURSE_NAME =
"Western Civi
lization"/>
  <CGA-MV SEMESTER = "SP94" COURSE_NBR = "FA121" COURSE_GRD = "A"
COURSE_NAME =
"Watercorlors" COURSE_NBR = "FA231" COURSE_GRD = "B" COURSE_NAME =
"Photography
Practicum" COURSE_NBR = "HY102" COURSE_GRD = "I" COURSE_NAME =
"Western Civiliza
tion - 1500 to 1945"/>
</STUDENT.F_record>
<STUDENT.F_record _ID = "221345665">
  <CGA-MV SEMESTER = "FA93" COURSE_NBR = "EG110" COURSE_GRD = "C"
COURSE_NAME =
```

```
"Engineering Principles" COURSE_NBR = "MA220" COURSE_GRD = "B"
COURSE_NAME = "Ca
lculus- I" COURSE_NBR = "PY100" COURSE_GRD = "B" COURSE_NAME =
"Introduction to
Psychology"/>
  <CGA-MV SEMESTER = "SP94" COURSE_NBR = "EG140" COURSE_GRD = "B"
COURSE_NAME =
"Fluid Mechanics" COURSE_NBR = "EG240" COURSE_GRD = "B"
COURSE_NAME = "Circut Th
eory" COURSE_NBR = "MA221" COURSE_GRD = "B" COURSE_NAME =
"Calculus - II"/>
</STUDENT.F_record>
<STUDENT.F_record _ID = "291222021">
  <CGA-MV SEMESTER = "SP94" COURSE_NBR = "FA100" COURSE_GRD = "B"
COURSE_NAME =
"Visual Thinking"/>
</STUDENT.F_record>
<STUDENT.F_record _ID = "414446545">
  <CGA-MV SEMESTER = "FA93" COURSE_NBR = "CS104" COURSE_GRD = "D"
COURSE_NAME =
"Database Design" COURSE_NBR = "MA101" COURSE_GRD = "C"
COURSE_NAME = "Math Prin
cipals" COURSE_NBR = "FA100" COURSE_GRD = "C" COURSE_NAME =
"Visual Thinking"/>
  <CGA-MV SEMESTER = "SP94" COURSE_NBR = "CS105" COURSE_GRD = "B"
COURSE_NAME =
"Database Design" COURSE_NBR = "MA102" COURSE_GRD = "C"
COURSE_NAME = "Algebra"
COURSE_NBR = "PY100" COURSE_GRD = "C" COURSE_NAME = "Introduction
to Psychology"
/>
</STUDENT.F_record>
</ROOT>
>
```

Creating a Mixed-mode XML Document

Using the mapping file described in "Creating a Mapping File," the following example creates a mixed-mode XML document. To use a mapping file, specify the XMLMAPPING keyword in the RetrieVe statement.

```
>LIST STUDENT.F LNAME FNAME SEMESTER COURSE_NBR COURSE_GRD
COURSE_NAME TOXML XMLMAPPING STUDENT_MAP
Validate XML name changed display name from '@ID' to '_ID'



<?xml version="1.0"?>
<SCHOOL
xmlns:IBM="HTTP://WWW.IBM.COM"
>
<IBM:STUDENT_REC _ID = "424325656" LNAME = "Martin" FNAME =
"Sally">
  <CGA-MV>
    <SEMESTER>SP94</SEMESTER>
    <CGA-MS>
      <COURSE_NBR>PY100</COURSE_NBR>
      <COURSE_GRD>C</COURSE_GRD>
      <COURSE_NAME>Introduction to Psychology</COURSE_NAME>
    </CGA-MS>
    <CGA-MS>
      <COURSE_NBR>PE100</COURSE_NBR>
      <COURSE_GRD>C</COURSE_GRD>
      <COURSE_NAME>Golf - I</COURSE_NAME>
    </CGA-MS>
  </CGA-MV>
</IBM:STUDENT_REC>
<IBM:STUDENT_REC _ID = "521814564" LNAME = "Smith" FNAME =
"Harry">
  <CGA-MV>
    <SEMESTER>FA93</SEMESTER>
    <CGA-MS>
      <COURSE_NBR>CS130</COURSE_NBR>
      <COURSE_GRD>A</COURSE_GRD>
      <COURSE_NAME>Intro to Operating Systems</COURSE_NAME>
    </CGA-MS>
    <CGA-MS>
      <COURSE_NBR>CS100</COURSE_NBR>
      <COURSE_GRD>B</COURSE_GRD>
      <COURSE_NAME>Intro to Computer Science</COURSE_NAME>
    </CGA-MS>
    <CGA-MS>
      <COURSE_NBR>PY100</COURSE_NBR>
      <COURSE_GRD>B</COURSE_GRD>
      <COURSE_NAME>Introduction to Psychology</COURSE_NAME>
    </CGA-MS>
  </CGA-MV>
  <CGA-MV>
```

```
      <SEMESTER>SP94</SEMESTER>
      <CGA-MS>
        <COURSE_NBR>CS131</COURSE_NBR>
        <COURSE_GRD>B</COURSE_GRD>
        <COURSE_NAME>Intro to Operating Systems</COURSE_NAME>
      </CGA-MS>
      <CGA-MS>
        <COURSE_NBR>CS101</COURSE_NBR>
        <COURSE_GRD>B</COURSE_GRD>
        <COURSE_NAME>Intro to Computer Science</COURSE_NAME>
      </CGA-MS>
      <CGA-MS>
        <COURSE_NBR>PE220</COURSE_NBR>
        <COURSE_GRD>A</COURSE_GRD>
        <COURSE_NAME>Racquetball</COURSE_NAME>
      </CGA-MS>
    </CGA-MV>
  </IBM:STUDENT_REC>
  <IBM:STUDENT_REC _ID = "978766676" LNAME = "Muller" FNAME =
  "Gerhardt">
    <CGA-MV>
      <SEMESTER>FA93</SEMESTER>
      <CGA-MS>
        <COURSE_NBR>FA120</COURSE_NBR>
        <COURSE_GRD>A</COURSE_GRD>
        <COURSE_NAME>Finger Painting</COURSE_NAME>
      </CGA-MS>
      <CGA-MS>
        <COURSE_NBR>FA230</COURSE_NBR>
        <COURSE_GRD>C</COURSE_GRD>
        <COURSE_NAME>Photography Principals</COURSE_NAME>
      </CGA-MS>
      <CGA-MS>
        <COURSE_NBR>HY101</COURSE_NBR>
        <COURSE_GRD>C</COURSE_GRD>
        <COURSE_NAME>Western Civilization</COURSE_NAME>
      </CGA-MS>
    </CGA-MV>
    <CGA-MV>
      <SEMESTER>SP94</SEMESTER>
      <CGA-MS>
        <COURSE_NBR>FA121</COURSE_NBR>
        <COURSE_GRD>A</COURSE_GRD>
        <COURSE_NAME>Watercorlors</COURSE_NAME>
      </CGA-MS>
      <CGA-MS>
        <COURSE_NBR>FA231</COURSE_NBR>
        <COURSE_GRD>B</COURSE_GRD>
        <COURSE_NAME>Photography Practicum</COURSE_NAME>
      </CGA-MS>
      <CGA-MS>
        <COURSE_NBR>HY102</COURSE_NBR>
        <COURSE_GRD>I</COURSE_GRD>
        <COURSE_NAME>Western Civilization - 1500 to
```

```
1945</COURSE_NAME>
    </CGA-MS>
  </CGA-MV>
</IBM:STUDENT_REC>
<IBM:STUDENT_REC _ID = "221345665" LNAME = "Miller" FNAME =
"Susan">
  <CGA-MV>
    <SEMESTER>FA93</SEMESTER>
    <CGA-MS>
      <COURSE_NBR>EG110</COURSE_NBR>
      <COURSE_GRD>C</COURSE_GRD>
      <COURSE_NAME>Engineering Principles</COURSE_NAME>
    </CGA-MS>
    <CGA-MS>
      <COURSE_NBR>MA220</COURSE_NBR>
      <COURSE_GRD>B</COURSE_GRD>
      <COURSE_NAME>Calculus- I</COURSE_NAME>
    </CGA-MS>
    <CGA-MS>
      <COURSE_NBR>PY100</COURSE_NBR>
      <COURSE_GRD>B</COURSE_GRD>
      <COURSE_NAME>Introduction to Psychology</COURSE_NAME>
    </CGA-MS>
  </CGA-MV>
  <CGA-MV>
    <SEMESTER>SP94</SEMESTER>
    <CGA-MS>
      <COURSE_NBR>EG140</COURSE_NBR>
      <COURSE_GRD>B</COURSE_GRD>
      <COURSE_NAME>Fluid Mechanics</COURSE_NAME>
    </CGA-MS>
    <CGA-MS>
      <COURSE_NBR>EG240</COURSE_NBR>
      <COURSE_GRD>B</COURSE_GRD>
      <COURSE_NAME>Circut Theory</COURSE_NAME>
    </CGA-MS>
    <CGA-MS>
      <COURSE_NBR>MA221</COURSE_NBR>
      <COURSE_GRD>B</COURSE_GRD>
      <COURSE_NAME>Calculus - II</COURSE_NAME>
    </CGA-MS>
  </CGA-MV>
</IBM:STUDENT_REC>
<IBM:STUDENT_REC _ID = "291222021" LNAME = "Smith" FNAME = "jojo">
  <CGA-MV>
    <SEMESTER>SP94</SEMESTER>
    <CGA-MS>
      <COURSE_NBR>FA100</COURSE_NBR>
      <COURSE_GRD>B</COURSE_GRD>
      <COURSE_NAME>Visual Thinking</COURSE_NAME>
    </CGA-MS>
  </CGA-MV>
</IBM:STUDENT_REC>
<IBM:STUDENT_REC _ID = "414446545" LNAME = "Offenbach" FNAME =
```

```
"Karl">
  <CGA-MV>
    <SEMESTER>FA93</SEMESTER>
    <CGA-MS>
      <COURSE_NBR>CS104</COURSE_NBR>
      <COURSE_GRD>D</COURSE_GRD>
      <COURSE_NAME>Database Design</COURSE_NAME>
    </CGA-MS>
    <CGA-MS>
      <COURSE_NBR>MA101</COURSE_NBR>
      <COURSE_GRD>C</COURSE_GRD>
      <COURSE_NAME>Math Principals</COURSE_NAME>
    </CGA-MS>
    <CGA-MS>
      <COURSE_NBR>FA100</COURSE_NBR>
      <COURSE_GRD>C</COURSE_GRD>
      <COURSE_NAME>Visual Thinking</COURSE_NAME>
    </CGA-MS>
  </CGA-MV>
  <CGA-MV>
    <SEMESTER>SP94</SEMESTER>
    <CGA-MS>
      <COURSE_NBR>CS105</COURSE_NBR>
      <COURSE_GRD>B</COURSE_GRD>
      <COURSE_NAME>Database Design</COURSE_NAME>
    </CGA-MS>
    <CGA-MS>
      <COURSE_NBR>MA102</COURSE_NBR>
      <COURSE_GRD>C</COURSE_GRD>
      <COURSE_NAME>Algebra</COURSE_NAME>
    </CGA-MS>
    <CGA-MS>
      <COURSE_NBR>PY100</COURSE_NBR>
      <COURSE_GRD>C</COURSE_GRD>
      <COURSE_NAME>Introduction to Psychology</COURSE_NAME>
    </CGA-MS>
  </CGA-MV>
</IBM:STUDENT_REC>
</SCHOOL>
>
```

Notice in the XML document that LNAME and FNAME are attribute-centric, and the rest of the elements are element-centric.

Creating an XML Document with a DTD

If you only include the TOXML keyword in the RetrieVe statement, the resulting XML document does not include the DTD. To create an XML document that includes a DTD, use the WITHDTD keyword.

