ULTRIX/SQL

digital

ULTRIX/SQL Reference Manual

Order Number: AA–PBZ6A–TE

ULTRIX

ULTRIX/SQL Reference Manual

Order Number: AA-PBZ6A-TE

June 1990

Software Version:	ULTRIX/SQL Version 1.0
Operating System and Version:	ULTRIX Version 4.0 or higher

This manual contains reference information about interactive SQL statements and ULTRIX/SQL operating system commands. ULTRIX/SQL Version 1.0 is based on Release 6.2 of INGRES.

digital equipment corporation
maynard, massachusetts

Restricted Rights: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause of DFARS 252.227-7013.

© Digital Equipment Corporation 1990
All rights reserved.

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital or its affiliated companies.

The following are trademarks of Digital Equipment Corporation:

digital
CDA
DDIF
DDIS
DEC
DECnet
DECstation
DECUS
DECwindows
DTIF
MASSBUS
MicroVAX
Q-bus
ULTRIX
ULTRIX Mail Connection
ULTRIX Worksystem Software
VAX
VAXstation
VMS
VMS/ULTRIX Connection
VT
XUI

UNIX is a registered trademark of AT&T in the USA and other countries.

Network File System and NFS are trademarks of Sun Microsystems, Inc.

INGRES is a trademark of Ingres Corporation.

Contents

Preface

Purpose of this Document

Intended Audience

Structure of This Manual

Compatibility with Remote Access to Rdb/VMS

Associated Documents

Conventions

References to Products

1 ULTRIX/SQL Syntax

1.1 Introduction 1-1

1.2 Notation and Terminology 1-1

1.2.1 Key Words 1-1

1.2.2 Names 1-2

1.2.3 Comments 1-2

1.2.4 Statement Separator 1-2

1.3 Data Types 1-2

1.3.1 Fixed-Length Character Strings 1-3

1.3.2 Varying-Length Character Strings 1-3

1.3.3 Integer Data Types 1-4

1.3.4 Floating-Point Numeric Data Types 1-4

1.3.5 Dates 1-5

1.3.5.1 Date Output 1-5

1.3.5.2 Date Input 1-5

1.3.5.3 International Date Formats 1-8

1.3.6 Money 1-8

1.3.7 Storage Formats for Data Types 1-9

1.4 Constants 1-10

1.4.1 String Constants 1-10

1.4.2 Numeric Constants 1-11

1.4.3 The null Constant 1-11

1.5 Structured Data 1-11

1.5.1 Tables 1-11

1.5.2 Columns 1-11

1.5.3 Rows 1-12

1.5.4 A Sample Database 1-12

1.5.5 Correlation Names 1-12

1.5.6 Groups 1-14

1.6 Expressions 1-14

1.6.1 Columns 1-15

1.6.2 Parentheses 1-15

1.6.3 Arithmetic Operations 1-15

1.6.3.1 Operators 1-15
1.6.3.2 Arithmetic Operations on Dates 1-16

1.6.4 Type Conversion 1-17

1.6.4.1 Default Numeric Type Conversion 1-17
1.6.4.2 Numeric Overflow 1-18
1.6.4.3 Default Character Type Conversion 1-18
1.6.4.4 Explicit Type Conversion Functions 1-19

1.6.5 Scalar Functions 1-19

1.6.5.1 Numeric Functions 1-20
1.6.5.2 String Functions 1-20
1.6.5.3 Date Functions 1-23
1.6.5.4 The Ifnull Function 1-25

1.6.6 Dbmsinfo( ) Function 1-25

1.6.7 Set Functions 1-26

1.6.7.1 The count Function 1-27
1.6.7.2 Restrictions on the Use of Set Functions 1-28
1.6.7.3 ifnull and Set Functions 1-28

1.7 Search Conditions 1-29

1.7.1 Subqueries 1-30

1.7.2 Comparison Predicate 1-31

1.7.3 like Predicate 1-31
1.7.4 between Predicate 1-32
1.7.5 in Predicate 1-33
1.7.6 any-or-all Predicate 1-33
1.7.7 exists Predicate 1-34
1.7.8 is null Predicate 1-34

1.8 Data Manipulation Statements 1-35
1.8.1 Select 1-35
1.8.2 Update 1-37
1.8.3 Delete 1-37
1.8.4 Insert 1-37

1.9 Relational Concepts 1-38
1.9.1 Expressing Relational Operators in SQL 1-38
1.9.1.1 Projection 1-38
1.9.1.2 Restriction 1-39
1.9.1.3 Cartesian Product 1-39
1.9.1.4 Join 1-39
1.9.2 Nulls and Defaults 1-40