```
LIST STUDENT.F SEMESTER COURSE_NBR COURSE_GRD COURSE_NAME TOXML
WITHDTD
Validate XML name changed display name from '@ID' to '_ID'



<?xml version="1.0"?>
<!DOCTYPE ROOT[
<!ELEMENT ROOT (STUDENT.F_record*)>
<!ELEMENT STUDENT.F_record ( CGA-MV* )>
<!ATTLIST STUDENT.F_record
        _ID CDATA #REQUIRED
>
<!ELEMENT CGA-MV  EMPTY>
<!ATTLIST CGA-MV
        SEMESTER CDATA #IMPLIED
        COURSE_NBR CDATA #IMPLIED
        COURSE_GRD CDATA #IMPLIED
        COURSE_NAME CDATA #IMPLIED
>
]>
<ROOT>
<STUDENT.F_record _ID = "424325656">
  <CGA-MV SEMESTER = "SP94" COURSE_NBR = "PY100" COURSE_GRD = "C"
COURSE_NAME =
"Introduction to Psychology" COURSE_NBR = "PE100" COURSE_GRD = "C"
COURSE_NAME =
 "Golf - I"/>
</STUDENT.F_record>
<STUDENT.F_record _ID = "521814564">
  <CGA-MV SEMESTER = "FA93" COURSE_NBR = "CS130" COURSE_GRD = "A"
COURSE_NAME =
"Intro to Operating Systems" COURSE_NBR = "CS100" COURSE_GRD = "B"
COURSE_NAME =
 "Intro to Computer Science" COURSE_NBR = "PY100" COURSE_GRD = "B"
COURSE_NAME =
 "Introduction to Psychology"/>
  <CGA-MV SEMESTER = "SP94" COURSE_NBR = "CS131" COURSE_GRD = "B"
COURSE_NAME =
"Intro to Operating Systems" COURSE_NBR = "CS101" COURSE_GRD = "B"
COURSE_NAME =
 "Intro to Computer Science" COURSE_NBR = "PE220" COURSE_GRD = "A"
COURSE_NAME =
 "Racquetball"/>
</STUDENT.F_record>
<STUDENT.F_record _ID = "978766676">
  <CGA-MV SEMESTER = "FA93" COURSE_NBR = "FA120" COURSE_GRD = "A"
```

```
COURSE_NAME =
"Finger Painting" COURSE_NBR = "FA230" COURSE_GRD = "C"
COURSE_NAME = "Photograp
hy Principals" COURSE_NBR = "HY101" COURSE_GRD = "C" COURSE_NAME =
"Western Civi
lization"/>
  <CGA-MV SEMESTER = "SP94" COURSE_NBR = "FA121" COURSE_GRD = "A"
COURSE_NAME =
"Watercorlors" COURSE_NBR = "FA231" COURSE_GRD = "B" COURSE_NAME =
"Photography
Practicum" COURSE_NBR = "HY102" COURSE_GRD = "I" COURSE_NAME =
"Western Civiliza
tion - 1500 to 1945"/>
</STUDENT.F_record>
<STUDENT.F_record _ID = "221345665">
  <CGA-MV SEMESTER = "FA93" COURSE_NBR = "EG110" COURSE_GRD = "C"
COURSE_NAME =
"Engineering Principles" COURSE_NBR = "MA220" COURSE_GRD = "B"
COURSE_NAME = "Ca
lculus- I" COURSE_NBR = "PY100" COURSE_GRD = "B" COURSE_NAME =
"Introduction to
Psychology"/>
  <CGA-MV SEMESTER = "SP94" COURSE_NBR = "EG140" COURSE_GRD = "B"
COURSE_NAME =
"Fluid Mechanics" COURSE_NBR = "EG240" COURSE_GRD = "B"
COURSE_NAME = "Circut Th
eory" COURSE_NBR = "MA221" COURSE_GRD = "B" COURSE_NAME =
"Calculus - II"/>
</STUDENT.F_record>
<STUDENT.F_record _ID = "291222021">
  <CGA-MV SEMESTER = "SP94" COURSE_NBR = "FA100" COURSE_GRD = "B"
COURSE_NAME =
"Visual Thinking"/>
</STUDENT.F_record>
<STUDENT.F_record _ID = "414446545">
  <CGA-MV SEMESTER = "FA93" COURSE_NBR = "CS104" COURSE_GRD = "D"
COURSE_NAME =
"Database Design" COURSE_NBR = "MA101" COURSE_GRD = "C"
COURSE_NAME = "Math Prin
cipals" COURSE_NBR = "FA100" COURSE_GRD = "C" COURSE_NAME =
"Visual Thinking"/>
  <CGA-MV SEMESTER = "SP94" COURSE_NBR = "CS105" COURSE_GRD = "B"
COURSE_NAME =
"Database Design" COURSE_NBR = "MA102" COURSE_GRD = "C"
COURSE_NAME = "Algebra"
COURSE_NBR = "PY100" COURSE_GRD = "C" COURSE_NAME = "Introduction
to Psychology"
/>
</STUDENT.F_record>
</ROOT>
>
```

Creating an XML Document with UniVerse SQL

In addition to RetrieVe, you can also create XML documents using UniVerse SQL. To create an XML document through UniVerse SQL, complete the following steps:

1. Analyze the DTD associated with the application to which you are sending the XML file. Determine which of your dictionary attributes correspond to the DTD elements.
2. Create an XML mapping file, if necessary.
3. List the appropriate fields using the UniVerse SQL SELECT command.

Create the &XML& File

UniVerse stores XML mapping files and XSL files in the &XML& directory file. To create this file, enter the following command:

```
CREATE.FILE &XML& 19
```

To create an XML document from UniVerse SQL, use the UniVerse SQL SELECT command.

```
SELECT clause FROM clause
        [WHERE clause]
        [WHEN clause [WHEN clause]...]
        [GROUP BY clause]
        [HAVING clause]
        [ORDER BY clause]
        [report_qualifiers]
        [processing_qualifiers]
        [TOXML [ELEMENTS] [WITHDTD]
                [XMLMAPPING mapping_file]]
        [XMLDATA extraction_mapping_file];
```

The following table describes each parameter of the syntax.

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.
HAVING clause	Specifies the criteria that grouped rows must meet to be selected.
ORDER BY clause	Sorts selected rows.
report_qualifiers	Formats a report generated by the SELECT statement.
processing_qualifiers	Modifies or reports on the processing of the SELECT statement.
TOXML	Outputs SELECT results in XML format.
ELEMENTS	Outputs results in element-centric format.
WITHDTD	Output produces a DTD corresponding to the query.
XMLMAPPING *'mapping_file'*	Specifies a mapping file containing transformation rules for display. This file must exist in the &XML& file.
XMLDATA *extraction_mapping_file*	Specifies the file containing the extraction rules for the XML document. This file is used for receiving an XML file.

SELECT Parameters

You must specify clauses in the SELECT statement in the order shown in the syntax. 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.

For a full discussion of the UniVerse SQL SELECT statement clauses, see the *UniVerse SQL Reference*.

Processing Rules for UniVerse SQL SELECT Statements

UniVerse processes SELECT statements much the same as it processes LIST statements, with a few exceptions.

The processing rules for a UniVerse SQL SELECT statement against a single table are the same as the RetrieVe LIST rules. For a discussion of how UniVerse SQL processes these statements, see "Creating an XML Document from RetrieVe."

Processing Multiple Tables

When processing a UniVerse SQL SELECT statement involving multiple files, UniVerse attempts to keep the nesting inherited in the query in the resulting XML document. Because of this, the order in which you specify the fields in the UniVerse SQL SELECT statement is important for determining how the elements are nested.

Processing in Attribute-centric Mode

As with RetrieVe, the attribute-centric mode is the default mapping mode. For more information about the attribute-centric mode, see "Attribute-centric Mode."

- In this mode, UniVerse uses the name of the file containing the first field you specify in the SELECT statement as the outer-most element in the XML output. Any singlevalued fields you specify in the SELECT statement that belong to this file become attributes of this element.
- UniVerse processes the SELECT statement in the order you specify. If it finds a field that belongs to another file, UniVerse creates a sub-element. The name of this sub-element is the new file name. All singlevalued fields found in the SELECT statement that belong to this file are created as attributes for the sub-element.
- If UniVerse finds a multivalued or multi-subvalued field in the SELECT statement, it creates a sub-element. The name of this element is the name of the association of which this field is a member.
- When you execute UNNEST against an SQL table, it flattens the multivalues into single values.

UniVerse processes the ELEMENTS, WITHDTD, and XMLMAPPING keywords in the same manner as it processes them for the RetrieVe LIST command.

Processing in Element-centric Mode

When using the element-centric mode, UniVerse automatically prefixes each file name to the association name. For example, the CGA association in the STUDENT file is named STUDENT_CGA in the resulting XML file.

XML Limitations in UniVerse SQL

The TOXML keyword is not allowed in the following cases:

- In a sub-query
- In a SELECT statement that is part of an INSERT statement.
- In a SELECT statement that is part of a UNION definition.
- In a SELECT statement that is part of a VIEW definition.

Examples

This section illustrates XML output from the UniVerse SQL SELECT statement. The examples use sample CUSTOMER, TAPES, and STUDENT files.

The following example lists the dictionary records from the CUSTOMER file that are used in the examples:

```
DICT CUSTOMER    04:31:35pm  11 Oct 2001  Page    1

                 Type &
Field......... Field. Field........ Conversion.. Column......... Output Depth &
Name.......... Number Definition... Code........ Heading........ Format Assoc..

NAME             D    1                          Customer Name   15T    S
TAPES_RENTED     D    7                          Tapes           10L    M TAPE_
                                                                        INFO
TAPE_INFO        PH     TAPES_RENTED
                        DATE_OUT
                        DATE_DUE
                        DAYS_BETWEEN
                        TAPE_COST
                        TAPE_NAME
                        UP_NAMES
                        TAPE_CAT

DICT TAPES    04:33:47pm  11 Oct 2001  Page    1

                 Type &
Field......... Field. Field........ Conversion.. Column......... Output Depth &
Name.......... Number Definition... Code........ Heading........ Format Assoc..

@ID              D    0                          TAPES           10L    S
NAME             D    1                          Tape Name       20T    S
CAT_NAME         I      TRANS('CATEGO            Tape Type       20L    S CATS
                        RIES',CATEGOR
```

```
                          IES,'NAME','X
                          ')

1 records listed.

>
```

Creating an XML Document From Multiple Files in Attribute-centric Mode

In the following example, UniVerse creates an XML document from the CUSTOMER.F and TAPES.F file in the attribute-centric mode.

```
>SELECT CUSTOMER.F.NAME, TAPES.F.NAME, CAT_NAME FROM
CUSTOMER.F,TAPES.F WHERETAPES_RENTED = TAPES.F.@ID ORDER BY
CUSTOMER.F.NAME TOXML;


<?xml version="1.0"?>
<ROOT>
<CUSTOMER.F_record NAME = "Barrie, Dick">
<TAPES.F NAME = "Citizen Kane">
  <TAPES.F_CATS_MV CAT_NAME = "Old Classic"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
</TAPES.F>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Best, George">
<TAPES.F NAME = "Love Story">
  <TAPES.F_CATS_MV CAT_NAME = "Romance"/>
  <TAPES.F_CATS_MV CAT_NAME = "Tear Jerker"/>
</TAPES.F>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Bowie, David">
<TAPES.F NAME = "The Stalker">
  <TAPES.F_CATS_MV CAT_NAME = "Avant Garde"/>
  <TAPES.F_CATS_MV CAT_NAME = "Science Fiction"/>
</TAPES.F>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Chase, Carl">
<TAPES.F NAME = "'Round Midnight">
  <TAPES.F_CATS_MV CAT_NAME = "Musical"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Chase, Carl">
<TAPES.F NAME = "American Graffiti ">
  <TAPES.F_CATS_MV CAT_NAME = "Comedy"/>
  <TAPES.F_CATS_MV CAT_NAME = "Childrens Movie"/>
</TAPES.F>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Chase, Carl">
<TAPES.F NAME = "Flash Gordon">
  <TAPES.F_CATS_MV CAT_NAME = "Science Fiction"/>
  <TAPES.F_CATS_MV CAT_NAME = "Childrens Movie"/>
</TAPES.F>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Faber, Harry">
<TAPES.F NAME = "To Kill A Mockingbird">
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
```

```
  <TAPES.F_CATS_MV CAT_NAME = "Political"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Fischer, Carrie">
<TAPES.F NAME = "Girl Friday">
  <TAPES.F_CATS_MV CAT_NAME = "Comedy"/>
  <TAPES.F_CATS_MV CAT_NAME = "Old Classic"/>
</TAPES.F>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "James, Bob">
<TAPES.F NAME = "Blue Velvet">
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
  <TAPES.F_CATS_MV CAT_NAME = "Avant Garde"/>
</TAPES.F>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jamieson, Dale">
<TAPES.F NAME = "2001">
  <TAPES.F_CATS_MV CAT_NAME = "Science Fiction"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Bob">
<TAPES.F NAME = "Z">
  <TAPES.F_CATS_MV CAT_NAME = "Political"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Freddie">
<TAPES.F NAME = "Help">
  <TAPES.F_CATS_MV CAT_NAME = "Childrens Movie"/>
  <TAPES.F_CATS_MV CAT_NAME = "Comedy"/>
  <TAPES.F_CATS_MV CAT_NAME = "Musical"/>
</TAPES.F>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Mable">
<TAPES.F NAME = "Psycho">
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Mable">
<TAPES.F NAME = "Gone With The Wind">
  <TAPES.F_CATS_MV CAT_NAME = "Romance"/>
</TAPES.F>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Samuel">
<TAPES.F NAME = "'Round Midnight">
  <TAPES.F_CATS_MV CAT_NAME = "Musical"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Samuel">
```

```
<TAPES.F NAME = "Flash Gordon">
  <TAPES.F_CATS_MV CAT_NAME = "Science Fiction"/>
  <TAPES.F_CATS_MV CAT_NAME = "Childrens Movie"/>
</TAPES.F>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Partner, Bonnie">
<TAPES.F NAME = "Tammy">
  <TAPES.F_CATS_MV CAT_NAME = "Romance"/>
</TAPES.F>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Partner, Bonnie">
<TAPES.F NAME = "Love Story">
  <TAPES.F_CATS_MV CAT_NAME = "Romance"/>
  <TAPES.F_CATS_MV CAT_NAME = "Tear Jerker"/>
</TAPES.F>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Smith, Harry">
<TAPES.F NAME = "Journey Abroad">
  <TAPES.F_CATS_MV CAT_NAME = "B - Movie"/>
</TAPES.F>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Smith, Harry">
<TAPES.F NAME = "Catch 22">
  <TAPES.F_CATS_MV CAT_NAME = "Comedy"/>
  <TAPES.F_CATS_MV CAT_NAME = "Avant Garde"/>
</TAPES.F>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Smith, Harry">
<TAPES.F NAME = "Blue Velvet">
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
  <TAPES.F_CATS_MV CAT_NAME = "Avant Garde"/>
</TAPES.F>
</CUSTOMER.F_record>
</ROOT>
>
```

The next example illustrates the results of a UniVerse SQL statement against the same fields with a different SELECT order and a different sorting option:

```
01 SELECT TAPES.F.NAME, CUSTOMER.F.NAME, CAT_NAME FROM CUSTOMER.F,
TAPES.F WHERE
 TAPES_RENTED = TAPES.F.@ID ORDER BY TAPES.F.NAME TOXML;


<?xml version="1.0"?>
<ROOT>
<TAPES.F_record NAME = "'Round Midnight">
<CUSTOMER.F NAME = "Jones, Samuel"/>
  <TAPES.F_CATS_MV CAT_NAME = "Musical"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F_record>
<TAPES.F_record NAME = "'Round Midnight">
<CUSTOMER.F NAME = "Chase, Carl"/>
  <TAPES.F_CATS_MV CAT_NAME = "Musical"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F_record>
<TAPES.F_record NAME = "2001">
<CUSTOMER.F NAME = "Jamieson, Dale"/>
  <TAPES.F_CATS_MV CAT_NAME = "Science Fiction"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F_record>
<TAPES.F_record NAME = "American Graffiti ">
<CUSTOMER.F NAME = "Chase, Carl"/>
  <TAPES.F_CATS_MV CAT_NAME = "Comedy"/>
  <TAPES.F_CATS_MV CAT_NAME = "Childrens Movie"/>
</TAPES.F_record>
<TAPES.F_record NAME = "Blue Velvet">
<CUSTOMER.F NAME = "Smith, Harry"/>
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
  <TAPES.F_CATS_MV CAT_NAME = "Avant Garde"/>
</TAPES.F_record>
<TAPES.F_record NAME = "Blue Velvet">
<CUSTOMER.F NAME = "James, Bob"/>
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
  <TAPES.F_CATS_MV CAT_NAME = "Avant Garde"/>
</TAPES.F_record>
<TAPES.F_record NAME = "Catch 22">
<CUSTOMER.F NAME = "Smith, Harry"/>
  <TAPES.F_CATS_MV CAT_NAME = "Comedy"/>
  <TAPES.F_CATS_MV CAT_NAME = "Avant Garde"/>
</TAPES.F_record>
<TAPES.F_record NAME = "Citizen Kane">
<CUSTOMER.F NAME = "Barrie, Dick"/>
  <TAPES.F_CATS_MV CAT_NAME = "Old Classic"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
</TAPES.F_record>
<TAPES.F_record NAME = "Flash Gordon">
```

```
<CUSTOMER.F NAME = "Chase, Carl"/>
  <TAPES.F_CATS_MV CAT_NAME = "Science Fiction"/>
  <TAPES.F_CATS_MV CAT_NAME = "Childrens Movie"/>
</TAPES.F_record>
<TAPES.F_record NAME = "Flash Gordon">
<CUSTOMER.F NAME = "Jones, Samuel"/>
  <TAPES.F_CATS_MV CAT_NAME = "Science Fiction"/>
  <TAPES.F_CATS_MV CAT_NAME = "Childrens Movie"/>
</TAPES.F_record>
<TAPES.F_record NAME = "Girl Friday">
<CUSTOMER.F NAME = "Fischer, Carrie"/>
  <TAPES.F_CATS_MV CAT_NAME = "Comedy"/>
  <TAPES.F_CATS_MV CAT_NAME = "Old Classic"/>
</TAPES.F_record>
<TAPES.F_record NAME = "Gone With The Wind">
<CUSTOMER.F NAME = "Jones, Mable"/>
  <TAPES.F_CATS_MV CAT_NAME = "Romance"/>
</TAPES.F_record>
<TAPES.F_record NAME = "Help">
<CUSTOMER.F NAME = "Jones, Freddie"/>
  <TAPES.F_CATS_MV CAT_NAME = "Childrens Movie"/>
  <TAPES.F_CATS_MV CAT_NAME = "Comedy"/>
  <TAPES.F_CATS_MV CAT_NAME = "Musical"/>
</TAPES.F_record>
<TAPES.F_record NAME = "Journey Abroad">
<CUSTOMER.F NAME = "Smith, Harry"/>
  <TAPES.F_CATS_MV CAT_NAME = "B - Movie"/>
</TAPES.F_record>
<TAPES.F_record NAME = "Love Story">
<CUSTOMER.F NAME = "Partner, Bonnie"/>
  <TAPES.F_CATS_MV CAT_NAME = "Romance"/>
  <TAPES.F_CATS_MV CAT_NAME = "Tear Jerker"/>
</TAPES.F_record>
<TAPES.F_record NAME = "Love Story">
<CUSTOMER.F NAME = "Best, George"/>
  <TAPES.F_CATS_MV CAT_NAME = "Romance"/>
  <TAPES.F_CATS_MV CAT_NAME = "Tear Jerker"/>
</TAPES.F_record>
<TAPES.F_record NAME = "Psycho">
<CUSTOMER.F NAME = "Jones, Mable"/>
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F_record>
<TAPES.F_record NAME = "Tammy">
<CUSTOMER.F NAME = "Partner, Bonnie"/>
  <TAPES.F_CATS_MV CAT_NAME = "Romance"/>
</TAPES.F_record>
<TAPES.F_record NAME = "The Stalker">
<CUSTOMER.F NAME = "Bowie, David"/>
  <TAPES.F_CATS_MV CAT_NAME = "Avant Garde"/>
  <TAPES.F_CATS_MV CAT_NAME = "Science Fiction"/>
</TAPES.F_record>
<TAPES.F_record NAME = "To Kill A Mockingbird">
<CUSTOMER.F NAME = "Faber, Harry"/>
```

```
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
  <TAPES.F_CATS_MV CAT_NAME = "Political"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F_record>
<TAPES.F_record NAME = "Z">
<CUSTOMER.F NAME = "Jones, Bob"/>
  <TAPES.F_CATS_MV CAT_NAME = "Political"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F_record>
</ROOT>
>
```

Creating an XML Document From Multiple Files in Element-centric Mode

The following example illustrates creating an XML document from multiple files in element-centric mode, using the ELEMENTS keyword.