1.10 Transactions 1-41
1.10.1 Transaction Control Statements 1-41
1.10.2 Committing Transactions 1-41
1.10.3 Transaction Rollback 1-42
1.10.4 Interrupt and Timeout Handling in Transactions 1-42
1.10.5 SQL Transaction Semantics 1-42

1.11 Database Procedures 1-43
1.11.1 Using Database Procedures 1-43
1.11.1.1 Permissions on Procedures 1-44
1.11.1.2 Error Handling 1-44
1.11.1.3 Message Handling 1-45
1.11.2 Creating and Executing a Procedure 1-45
1.11.2.1 Creating a Procedure 1-45
1.11.2.2 Executing a Procedure 1-47
1.11.3 Dropping a Procedure 1-47

1.12 Multi-File System Databases 1-47
1.12.1 ULTRIX/SQL Locationnames and Areas 1-47

1.12.2 Assigning Database Tables to Single Areas 1-48
1.12.2.1 Relocating the Database User Tables 1-48
1.12.2.2 Multi-Location Tables 1-48

2 ULTRIX/SQL Statements

2.1 Introduction 2-1

2.2 commit 2-2

2.3 copy 2-3

2.4 create index 2-12

2.5 create integrity 2-15

2.6 create procedure 2-16

2.7 create table 2-19

2.8 create view 2-22

2.9 declare 2-24

2.10 delete 2-25

2.11 drop 2-26

2.12 drop integrity 2-27

2.13 drop permit 2-28

2.14 drop procedure 2-29

2.15 grant 2-30

2.16 help 2-31

2.17 if-then-else 2-34

2.18 insert 2-37

2.19 message 2-39

2.20 modify 2-41

2.21 return 2-47

2.22 rollback 2-48

2.23 save 2-49

2.24 select 2-50

2.25 set 2-54

2.26 update 2-60

2.27 while - endwhile 2-62

3 Terminal Monitor Command Line Interface to ULTRIX/SQL

3.1 Introduction 3-1

3.2 Messages, Prompts and Diagnostics 3-1

3.3 Terminal Monitor Commands 3-2

3.4 Flags 3-4

4 Forms-Based Interface to ULTRIX/SQL

4.1 Overview 4-1

4.2 Entering ULTRIX/SQL Statements 4-1

4.2.1 isql Menu Items 4-2
4.2.2 Help 4-3

4.3 Input/Output Screens 4-3

4.3.1 Input Screen 4-3
4.3.1.1 Loading a File (Read) 4-4
4.3.1.2 Writing to a File (Write) 4-4
4.3.1.3 Clearing the Work Space (Blank) 4-4
4.3.2 Output Screen 4-4
4.3.2.1 Output Frame 4-5
4.3.2.2 Returning to the Input Frame 4-7
4.3.3 Error Messages 4-7

5 ULTRIX/SQL Operating System Commands

5.1 Introduction 5-1

5.2 accessdb 5-2

5.3 auditdb 5-3

5.4 catalogdb 5-6
5.5 ckpdb 5-8
5.6 copydb 5-10
5.7 createdb 5-12
5.8 destroydb 5-15
5.9 finddbs 5-16
5.10 isql 5-17
5.11 optimizedb 5-20
5.12 rollforwarddb 5-24
5.13 sql 5-26
5.14 statdump 5-29
5.15 sysmod 5-31
5.16 unloaddb 5-32

A Key Words

A.1 ULTRIX/SQL A-1
A.2 Embedded ULTRIX/SQL A-1
A.3 ANSI SQL A-2
A.4 Host Language Key Words A-2