```
>SELECT CUSTOMER.NAME, TAPES.NAME, CAT_NAME FROM CUSTOMER, TAPES
WHERE TAPES_RENTED = TAPES.@ID ORDER BY CUSTOMER.NAME TOXML
ELEMENTS;
Validate XML name changed display name from 'Customer Name' to
'Customer_Name'
Validate XML name changed display name from 'Tape Name' to
'Tape_Name'
Validate XML name changed display name from 'Tape Type' to
'Tape_Type'


<?xml version="1.0"?>
<ROOT>
<CUSTOMER_record>
  <Customer_Name>Chase, Carl</Customer_Name>
<TAPES>
  <Tape_Name>American Graffiti</Tape_Name>
  <TAPES_CATS_MV>
    <Tape_Type>Comedy</Tape_Type>
  </TAPES_CATS_MV>
  <TAPES_CATS_MV>
    <Tape_Type>Childrens Movie</Tape_Type>
  </TAPES_CATS_MV>
</TAPES>
</CUSTOMER_record>
<CUSTOMER_record>
  <Customer_Name>Chase, Carl</Customer_Name>
<TAPES>
  <Tape_Name>Flash Gordon</Tape_Name>
  <TAPES_CATS_MV>
    <Tape_Type>Science Fiction</Tape_Type>
  </TAPES_CATS_MV>
  <TAPES_CATS_MV>
    <Tape_Type>Childrens Movie</Tape_Type>
  </TAPES_CATS_MV>
</TAPES>
</CUSTOMER_record>
<CUSTOMER_record>
  <Customer_Name>Chase, Carl</Customer_Name>
<TAPES>
  <Tape_Name>'Round Midnight</Tape_Name>
  <TAPES_CATS_MV>
    <Tape_Type>Musical</Tape_Type>
  </TAPES_CATS_MV>
  <TAPES_CATS_MV>
    <Tape_Type>Drama</Tape_Type>
  </TAPES_CATS_MV>
```

```
</TAPES>
</CUSTOMER_record>
<CUSTOMER_record>
  <Customer_Name>Jamieson, Dale</Customer_Name>
<TAPES>
  <Tape_Name>2001</Tape_Name>
  <TAPES_CATS_MV>
    <Tape_Type>Science Fiction</Tape_Type>
  </TAPES_CATS_MV>
  <TAPES_CATS_MV>
    <Tape_Type>Drama</Tape_Type>
  </TAPES_CATS_MV>
</TAPES>
</CUSTOMER_record>
<CUSTOMER_record>
  <Customer_Name>Jones, Bob</Customer_Name>
<TAPES>
  <Tape_Name>Z</Tape_Name>
  <TAPES_CATS_MV>
    <Tape_Type>Political</Tape_Type>
  </TAPES_CATS_MV>
  <TAPES_CATS_MV>
    <Tape_Type>Drama</Tape_Type>
  </TAPES_CATS_MV>
</TAPES>
</CUSTOMER_record>
</ROOT>
>
```

Creating an XML Document From Multiple Files with a Multivalued Field

The next example illustrates creating an XML document from multiple files with a multivalued field. In the example, TAPES_RENTED is multivalued and belongs to the TAPE_INFO association in the CUSTOMER file. In the XML document, TAPES_RENTED appears in the CUSTOMER_TAPE_INFO_MV element.

```
>SELECT CUSTOMER.F.NAME, TAPES.F.NAME, CAT_NAME, TAPES_RENTED FROM
CUSTOMER.F, TAPES.F WHERE TAPES_RENTED = TAPES.F.@ID ORDER BY
CUSTOMER.F.NAME TOXML;


<?xml version="1.0"?>
<ROOT>
<CUSTOMER.F_record NAME = "Barrie, Dick">
<TAPES.F NAME = "Citizen Kane">
  <TAPES.F_CATS_MV CAT_NAME = "Old Classic"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V996"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Best, George">
<TAPES.F NAME = "Love Story">
  <TAPES.F_CATS_MV CAT_NAME = "Romance"/>
  <TAPES.F_CATS_MV CAT_NAME = "Tear Jerker"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "B2297"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Bowie, David">
<TAPES.F NAME = "The Stalker">
  <TAPES.F_CATS_MV CAT_NAME = "Avant Garde"/>
  <TAPES.F_CATS_MV CAT_NAME = "Science Fiction"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V9961"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Chase, Carl">
<TAPES.F NAME = "'Round Midnight">
  <TAPES.F_CATS_MV CAT_NAME = "Musical"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V8481"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V1254"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V4951"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Chase, Carl">
<TAPES.F NAME = "American Graffiti ">
  <TAPES.F_CATS_MV CAT_NAME = "Comedy"/>
  <TAPES.F_CATS_MV CAT_NAME = "Childrens Movie"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V8481"/>
```

```
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V1254"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V4951"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Chase, Carl">
<TAPES.F NAME = "Flash Gordon">
  <TAPES.F_CATS_MV CAT_NAME = "Science Fiction"/>
  <TAPES.F_CATS_MV CAT_NAME = "Childrens Movie"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V8481"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V1254"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V4951"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Faber, Harry">
<TAPES.F NAME = "To Kill A Mockingbird">
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
  <TAPES.F_CATS_MV CAT_NAME = "Political"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V5151"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Fischer, Carrie">
<TAPES.F NAME = "Girl Friday">
  <TAPES.F_CATS_MV CAT_NAME = "Comedy"/>
  <TAPES.F_CATS_MV CAT_NAME = "Old Classic"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V110"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "James, Bob">
<TAPES.F NAME = "Blue Velvet">
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
  <TAPES.F_CATS_MV CAT_NAME = "Avant Garde"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V2001"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jamieson, Dale">
<TAPES.F NAME = "2001">
  <TAPES.F_CATS_MV CAT_NAME = "Science Fiction"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V6670"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Bob">
<TAPES.F NAME = "Z">
  <TAPES.F_CATS_MV CAT_NAME = "Political"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V4341"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Freddie">
<TAPES.F NAME = "Help">
  <TAPES.F_CATS_MV CAT_NAME = "Childrens Movie"/>
  <TAPES.F_CATS_MV CAT_NAME = "Comedy"/>
  <TAPES.F_CATS_MV CAT_NAME = "Musical"/>
```

```
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V9431"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Mable">
<TAPES.F NAME = "Psycho">
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V1249"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V4499"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Mable">
<TAPES.F NAME = "Gone With The Wind">
  <TAPES.F_CATS_MV CAT_NAME = "Romance"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V1249"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V4499"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Samuel">
<TAPES.F NAME = "'Round Midnight">
  <TAPES.F_CATS_MV CAT_NAME = "Musical"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V1254"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V8481"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Samuel">
<TAPES.F NAME = "Flash Gordon">
  <TAPES.F_CATS_MV CAT_NAME = "Science Fiction"/>
  <TAPES.F_CATS_MV CAT_NAME = "Childrens Movie"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V1254"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V8481"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Partner, Bonnie">
<TAPES.F NAME = "Tammy">
  <TAPES.F_CATS_MV CAT_NAME = "Romance"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "B914"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "B2297"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Partner, Bonnie">
<TAPES.F NAME = "Love Story">
  <TAPES.F_CATS_MV CAT_NAME = "Romance"/>
  <TAPES.F_CATS_MV CAT_NAME = "Tear Jerker"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "B914"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "B2297"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Smith, Harry">
<TAPES.F NAME = "Journey Abroad">
  <TAPES.F_CATS_MV CAT_NAME = "B - Movie"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V2001"/>
```

```
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V5004"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V8181"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Smith, Harry">
<TAPES.F NAME = "Catch 22">
  <TAPES.F_CATS_MV CAT_NAME = "Comedy"/>
  <TAPES.F_CATS_MV CAT_NAME = "Avant Garde"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V2001"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V5004"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V8181"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Smith, Harry">
<TAPES.F NAME = "Blue Velvet">
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
  <TAPES.F_CATS_MV CAT_NAME = "Avant Garde"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V2001"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V5004"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V8181"/>
</CUSTOMER.F_record>
</ROOT>
>
```

Creating an XML Document From Multiple Files with a DTD

The following example illustrates creating an XML document from multiple files with a DTD. To include the DTD, use the WITHDTD keyword.

```
>SELECT CUSTOMER.F.NAME, TAPES.F.NAME, CAT_NAME, TAPES_RENTED FROM
CUSTOMER.F, TAPES.F WHERE TAPES_RENTED = TAPES.F.@ID ORDER BY
CUSTOMER.F.NAME TOXML WITHDTD;


<?xml version="1.0"?>
<!DOCTYPE ROOT[
<!ELEMENT ROOT (CUSTOMER.F_record*)>
<!ELEMENT CUSTOMER.F_record ( TAPES.F* , CUSTOMER.F_TAPE_INFO_MV*
)>
<!ATTLIST CUSTOMER.F_record
        NAME CDATA #REQUIRED
>
<!ELEMENT TAPES.F ( TAPES.F_CATS_MV* )>
<!ATTLIST TAPES.F
        NAME CDATA #IMPLIED
>
<!ELEMENT TAPES.F_CATS_MV  EMPTY>
<!ATTLIST TAPES.F_CATS_MV
        CAT_NAME CDATA #IMPLIED
>
<!ELEMENT CUSTOMER.F_TAPE_INFO_MV  EMPTY>
<!ATTLIST CUSTOMER.F_TAPE_INFO_MV
        TAPES_RENTED CDATA #IMPLIED
>
]>
<ROOT>
<CUSTOMER.F_record NAME = "Barrie, Dick">
<TAPES.F NAME = "Citizen Kane">
  <TAPES.F_CATS_MV CAT_NAME = "Old Classic"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V996"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Best, George">
<TAPES.F NAME = "Love Story">
  <TAPES.F_CATS_MV CAT_NAME = "Romance"/>
  <TAPES.F_CATS_MV CAT_NAME = "Tear Jerker"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "B2297"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Bowie, David">
<TAPES.F NAME = "The Stalker">
  <TAPES.F_CATS_MV CAT_NAME = "Avant Garde"/>
  <TAPES.F_CATS_MV CAT_NAME = "Science Fiction"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V9961"/>
```

```
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Chase, Carl">
<TAPES.F NAME = "'Round Midnight">
  <TAPES.F_CATS_MV CAT_NAME = "Musical"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V8481"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V1254"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V4951"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Chase, Carl">
<TAPES.F NAME = "American Graffiti ">
  <TAPES.F_CATS_MV CAT_NAME = "Comedy"/>
  <TAPES.F_CATS_MV CAT_NAME = "Childrens Movie"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V8481"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V1254"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V4951"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Chase, Carl">
<TAPES.F NAME = "Flash Gordon">
  <TAPES.F_CATS_MV CAT_NAME = "Science Fiction"/>
  <TAPES.F_CATS_MV CAT_NAME = "Childrens Movie"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V8481"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V1254"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V4951"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Faber, Harry">
<TAPES.F NAME = "To Kill A Mockingbird">
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
  <TAPES.F_CATS_MV CAT_NAME = "Political"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V5151"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Fischer, Carrie">
<TAPES.F NAME = "Girl Friday">
  <TAPES.F_CATS_MV CAT_NAME = "Comedy"/>
  <TAPES.F_CATS_MV CAT_NAME = "Old Classic"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V110"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "James, Bob">
<TAPES.F NAME = "Blue Velvet">
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
  <TAPES.F_CATS_MV CAT_NAME = "Avant Garde"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V2001"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jamieson, Dale">
<TAPES.F NAME = "2001">
  <TAPES.F_CATS_MV CAT_NAME = "Science Fiction"/>
```

```
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V6670"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Bob">
<TAPES.F NAME = "Z">
  <TAPES.F_CATS_MV CAT_NAME = "Political"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V4341"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Freddie">
<TAPES.F NAME = "Help">
  <TAPES.F_CATS_MV CAT_NAME = "Childrens Movie"/>
  <TAPES.F_CATS_MV CAT_NAME = "Comedy"/>
  <TAPES.F_CATS_MV CAT_NAME = "Musical"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V9431"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Mable">
<TAPES.F NAME = "Psycho">
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V1249"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V4499"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Mable">
<TAPES.F NAME = "Gone With The Wind">
  <TAPES.F_CATS_MV CAT_NAME = "Romance"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V1249"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V4499"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Samuel">
<TAPES.F NAME = "'Round Midnight">
  <TAPES.F_CATS_MV CAT_NAME = "Musical"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V1254"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V8481"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Samuel">
<TAPES.F NAME = "Flash Gordon">
  <TAPES.F_CATS_MV CAT_NAME = "Science Fiction"/>
  <TAPES.F_CATS_MV CAT_NAME = "Childrens Movie"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V1254"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V8481"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Partner, Bonnie">
<TAPES.F NAME = "Tammy">
  <TAPES.F_CATS_MV CAT_NAME = "Romance"/>
</TAPES.F>
```

```
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "B914"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "B2297"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Partner, Bonnie">
<TAPES.F NAME = "Love Story">
  <TAPES.F_CATS_MV CAT_NAME = "Romance"/>
  <TAPES.F_CATS_MV CAT_NAME = "Tear Jerker"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "B914"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "B2297"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Smith, Harry">
<TAPES.F NAME = "Journey Abroad">
  <TAPES.F_CATS_MV CAT_NAME = "B - Movie"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V2001"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V5004"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V8181"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Smith, Harry">
<TAPES.F NAME = "Catch 22">
  <TAPES.F_CATS_MV CAT_NAME = "Comedy"/>
  <TAPES.F_CATS_MV CAT_NAME = "Avant Garde"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V2001"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V5004"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V8181"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Smith, Harry">
<TAPES.F NAME = "Blue Velvet">
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
  <TAPES.F_CATS_MV CAT_NAME = "Avant Garde"/>
</TAPES.F>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V2001"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V5004"/>
  <CUSTOMER.F_TAPE_INFO_MV TAPES_RENTED = "V8181"/>
</CUSTOMER.F_record>
</ROOT>
>
```

Creating an XML Document From Multiple Files Using a Mapping File

As with RetrieVe, you can create a mapping file to define transformation rules differing from the defaults. For information about creating the mapping file, see "Creating a Mapping File."

The following mapping file defines rules for the CUSTOMER and TAPES file.

```
<?XML version= "1.0" ?>
<U2xml-mapping xmlns:U2xml="http://www.informix.com/U2-xml">
<! -- CUSTOMER AND TAPE MAPPING FILE -->
<U2xml:mapping file = "TAPES.F"
         field = "CAT_NAME"
   map-to= Cat_name
   TYPE= "MV"
/>
<U2xml:mapping file = "CUSTOMER.F"
         field = "TAPES_RENTED"
   map-to="Tapes_rented"
   TYPE="MV"
/>
<U2xml:mapping file = "CUSTOMER.F"
         field = "DATE_OUT"
   TYPE="MV"
/>
<U2xml:mapping file = "CUSTOMER.F"
         field = "DATE_DUE"
   TYPE="MV"
/>
</U2xml-mapping>
```

To use this mapping file in the SELECT statement, specify the XMLMAPPING keyword, as shown in the following example:

***Note**: You must surround the name of the mapping file in single quotation marks.*

```
02 SELECT CUSTOMER.F.NAME, TAPES.F.NAME, CAT_NAME, DATE_OUT,
DATE_DUE FROM CUSTO
MER.F, TAPES.F WHERE TAPES_RENTED = TAPES.F.@ID ORDER BY
CUSTOMER.F.NAME TOXML X
MLMAPPING 'CUST.TAPE.MAP';


<?xml version="1.0"?>
<ROOT>
<CUSTOMER.F_record NAME = "Barrie, Dick">
<TAPES.F NAME = "Citizen Kane">
  <TAPES.F_CATS_MV CAT_NAME = "Old Classic"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
</TAPES.F>
  <TAPES.F_CATS-MV DATE_OUT = "03/29/94" DATE_DUE = "03/31/94"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Best, George">
<TAPES.F NAME = "Love Story">
  <TAPES.F_CATS_MV CAT_NAME = "Romance"/>
  <TAPES.F_CATS_MV CAT_NAME = "Tear Jerker"/>
</TAPES.F>
  <TAPES.F_CATS-MV DATE_OUT = "03/29/94" DATE_DUE = "03/31/94"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Bowie, David">
<TAPES.F NAME = "The Stalker">
  <TAPES.F_CATS_MV CAT_NAME = "Avant Garde"/>
  <TAPES.F_CATS_MV CAT_NAME = "Science Fiction"/>
</TAPES.F>
  <TAPES.F_CATS-MV DATE_OUT = "04/15/94" DATE_DUE = "04/17/94"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Chase, Carl">
<TAPES.F NAME = "'Round Midnight">
  <TAPES.F_CATS_MV CAT_NAME = "Musical"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
  <TAPES.F_CATS-MV DATE_OUT = "04/20/94" DATE_DUE = "04/22/94"/>
  <TAPES.F_CATS-MV DATE_OUT = "04/20/94" DATE_DUE = "04/22/94"/>
  <TAPES.F_CATS-MV DATE_OUT = "04/21/94" DATE_DUE = "04/23/94"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Chase, Carl">
<TAPES.F NAME = "American Graffiti ">
  <TAPES.F_CATS_MV CAT_NAME = "Comedy"/>
  <TAPES.F_CATS_MV CAT_NAME = "Childrens Movie"/>
</TAPES.F>
  <TAPES.F_CATS-MV DATE_OUT = "04/20/94" DATE_DUE = "04/22/94"/>
  <TAPES.F_CATS-MV DATE_OUT = "04/20/94" DATE_DUE = "04/22/94"/>
  <TAPES.F_CATS-MV DATE_OUT = "04/21/94" DATE_DUE = "04/23/94"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Chase, Carl">
<TAPES.F NAME = "Flash Gordon">
  <TAPES.F_CATS_MV CAT_NAME = "Science Fiction"/>
```

```
  <TAPES.F_CATS_MV CAT_NAME = "Childrens Movie"/>
</TAPES.F>
  <TAPES.F_CATS-MV DATE_OUT = "04/20/94" DATE_DUE = "04/22/94"/>
  <TAPES.F_CATS-MV DATE_OUT = "04/20/94" DATE_DUE = "04/22/94"/>
  <TAPES.F_CATS-MV DATE_OUT = "04/21/94" DATE_DUE = "04/23/94"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Faber, Harry">
<TAPES.F NAME = "To Kill A Mockingbird">
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
  <TAPES.F_CATS_MV CAT_NAME = "Political"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
  <TAPES.F_CATS-MV DATE_OUT = "04/19/94" DATE_DUE = "04/21/94"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Fischer, Carrie">
<TAPES.F NAME = "Girl Friday">
  <TAPES.F_CATS_MV CAT_NAME = "Comedy"/>
  <TAPES.F_CATS_MV CAT_NAME = "Old Classic"/>
</TAPES.F>
  <TAPES.F_CATS-MV DATE_OUT = "04/23/94" DATE_DUE = "04/25/94"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "James, Bob">
<TAPES.F NAME = "Blue Velvet">
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
  <TAPES.F_CATS_MV CAT_NAME = "Avant Garde"/>
</TAPES.F>
  <TAPES.F_CATS-MV DATE_OUT = "04/25/94" DATE_DUE = "04/27/94"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jamieson, Dale">
<TAPES.F NAME = "2001">
  <TAPES.F_CATS_MV CAT_NAME = "Science Fiction"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
  <TAPES.F_CATS-MV DATE_OUT = "04/24/94" DATE_DUE = "04/26/94"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Bob">
<TAPES.F NAME = "Z">
  <TAPES.F_CATS_MV CAT_NAME = "Political"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
  <TAPES.F_CATS-MV DATE_OUT = "04/24/94" DATE_DUE = "04/26/94"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Freddie">
<TAPES.F NAME = "Help">
  <TAPES.F_CATS_MV CAT_NAME = "Childrens Movie"/>
  <TAPES.F_CATS_MV CAT_NAME = "Comedy"/>
  <TAPES.F_CATS_MV CAT_NAME = "Musical"/>
</TAPES.F>
  <TAPES.F_CATS-MV DATE_OUT = "04/23/94" DATE_DUE = "04/25/94"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Mable">
<TAPES.F NAME = "Psycho">
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
```

```
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
  <TAPES.F_CATS-MV DATE_OUT = "04/23/94" DATE_DUE = "04/25/94"/>
  <TAPES.F_CATS-MV DATE_OUT = "04/25/94" DATE_DUE = "04/27/94"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Mable">
<TAPES.F NAME = "Gone With The Wind">
  <TAPES.F_CATS_MV CAT_NAME = "Romance"/>
</TAPES.F>
  <TAPES.F_CATS-MV DATE_OUT = "04/23/94" DATE_DUE = "04/25/94"/>
  <TAPES.F_CATS-MV DATE_OUT = "04/25/94" DATE_DUE = "04/27/94"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Samuel">
<TAPES.F NAME = "'Round Midnight">
  <TAPES.F_CATS_MV CAT_NAME = "Musical"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
</TAPES.F>
  <TAPES.F_CATS-MV DATE_OUT = "04/24/94" DATE_DUE = "04/26/94"/>
  <TAPES.F_CATS-MV DATE_OUT = "04/25/94" DATE_DUE = "04/27/94"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Jones, Samuel">
<TAPES.F NAME = "Flash Gordon">
  <TAPES.F_CATS_MV CAT_NAME = "Science Fiction"/>
  <TAPES.F_CATS_MV CAT_NAME = "Childrens Movie"/>
</TAPES.F>
  <TAPES.F_CATS-MV DATE_OUT = "04/24/94" DATE_DUE = "04/26/94"/>
  <TAPES.F_CATS-MV DATE_OUT = "04/25/94" DATE_DUE = "04/27/94"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Partner, Bonnie">
<TAPES.F NAME = "Tammy">
  <TAPES.F_CATS_MV CAT_NAME = "Romance"/>
</TAPES.F>
  <TAPES.F_CATS-MV DATE_OUT = "01/01/94" DATE_DUE = "01/03/94"/>
  <TAPES.F_CATS-MV DATE_OUT = "01/03/94" DATE_DUE = "01/05/94"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Partner, Bonnie">
<TAPES.F NAME = "Love Story">
  <TAPES.F_CATS_MV CAT_NAME = "Romance"/>
  <TAPES.F_CATS_MV CAT_NAME = "Tear Jerker"/>
</TAPES.F>
  <TAPES.F_CATS-MV DATE_OUT = "01/01/94" DATE_DUE = "01/03/94"/>
  <TAPES.F_CATS-MV DATE_OUT = "01/03/94" DATE_DUE = "01/05/94"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Smith, Harry">
<TAPES.F NAME = "Journey Abroad">
  <TAPES.F_CATS_MV CAT_NAME = "B - Movie"/>
</TAPES.F>
  <TAPES.F_CATS-MV DATE_OUT = "04/24/94" DATE_DUE = "04/26/94"/>
  <TAPES.F_CATS-MV DATE_OUT = "04/23/94" DATE_DUE = "04/25/94"/>
  <TAPES.F_CATS-MV DATE_OUT = "04/24/94" DATE_DUE = "04/26/94"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Smith, Harry">
<TAPES.F NAME = "Catch 22">
  <TAPES.F_CATS_MV CAT_NAME = "Comedy"/>
```