B Standards Compliance and Compatibility Information

B.1 Conventions B-1
B.2 Data Types B-2
B.3 Statements B-3

C Using Forms-Based Applications

C.1 Overview C-1

C.2 Accessing Databases C-1

C.3 Menus C-2

C.3.1 Menu Key C-2

C.3.2 Long Menus C-2

C.4 Selecting an Operation from the Menu C-3

C.4.1 Selection by Function Key C-3

C.4.2 Selection by Name C-4

C.4.3 Moving Between Menus C-4

C.5 Standard ULTRIX/SQL Operations C-4

C.6 ULTRIX/SQL Keys C-5

C.6.1 Function Keys C-5

C.6.2 Cursor Movement and Editing Keys C-5

C.6.3 Insert and Overstrike C-6

C.7 On-Screen Help C-6

C.8 Error Messages C-7

D Defining Your Terminal

D.1 Overview D-1

D.2 Defining Your Terminal D-1

D.3 Additional Features of Terminal Use D-2

D.3.1 Printing the Screen D-2

D.3.2 Redrawing the Screen D-3

D.4 Terminal Names for ULTRIX/SQL D-3

E Defining Function and Control Keys

E.1 Overview E-1

E.2 The Purpose of the ULTRIX/SQL Termcap File E-1

E.3 Defining Function and Control Key Mappings E-3

E.4 Types of Mapping Objects E-4

E.4.1 FRS Commands E-5

E.4.2 Menu Items E-6
E.4.3 FRS Keys E-7
E.4.4 Mapping File Syntax E-8
E.4.4.1 Mapping Statements E-9
E.4.4.2 Disabling Statements E-10
E.4.4.3 Comments E-10
E.4.4.4 Mapping File Errors E-11
E.5 Levels of Mapping E-11
E.5.1 Installation-Level Mapping E-11
E.5.2 Terminal-Type Level Mapping E-13
E.5.2.1 VT100 Terminals E-13
E.5.2.2 VT220 Terminals E-15
E.5.3 User-Level Mapping E-17
E.6 Obtaining Information on Mappings E-18
E.7 FRS Command Defaults E-18
E.8 Mapping Restrictions and Troubleshooting E-19
E.8.1 Restrictions and Limitations E-20
E.8.2 Troubleshooting Checklist E-21

F How to Write ULTRIX/SQL Termcap Descriptions

F.1 Overview F-1
F.2 Writing the Description F-2
F.2.1 Preparing the Description F-2
F.2.2 General Format F-2
F.2.3 Special Characters F-3
F.2.4 Names F-4
F.2.5 Capabilities F-4
F.2.6 Suggested Approach to Getting Started F-5
F.3 The Eleven Basic Commands F-6
F.4 Optional Termcap Entries for Advanced Features F-10
F.4.1 Commands Used to Program Video Attributes F-10
F.4.2 Commands Needed for Boxing Characters F-11
F.4.3 Commands Needed for Function Keys F-12

F.5 Commands Needed for Arrow Keys F-14
F.5.1 Commands Used for Color F-14
F.5.2 Command to Specify FRS Mapping File for Terminal F-15
F.5.3 Commands to Optimize Cursor Movement F-15

F.6 Commands for Special Situations F-16
F.6.1 Commands from the ULTRIX Termcap File F-16
F.6.2 Commands for Specific Terminals F-16

F.7 Examples of Termcap Descriptions F-17
F.7.1 VT100 (All-Inclusive) F-17
F.7.2 VT100 (Simple) F-17
F.7.3 Envision 230 .... F-18

G The ULTRIX/SQL Standard Catalog Interface

G.1 Introduction G-1

G.2 Standard Catalog Interface G-2
G.2.1 The iidbcapabilities Catalog G-2
G.2.2 The iidbconstants Catalog G-3
G.2.3 The iitables Catalog G-4
G.2.4 The iicolumns Catalog G-8
G.2.5 The iiphysical_tables Catalog G-10
G.2.6 The iiviews Catalog G-11
G.2.7 The iiindexes Catalog G-11
G.2.8 The iiindex_columns Catalog G-12
G.2.9 The iialt_columns Catalog G-13
G.2.10 The iistats Catalog G-13
G.2.11 The iihistograms Catalog G-14
G.2.12 The iipermits Catalog G-14
G.2.13 The iiintegrities Catalog G-15
G.2.14 The iimulti_locations Catalog G-15
G.2.15 The iiprocedures Catalog G-16
G.2.16 The iiregistrations Catalog G-16

G.3 The DBMS System Catalogs G-17

Index

Preface

Purpose of this Document

The *ULTRIX/SQL Reference Manual* describes the ULTRIX/SQL relational database system and query language. It serves as the primary reference to the syntax and function of ULTRIX/SQL commands and files. It is not intended to serve as a tutorial on structured query languages in general or on the use of relational database systems.

This manual assumes you are using ULTRIX/SQL in an interactive capacity. If you intend to embed SQL commands in a host language application, you should also refer to the *ULTRIX/SQL Reference Guide to Embedded SQL*, which describes the embedded ULTRIX/SQL command set and those features that differ from ULTRIX/SQL in their embedded implementation.

Intended Audience

The *ULTRIX/SQL Reference Manual* is intended for readers who have a basic understanding of how SQL and relational database systems work. The reader is not required to have a detailed understanding of the computer's operating system. However, readers should be familiar with logging on and off, as well as with the computer's file system if advanced features are to be used.

This manual is also intended as the primary reference for the ULTRIX/SQL System Administrator and thus contains some information useful in maintaining the ULTRIX/SQL system.