```
  <TAPES.F_CATS_MV CAT_NAME = "Avant Garde"/>
</TAPES.F>
  <TAPES.F_CATS-MV DATE_OUT = "04/24/94" DATE_DUE = "04/26/94"/>
  <TAPES.F_CATS-MV DATE_OUT = "04/23/94" DATE_DUE = "04/25/94"/>
  <TAPES.F_CATS-MV DATE_OUT = "04/24/94" DATE_DUE = "04/26/94"/>
</CUSTOMER.F_record>
<CUSTOMER.F_record NAME = "Smith, Harry">
<TAPES.F NAME = "Blue Velvet">
  <TAPES.F_CATS_MV CAT_NAME = "Horror"/>
  <TAPES.F_CATS_MV CAT_NAME = "Drama"/>
  <TAPES.F_CATS_MV CAT_NAME = "Avant Garde"/>
</TAPES.F>
  <TAPES.F_CATS-MV DATE_OUT = "04/24/94" DATE_DUE = "04/26/94"/>
  <TAPES.F_CATS-MV DATE_OUT = "04/23/94" DATE_DUE = "04/25/94"/>
  <TAPES.F_CATS-MV DATE_OUT = "04/24/94" DATE_DUE = "04/26/94"/>
</CUSTOMER.F_record>
</ROOT>
>
```

Chapter

12

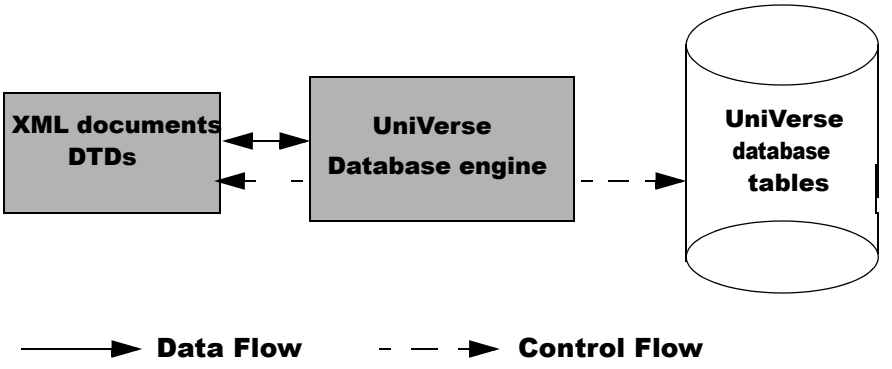
Receiving an XML Document with UniVerse SQL

Receiving an XML Document through UniVerse BASIC

XML documents are text documents, intended to be processed by an application, such as a web browser. UniVerse enables you to receive and create XML documents, and process them through UniVerse BASIC, UniVerse SQL, or RetrieVe.

You can receive an XML document, then read the document through UniVerse BASIC, and execute UniVerse BASIC commands against the XML data.

The following example illustrates the UniVerse implementation of receiving XML documents:



Defining Extraction Rules

You must define the extraction rules for each XML document you receive. This extraction file defines where to start extracting data from the XML document, how to construct UniVerse data file fields from the data, the name of the data file dictionary to use, and how to treat a missing value.

***Note**: The extraction file can reside anywhere. We recommend that it reside in the &XML& file, and have a file extension of .ext.*

Extraction File Syntax

An extraction file has the following format:

```
<?XML version = “1.0”?>
<U2xml-extraction xmlns:U2xml=”http://www.ibm.com/U2-xml”>
 <!-- there must be one and only one <U2xml:extraction> element with
mode/start/dictionary -->
 <U2xml:extraction
                start=”xpath_expression”
                dictionary=”dict1 filename ...”
                null=”NULL” | “EMPTY”
/>
<! -- there can be zero or multiple <U2xml:extraction> elements with
field/path/format -->
 <U2xml:field_extraction
                field=”field name”
                path=”xpath_expression”
/>

...

</U2xml_extraction>
```

The following tables describes the elements of the extraction file.

Element	Description
XML version	The XML version number.
Namespace	The name of the namespace. A namespace is a unique identifier that links an XML markup element to a specific DTD. They indicate to the processing application, for example, a browser, which DTD you are using.
start	Defines the starting node in the XML file. This specifies where UniVerse should begin extracting data from the XML file.
dictionary	Specifies the UniVerse dictionary of the file name to use when viewing the XML data.

Extraction File Elements

Element	Description
null	Determines how to treat a missing node. If null is set to "NULL," a missing node will be result in the null value in the resulting output. If null is set to EMPTY, a missing node will be replaced with an empty string.
field	The field name.
path	The XPath definition for the field you are extracting.

Extraction File Elements (Continued)

Defining the XPath

***Note**: The examples in this section use the STUDENT.F and COURSES files. To create these files, execute the MAKE.DEMO.FILES from the TCL prompt.*

In XML, the XPath language describes how to navigate an XML document, and describes a section of the document that needs to be transformed. It also enables you to point to certain part of the document.

***Note**: For the full XPath specification, see http:/www.w3.org/TR/xpath.*

At this release, UniVerse supports the following XPath syntax:

Parameter	Description
/	Node path divider.
.	Current node.
..	Parent node.
@	Attributes
text()	The contents of the element.
xmldata()	The remaining, unparsed, portion of the selected node.
,	Node path divider, and also specifies multivalue or multi-subvalued field.

Extraction File Parameters

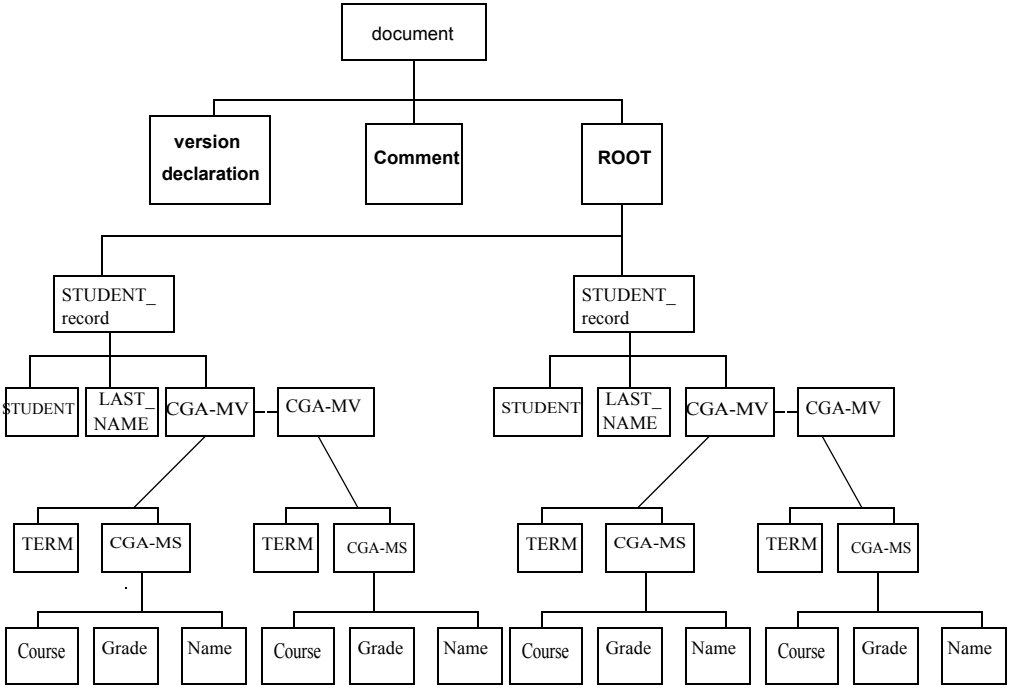
Consider the following DTD and XML document:

```
<?xml version="1.0"?>
<!DOCTYPE ROOT[
<!ELEMENT ROOT (STUDENT_record*)>
<!ELEMENT STUDENT_record ( STUDENT , Last_Name , CGA-MV* )>
<!ELEMENT STUDENT (#PCDATA) >
<!ELEMENT Last_Name (#PCDATA) >
<!ELEMENT CGA-MV ( Term* , CGA-MS* )
<!ELEMENT Term (#PCDATA) >
<!ELEMENT CGA-MS ( Crs__* , GD* , Course_Name* )>
<!ELEMENT Crs__ (#PCDATA) >
<!ELEMENT GD (#PCDATA) >
<!ELEMENT Course_Name (#PCDATA) >
]>

 <ROOT>
 <STUDENT_record>
   <STUDENT>424-32-5656</STUDENT>
   <Last_Name>Martin</Last_Name>
   <CGA-MV>
      <Term>SP94</Term>
      <CGA-MS>
        <Crs__>PY100</Crs__>
        <GD>C</GD>
        <Course_Name>Introduction to Psychology</Course_Name>
      </CGA-MS>
      <CGA-MS>
        <Crs__>PE100</Crs__>
        <GD>C</GD>
        <Course_Name>Golf - I </Course_Name>
      </CGA-MS>
    </CGA-MV>
 </STUDENT_record>
 <STUDENT_record>
   <STUDENT>414-44-6545</STUDENT>
   <Last_Name>Offenbach</Last_Name>
   <CGA-MV>
      <Term>FA93</Term>
      <CGA-MS>
        <Crs__>CS104</Crs__>
        <GD>D</GD>
        <Course_Name>Database Design</Course_Name>
      </CGA-MS>
      <CGA-MS>
        <Crs__>MA101</Crs__>
        <GD>C</GD>
        <Course_Name>Math Principles </Course_Name>
      </CGA-MS>
      <CGA-MS>
        <Crs__>FA100</Crs__>
        <GD>C</GD>
        <Course_Name>Visual Thinking </Course_Name>
      </CGA-MS>
```