Structure of This Manual

The *ULTRIX/SQL Reference Manual* is divided into the following parts:

- Chapter 1 describes and explains the syntactic elements of the structured query language, ULTRIX/SQL.

- Chapter 2 is a reference section that describes each SQL command.

- Chapter 3 describes the Terminal Monitor command line interface to interactive ULTRIX/SQL, which you invoke with the **sql** command. The Terminal Monitor interface allows you to enter, edit, save, print or execute an SQL query, as well as perform other useful tasks, by typing special commands on the Terminal Monitor command line.

- Chapter 4 describes how to use the forms-based interface to interactive ULTRIX/SQL, which you invoke with the **isql** command. The forms-based interface allows you to enter, edit, save or execute a query by choosing menu options and entering text in a form displayed on the screen.

- Chapter 5 describes the ULTRIX/SQL operating system commands that allow you to create or destroy databases, execute SQL and embedded SQL programs, and perform some database maintenance functions.

- Appendix A lists the words that are reserved in the ULTRIX/SQL environments.

- Appendix B provides tables that summarize ULTRIX/SQL compliancy with ANSI Standard SQL and X/Open SQL, as well as ULTRIX/SQL compatibility with VAX Rdb/VMS SQL.

- Appendix C describes how to access databases, use menus, and move between fields using the ULTRIX/SQL forms-based utilities, **accessdb**, **catalogdb**, and **isql**.

- Appendix D explains how to define your terminal to ULTRIX/SQL.

- Appendix E explains how to customize the user environment by mapping ULTRIX/SQL operations and cursor movement to function and control keys.

- Appendix F explains how to write termcap definitions for terminals not defined in the standard ULTRIX/SQL termcap file, and how to modify an existing termcap entry.

- Appendix G contains an in-depth description of the system catalogs required to operate an ULTRIX/SQL environment.

Compatibility with Remote Access to Rdb/VMS

This document assumes that your installation does not include Remote Access to Rdb/VMS. If your installation includes this option, be sure to check your documentation for Remote Access to Rdb/VMS for information about syntax that may differ from that described in this manual. Remote Access to Rdb/VMS is a VMS layered product installed on a VMS system running Rdb/VMS, which is connected to your ULTRIX/SQL system(s).

Areas that may differ include:

- Length of **varchar** data type

- Legal row size

- Command usage

- Name length

- Table size

Associated Documents

The following associated manuals are included in your ULTRIX/SQL base system documentation set:

ULTRIX/SQL Database Administrator's Guide
ULTRIX/SQL NET User's Guide
ULTRIX/SQL Operations Guide
ULTRIX/SQL Reference Manual
ULTRIX/SQL Release Notes

Conventions

The following conventions are used to describe syntax in this manual:

- **Boldface** type is used to identify reserved words and required symbols and punctuation in syntax that must be typed as shown when used. Boldface is also used to indicate data types and key names.

- Words in *italics* within text and syntax diagrams represent variable elements of syntax that are to be supplied by the program or the user. Italics are also used within text to introduce new terminology or to show emphasis.

- Color indicates components of the SQL language that are specific to ULTRIX/SQL. These components are not included in current versions of the ANSI or X/Open standard for SQL and, therefore, may not be portable to other implementations of the SQL language.

- Double quotes (" ") within the general text indicate a specific value of a parameter. Double quotes (" ") and single quotes (' ') within syntax and in code examples have specific meanings within the context of SQL or a host programming language.

- Reserved words are shown in boldface, lowercase letters (except in host language examples, where embedded SQL statements appear in uppercase to distinguish them from the host language code). Although ULTRIX/SQL does not actually distinguish between uppercase and lowercase in reserved words, it does convert any uppercase letters to lowercase. This is true only for reserved words. Variables are case sensitive.

- This documentation uses generic keyboard key names. The key names on your particular keyboard may vary slightly from those used in this documentation. Key names joined by a hyphen (such as **Control-P**) indicate that the user is to press the named keys simultaneously.

- Syntax diagrams may continue over several lines. Line wraps and additional lines in statement and command line syntax are indented under the first line of the statement or command.

- Clauses enclosed in square brackets ([ ]) within syntax diagrams are optional.

- Clauses or arguments enclosed in braces ({ }) within syntax diagrams are optional and can be repeated zero or more times.

- Clauses or reserved words separated by vertical bars (|) within syntax diagrams indicate lists from which one element is to be chosen.
- Examples of code are separated from the text and are shown in a special, constant-width typeface.
- *Pseudocode*, a description of an operation without the actual code, is shown in *italics* within examples. This generic program code is used to clarify overall syntax structure without unnecessary detail.
- Within examples, the percent sign (%) represents an operating system prompt, although your system may use another customized prompt.