```
        </CGA-MV>
       <CGA-MV>
          <Term>SP94</Term>
         <CGA-MS>
            <Crs__>CS105</Crs__>
            <GD>B</GD>
            <Course_Name>Database Design</Course_Name>
          <CGA-MS>
            <Crs__>MA102</Crs__>
            <GD>C</GD>
            <Course_Name>Introduction of Psychology</Course_Name>
         </CGA-MS>
       </CGA-MV>
      <STUDENT_record>
        <STUDENT>221-34-5665</STUDENT>
        <Last_Name>Miller</Last_Name>
        <CGA-MV>
          <Term>FA93</Term>
          <CGA-MS>
            <Crs__>EG110</Crs__>
            <GD>C</GD>
            <Course_Name>Engineering Principles</Course_Name>
          </CGA-MS>
          <CGA-MS>
            <Crs__>PY100</Crs__>
            <GD>B</GD>
            <Course_Name>Introduction to Psychology</Course_Name>
          </CGA-MS>
        </CGA-MV>
          <Term>SP94</Term>
          <CGA-MS>
            <Crs__>EG140</Crs__>
            <GD>B</GD>
            <Course_Name>Fluid Mechanics</Course_Name>
          </CGA-MS>
          <CGA-MS>
            <Crs__>MA221</Crs__>
            <GD>B</GD>
            <Course_Name>Calculus -- II</Course_Name>
          </CGA-MS>
        </CGA-MV>
      </STUDENT_record>
    </ROOT>
```

This document could be displayed as a tree, as shown in the following example:


document
version declaration
Comment
ROOT
STUDENT_ record
STUDENT_ record
STUDENT
LAST_ NAME
CGA-MV
CGA-MV
STUDENT
LAST_ NAME
CGA-MV
CGA-MV
TERM
CGA-MS
TERM
CGA-MS
TERM
CGA-MS
TERM
CGA-MS
Course
Grade
Name
Course
Grade
Name
Course
Grade
Name
Course
Grade
Name

In the previous example, each element in the XML document appears in a box. These boxes are called nodes when using XPath terminology. As shown in the example, nodes are related to each other. The relationships in this example are:

- The document node contains the entire XML document.
- The document node contains three children: the version declaration, the comment node, and the ROOT node. These three children are siblings.
- The ROOT node contains two STUDENT nodes, which are children of ROOT, and are siblings of each other.
- The STUDENT node contains three nodes: the ID, NAME, and CGA-MV. These nodes are children of the STUDENT node, and are siblings of each other.
- The CGA-MV node contains TERM nodes and CGA-MS nodes. These nodes are children of the CGA-MV node, and are siblings of each other.
- Finally, the CGA-MS node contains three nodes: the Course, Grade, and Name nodes. These three nodes are children of the CGA-MS node, and are siblings of each other.

When you define the XPath in the extraction file, you must indicate how to treat these different nodes.

Defining the Starting Location

The first thing to define in the extraction file is the starting node in the XML document from which you want to begin extracting data. In our example, we want to start at the STUDENT_record node. You can also define the dictionary file to use when executing RetrieVe LIST statements or UniVerse SQL SELECT statements against the data.

The following example illustrates how to specify the STUDENT_record node as the starting node, and use the STUDENT dictionary file:

```
<file_extraction start = "ROOT/STUDENT_record " dictionary =
"STUDENT"/>
```

If you want to start the extraction at the CGA-MV node, specify the file extraction node as follows:

```
<file_extraction start = "ROOT/STUDENT_record/CGA-MV" dictionary =
"STUDENT"/>
```

Specifying Field Equivalents

Next, you specify the rules for extracting fields from the XML document. In this example, there are six fields to extract (@ID, NAME, TERM, COURSE, GRADE and NAME).

Extracting Singlevalued Fields

The following example illustrates how to define the extraction rule for two singl-evalued fields:

```
<field_extraction field = "@ID" path = "STUDENT/text()",/>
<field_extraction field = "LNAME" path = "Last_Name/text()",/>
```

In the first field extraction, the @ID value in the UniVerse record will be extracted from the STUDENT node. The text in the STUDENT node will be the value of @ID.

In the next field extraction rule, the LNAME field will be extracted from the text found in the Last_Name node in the XML document.

Extracting Multivalued Fields

To access multivalued data in the XML document, you must specify the location path relative to the start node (full location path).

UniVerse uses the "/" character to specify levels of the XML document. The "/" tells the xmlparser to go to the next level when searching for data.

Use a comma (",") to tell the xmlparser where to place marks in the data.

The following example illustrates how to define the path for a multivalued field (SEMESTER) in the XML document:

```
<field_extraction field "SEMESTER" path = "CGA-MV,Term/text()" />
```

In this example, the value of the SEMESTER field in the UniVerse data file will be the text in the Term node. The "/" in the path value specifies multiple levels in the XML document, as follows:

1. Start at the CGA-MV node in the XML document.
2. From the CGA-MV node, go to the next level, the Term node.
3. Return the text from the Term node as the first value of the SEMESTER field in the UniVerse data file.

4. Search for the next CGA-MV node under the same STUDENT, and extract the text from the Term node belonging to that CGA-MV node, and make it the next multivalue. The comma tells the xmlparser to get the node preceding the command for the next sibling.
5. Continue processing all the CGA-MV nodes belonging to the same parent.

The SEMESTER field will appear in the following manner:

Term<Value mark>Term<Value Mark>...

Extracting Multisubvalued Fields

As with multivalued fields, UniVerse uses the "/" character to specify levels of the XML document. The "/" tells the xmlparser to go to the next level when searching for data.

Use the comma (",") to define where to place marks in the data. You can specify 2 levels of marks, value marks and subvalue marks.

Consider the following example of a field extraction XPath definition:

```
<field_extraction field = "COURSE_NBR" path = "CGA-MV, CGA-MS,
Course_Name/ text()" />
```

In this case, the resulting data will appear as follows:

<Value Mark>Course_Name <subvalue mark>Course_Name<subvalue mark>Course_Name...<Value Mark>...

Suppose the XPath definition contains another level of data, as shown in the next example:

```
<field_extraction field = "COURSE_NBR" path = "CGA-MV/CGA-MS/
Course_Name/Comment/text()"/>
```

You must determine where you want the marks to appear in the resulting data. If you want Comment to represent the multi-subvalue, begin inserting commas after CGA-MS, since the Comment is three levels below CGA-MS.

```
<field_extraction field = "COURSE_NBR" path = "CGA-MV/CGA-MS,
Course_Name,Comment/text()" />
```

Suppose we add yet another level of data to XPath definition:

```
<field_extraction field = "COURSE_NBR" path = "CGA-MV/CGA-MS,
Course_Name,Comment,activities/text()" />
```

This is not a valid XPath, since there are more than three levels of XML data. If you want your data to have subvalue marks between Comment and activities, change the XPath definition as follows:

```
<field_extraction field = "COURSE_NBR" path = "CGA-MV/CGA-MS/
Course_Name,Comment,activities/text()" />
```

The "/" and the "," characters are synonymous when defining the navigation path, UniVerse still uses the "/" ***AND*** the "," to define the navigation path of the data, but only the "," to determine the location of the marks in the resulting data.

Like multivalued fields, you must start at the XPath with the parent node of the multivalue.

The next example illustrates how to extract data for a multi-subvalued field:

```
<field_extraction field = "COURSE_NBR" path = "CGA-MV, CGA-MS,
Crs__/text()" />
```

The COURSE_NBR field in the UniVerse data file will be extracted as follows:

1. Start at the CGA-MV node in the XML document, under the start node (ROOT/STUDENT_record).
2. From the first CGA-MV node, go to the next level, the CGA-MS node.
3. From the first CGA-MS node, go to the Crs__ node. Return the text from the Crs__node, and make that text the first multi-subvalue of COURSE_NBR.
4. Go back to the CGA-MS node, and search the siblings of the CGA-MS nodes to see if there are any more CGA-MS nodes of the same name. If any are found, return the Crs__/text() under these nodes, and make them the next multi-subvalues of COURSE_NBR.
5. Go back to the CGA-MV node and search for siblings of the CGA-MS node that have the same CGA-MV node name. If any are found, repeat steps 3 and 4 to get the values for these CGA-MV nodes, and make them multivalues.

The COURSE_NBR field will look like this:

<Field Mark>Crs__text() value under 1st CGA-MS node of 1st CGA-MV node<multi-subvalue mark>Crs__text() under 2nd CGA-MS node of 1st CGA-MV node<multi-subvalue mark>...<multivalue mark>Crs__text() under 1st CGA-MS node of the 2nd CGA-MV node<multi-subvalue mark>Crs__text() under 2nd CGA-MS node of the 2nd CGA-MV node<multi-subvalue mark>Crs__text() value under the 3rd CGS-MS node of the 2nd CGA-MV node>...<Field Mark>

The following example illustrates the complete extraction file for the above examples:

```
 <U2XML_extraction>
    <file_extraction start = "/ROOT/STUDENT_record" dictionary =
"D_MYSTUDENT"
    <!--field extraction rule in element mode-->
      <field_extraction field = "@ID" path = "STUDENT/text()"/>
     <field_extraction field = "LNAME" path = "Last_Name/text()"/>
     <field_extraction field = "SEMESTER" path = "CGA-MV/Term/text()"/>
      <field_extraction field = "COURSE_NBR" path = "CGA-MV, CGA-MS,
Crs__/text"/>
      <field_extraction field = "COURSE_GRD" path = "CGA-MV, CGA-MS,
GD/text()"/>
      <field_extraction field = "COURSE_NAME" path = "CGA-MV, CGA-MS,
Course_Name/text()"/>
 </U2XML_extraction>
```

Extracting XML Data through UniVerse BASIC

Complete the following steps to access the XML data through UniVerse BASIC:

1. Familiarize yourself with the elements of the DTD associated with the XML data you are receiving.
2. Create the extraction file for the XML data.
3. Prepare the XML document using the UniVerse BASIC PrepareXML function.
4. Open the XML document using the UniVerse BASIC OpenXMLData function.
5. Read the XML data using the UniVerse BASIC ReadXMLData function.
6. Close the XML document using the UniVerse BASIC CloseXMLData function.
7. Release the XML document using the UniVerse BASIC ReleaseXML function.

Preparing the XML Document

You must first prepare the XML document in the UniVerse BASIC program. This step allocates memory for the XML document, opens the document, determines the file structure of the document, and returns the file structure.

Status=PrepareXML(*xml_file*,*xml_handle*)

The following table describes each parameter of the syntax.

Parameter	Description
xml_file	The path to the file where the XML document resides.
xml_handle	The return value. The return value is the UniVerse BASIC variable for *xml_handle*. Status is one of the following return values: XML.SUCCESS Success XML.ERROR Error

PrepareXML Parameters

Example

The following example illustrates use of the PrepareXML function:

```
STATUS = PrepareXML("&XML&/MYSTUDENT.XML",STUDENT_XML)
IF STATUS=XML.ERROR THEN
   STATUS = XMLError(errmsg)
   PRINT "error message ":errmsg
   STOP "Error when preparing XML document "
END
```

Opening the XML Document

After you prepare the XML document, open it using the OpenXMLData function.

Status=OpenXMLData(*xml_handle*,*xml_data_extraction_rule*, *xml_data_handle*)

The following table describes each parameter of the syntax.

Parameter	Description
xml_handle	The XML handle generated by the PrepareXML() function.
xml_data_extraction_ rule	The path to the XML extraction rule file.
xml_data_handle	The XML data file handle. The following are the possible return values: XML.SUCCESS Success. XML.ERROR Failed XML.INVALID.HANDLE Invalid XML handle

OpenXMLData Parameters

Example

The following example illustrates use of the OpenXMLData function:

```
status = OpenXMLData("STUDENT_XML",
"&XML&/MYSTUDENT.ext",STUDENT_XML_DATA)
If status = XML.ERROR THEN
   STOP "Error when opening the XML document. "
END
IF status = XML.INVALID.HANDLE THEN
   STOP "Error: Invalid parameter passed."
END
```

Reading the XML Document

After opening the XML document, read the document using the ReadXMLData function. UniVerse BASIC returns the XML data as a dynamic array.

Status=ReadXMLData(*xml_data_handle, rec*)

The following table describes each parameter of the syntax.

Parameter	Description
xml_data_handle	A variable that holds the XML data handle created by the OpenXMLData function.
rec	A mark-delimited dynamic array containing the extracted data. Status if one of the following: XML.SUCCESS Success XML.ERROR Failure XML.INVALID.HANDLE 2 Invalid *xml_data_handle* XML.EOF End of data

ReadXMLData Parameters

After you read the XML document, you can execute any UniVerse BASIC statement or function against the data.

Example

The following example illustrates use of the ReadXMLData function:

```
MOREDATA=1
LOOP WHILE (MOREDATA=1)
   status = ReadXMLData(STUDENT_XML,rec)
   IF status = XML.ERROR THEN
      STOP "Error when preparing the XML document. "
   END ELSE IF status = XML.EOF THEN
      PRINT "No more data"
      MOREDATA = 0
   END ELSE
      PRINT "rec = ":rec
   END
REPEAT
```

Closing the XML Document

After you finish using the XML data, use CloseXMLData to close the dynamic array variable.

Status=CloseXMLData(*xml_data_handle*)

where *xml_data_handle* is the name of the XML data file handle created by the OpenXMLData() function.

The return value is one of the following:

XML.SUCCESS	Success
XML.ERROR	Failure
XML.INVALID.HANDLE2	Invalid *xml_data_handle*

Example

The following example illustrates use of the CloseXMLData function:

```
status = CloseXMLData(STUDENT_XML)
```

Releasing the XML Document

Finally, release the dynamic array variable using the ReleaseXML function.

ReleaseXML(*XMLhandle*)

where *XMLhandle* is the XML handle created by the PrepareXML() function. ReleaseXML destroys the internal DOM tree and releases the associated memory.

Getting Error Messages

Use the XMLError function to get the last error message.,

XMLError(errmsg)

Where errmsg is the error message string, or one of the following return values:

XML.SUCCESS	Success
XML.ERROR	Failure

Example

The following example illustrates a UniVerse BASIC program that prepares, opens, reads, closes, and releases an XML document:

```
# INCLUDE UNIVERSE.INCLUDE XML.H
STATUS=PrepareXML("&XML&/MYSTUDENT.XML",STUDENT_XML)
IF STATUS=XML.ERROR THEN
   STATUS = XMLError(errmsg)
   PRINT "error message ":errmsg
   STOP "Error when preparing XML document "
   END

STATUS =
OpenXMLData("STUDENT_XML","&XML&/MYSTUDENT.ext",STUDENT_XML_DATA)

IF STATUS = XML.ERROR THEN
   STOP "Error when opening the XML document. "
END

IF STATUS = XML.INVALID.HANDLE THEN
   STOP "Error: Invalid parameter passed." END

MOREDATA=1
  LOOP WHILE (MOREDATA=1)
   STATUS=ReadXMLData(STUDENT_XML_DATA,rec)
   IF STATUS = XML.ERROR THEN
        STOP "Error when preparing the XML document. "
   END ELSE IF STATUS = XML.EOF THEN
       PRINT "No more data"
       MOREDATA = 0
   END ELSE
       PRINT "rec = ":rec
       PRINT "rec = ":rec
   END
REPEAT
STATUS = CloseXMLData(STUDENT_XML_DATA)
STATUS = ReleaseXML(STUDENT_XML)
```

Displaying an XML Document through RetrieVe

You can display the contents of an XML file through RetrieVe by defining an extraction file, preparing the XML document, then using LIST to display the contents.

Preparing the XML Document

Before you execute the LIST statement against the XML data, you must first prepare the XML file using the PREPARE XML command.

PREPARE.XML *xml_file xml_data*

xml_file is the path to the location of the XML document.

xml_data is the name of the working file you assign to the XML data.

The following example illustrates preparing the MYSTUDENT.XML document:

```
PREPARE.XML "&XML&/MYSTUDENT.XML" STUDENT_XML
PREPARE.XML successful.
```

Listing the XML Data

Use the RetrieVe LIST command with the XMLDATA option to list the XML data.

LIST XMLDATA *xml_data* "***extraction_file" [fields]***

The following table describes each parameter of the syntax.

Parameter	Description
XMLDATA *xml_data*	Specifies to list the records from the *xml_data* you prepared.
extraction_file	The full path to the location of the extraction file. You must surround the path in quotation marks.
fields	The fields from the dictionary you specified in the extraction file that you want to display.

LIST Parameters for Listing XML Data

When you list an XML document, RetrieVe uses the dictionary you specify in the extraction file. The following example lists the dictionary records for the MYSTUDENT dictionary:

```
>LIST DICT MYSTUDENT

DICT MYSTUDENT    10:25:32am  19 Oct 2001  Page    1

               Type &
Field......... Field. Field........ Conversion.. Column......... Output
Depth &
Name.......... Number Definition... Code........ Heading........ Format
Assoc..

@ID            D    0                           MYSTUDENT       10L    S
LNAME          D    1                           Last Name       15T    S
SEMESTER       D    2                           Term            4L     M
CGA
COURSE_NBR     D    3                           Crs #           5L     M
CGA
COURSE_GRD     D    4                           GD              3L     M
CGA

5 records listed.
```

The fields in the dictionary record must correspond to the position of the fields in the XML extraction file. In the following extraction file, @ID is position 0, LNAME is position 1, SEMESTER is position 2, COURSE_NBR is position 3, COURSE_GRD is position 4, and COURSE_NAME is position 5. The dictionary of the MYSTUDENT file matches these positions.

The following example illustrates listing the fields from the MYSTUDENT XML document, using the MYSTUDENT.EXT extraction file:

```
LIST XMLDATA STUDENT_XML "&XML&/MYSTUDENT.EXT" LNAME SEMESTER COURSE_NBR
COURSE
_GRD COURSE_NAME 11:58:01am  19 Oct 2001  PAGE    1
MYSTUDENT.   Last Name......   Term   Crs #   GD.   Course
Name..............

424-32-565   Martin            SP94   PY100   C     Introduction to
Psycholog
6                                                   y
                                       PE100   C     Golf - I
414-44-654   Offenbach         FA93   CS104   D     Database Design
5
                                       MA101   C     Math Principals
                                       FA100   C     Visual Thinking
                                 SP94  CS105   B     Database Design
                                       MA102   C     Algebra
                                       PY100   C     Introduction to
Psycholog
                                                    y
221-34-566   Miller            FA93   EG110   C     Engineering Principles
5
                                       MA220   B     Calculus- I
                                       PY100   B     Introduction to
Psycholog
                                                    y
                                 SP94  EG140   B     Fluid Mechanics
                                       EG240   B     Circut Theory
                                       MA221   B     Calculus - II

978-76-667   Muller            FA93   FA120   A     Finger Painting
6
                                       FA230   C     Photography Principals
                                       HY101   C     Western Civilization
                                 SP94  FA121   A     Watercorlors
                                       FA231   B     Photography Practicum
                                       HY102   I     Western Civilization -
15
                                                    00 to 1945
521-81-456   Smith             FA93   CS130   A     Intro to Operating
System
4                                                   s
                                       CS100   B     Intro to Computer
Science
                                       PY100   B     Introduction to
Psycholog
                                                    y
                                 SP94  CS131   B     Intro to Operating
System
                                                    s
                                       CS101   B     Intro to Computer
Science
                                       PE220   A     Racquetball
291-22-202   Smith             SP94   FA100   B     Visual Thinking
1
6 records listed.
>
```

Release the XML Document

When you finish with the XML document, release it using the RELEASE.XML.

RELEASE.XML *xml_data*

Displaying an XML Document through UniVerse SQL

You can display an XML document through UniVerse SQL using the SELECT statement.

Preparing the XML Document

Before you execute the SELECT statement against the XML data, you must first prepare the XML file using the PREPARE XML command.

PREPARE.XML *xml_file xml_data*

xml_file is the path to the location of the XML document.

xml_data is the name of the working file you assign to the XML data.

The following example illustrates preparing the MYSTUDENT.XML document:

```
PREPARE.XML "&XML&/MYSTUDENT.XML" STUDENT_XML
PREPARE.XML successful.
```

Listing the XML Data

Use the UniVerse SQL SELECT command with the XMLDATA option to list the XML data.

SELECT clause FROM XMLDATA *xml_data extraction_file*

[WHERE clause]
[WHEN clause [WHEN clause]...]
[GROUP BY clause]
[HAVING clause]
[ORDER BY clause]
[*report_qualifiers*]
[*processing_qualifiers*]

The following table describes each parameter of the syntax.

Parameter	Description
SELECT clause	Specifies the columns to select from the database.
FROM XMLDATA *xml_data*	Specifies the XML document you prepared from which you want o list data.
extraction_file	Specifies the file containing the extraction rules for the XML document.
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.
HAVING clause	Specifies the criteria that grouped rows must meet to be selected.
ORDER BY clause	Sorts selected rows.
report_qualifiers	Formats a report generated by the SELECT statement.
processing_qualifiers	Modifies or reports on the processing of the SELECT statement.

SELECT Parameters

You must specify clauses in the SELECT statement in the order shown in the syntax. 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.

For a full discussion of the UniVerse SQL SELECT statement clauses, see the *UniVerse SQL Reference*.

The following example illustrates displaying the XML document using the UniVerse SQL SELECT statement:

```
>SELECT * FROM XMLDATA STUDENT_XML "&XML&/MYSTUDENT.EXT";
MYSTUDENT.   Last Name......   Term   Crs #   Course Name.............
GD.

424-32-565   Martin            SP94   PY100   Introduction to Psycholog   C
6                                             y
                                     PE100   Golf - I                    C
414-44-654   Offenbach         FA93   CS104   Database Design             D
5
                                     MA101   Math Principals             C
                                     FA100   Visual Thinking             C
                               SP94  CS105   Database Design             B
                                     MA102   Algebra                     C
                                     PY100   Introduction to Psycholog   C
                                             y
221-34-566   Miller            FA93   EG110   Engineering Principles      C
5
                                     MA220   Calculus- I                 B
                                     PY100   Introduction to Psycholog   B
                                             y
                               SP94  EG140   Fluid Mechanics             B
                                     EG240   Circut Theory               B
                                     MA221   Calculus - II               B
978-76-667   Muller            FA93   FA120   Finger Painting             A
6
Press any key to continue...

MYSTUDENT.   Last Name......   Term   Crs #   Course Name.............
GD.

                                     FA230   Photography Principals      C
                                     HY101   Western Civilization        C
                               SP94  FA121   Watercorlors                A
                                     FA231   Photography Practicum       B
                                     HY102   Western Civilization - 15   I
                                             00 to 1945
521-81-456   Smith             FA93   CS130   Intro to Operating System   A
4                                            s
                                     CS100   Intro to Computer Science   B
                                     PY100   Introduction to Psycholog   B
                                             y
                               SP94  CS131   Intro to Operating System   B
                                             s
                                     CS101   Intro to Computer Science   B
                                     PE220   Racquetball                 A
291-22-202   Smith             SP94   FA100   Visual Thinking             B
1

6 records listed.
>
```

Release the XML Document

When you finish with the XML document, release it using the RELEASE.XML.

RELEASE.XML *xml_data*

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X