References to Products

The ULTRIX/SQL documentation to which this manual belongs often refers to products by their abbreviated names:

- ULTRIX/SQL refers to ULTRIX/SQL database software and to its implementation of the SQL language. (Repetitive occurrences of ULTRIX/SQL have been shortened to SQL.)
- Rdb/VMS refers to VAX Rdb/VMS database software.

ULTRIX/SQL Syntax 1

1.1 Introduction

The ULTRIX/SQL structured query language (SQL) allows you to retrieve, manage and maintain data in an existing ULTRIX/SQL database. SQL statements are high-level descriptions of *what* needs to be done rather than *how* it should be done. In relational database terminology, SQL provides "automatic navigation" to the data in the database.

SQL statements can be used in any of several contexts. First, they may be entered directly through the ULTRIX/SQL Terminal Monitor. Second, they can be embedded within programs written in high level languages using embedded SQL.

Consult Chapter 3 of this manual for information about the ULTRIX/SQL Terminal Monitor. For information about embedded ULTRIX/SQL, consult the *ULTRIX/SQL Reference Guide to Embedded SQL* and the ULTRIX/SQL Companion Guides for the host language at your installation.

There are four major ULTRIX/SQL statements, each beginning with one of the following key words:

select
insert
delete
update

As the key words suggest, the statements are used respectively for selecting data, inserting data, deleting data and updating data values. The syntactical forms of the four statements are similar, that of **select** statements being the most general. For that reason, **select** statements are used in this chapter to illustrate SQL syntax.

1.2 Notation and Terminology

1.2.1 Key Words

A list of all key words in ULTRIX/SQL is included as Appendix A to this document. There you will also find the key words of embedded ULTRIX/SQL and ANSI standard SQL.

1.2.2 Names

Names in ULTRIX/SQL are sequences of no more than 32 alphanumeric characters. The underscore (_), number sign (#), at sign (@) and dollar sign ($) characters are considered to be part of the alphanumeric character set. Names must begin with an alphabetic character or an underscore (_). Thus, a name may begin with "a" through "z" (uppercase or lowercase) or underscore (_). The rest of the name may contain any of those characters, as well as the numerals 0 through 9, the number sign (#), the at sign (@) and the dollar sign ($). You may not begin a name with "ii" because names beginning with "ii" are reserved for use by ULTRIX/SQL.

ULTRIX/SQL converts all uppercase letters in a name to lowercase.

1.2.3 Comments

A comment is an arbitrary sequence of characters bounded by /* on the left and by */ on the right. For example:

```
/* This is a comment */
```

A comment so bounded is ignored in query processing.

1.2.4 Statement Separator

Interactive ULTRIX/SQL does not require a statement terminator. Therefore, statement syntax descriptions do not include a semicolon (;).

The semicolon is required as a statement separator when entering queries in the Terminal Monitor if more than one statement precedes **\g**. (Chapter 3 discusses the Terminal Monitor.) A group of statements followed by \g is called a *go block.*

Examples showing modules of code in this manual include optional semicolons as statement separators. Since SQL is used in a variety of contexts, the optional statement separator helps avoid side effects which could result if the context were to change.

1.3 Data Types

There are three classes of data type: character, numeric, and abstract. Character strings can be fixed length (**c** and **char**) or variable length (**text** and **varchar**). Numeric strings may be exact numeric (**integer, smallint, and integer1**) or approximate numeric (**float** and **float4**). The abstract data types are **date** and **money**.

Note

> The following are valid synonyms for the data types discussed in the preceding paragraph: **character** is equivalent to **char**; **integer2** is equivalent to **smallint**; **int** and **integer4** are equivalent to **integer**; **real** is equivalent to **float4**; **double precision** and **float8** are equivalent to **float**.

Class	Sub-Class	Data Type
Numeric	Approximate numeric	**float (float8, double precision)**
		float4 (real)
	Exact numeric	**integer (int, integer4)**
		smallint (integer2)
		integer1
Character	Fixed length	**c**
		char (character)
	Varying length	**text**
		varchar
Abstract		**date**
		money

1.3.1 Fixed-Length Character Strings

Fixed-length character strings are sequences of no more than 2000 ASCII characters. Uppercase and lowercase alphabetic characters within character strings are accepted literally.

Two types of fixed-length character strings are supported in ULTRIX/SQL: **char** and **c**. Strings of type **char** may contain any character, printing or non-printing. Blanks are significant when comparing **char** strings. The preferred fixed-length character type is **char.**

Only characters that can be printed are allowed within **c** strings. Non-printing characters (for instance, control characters) are converted to blanks.

Blanks are ignored when comparing **c** strings. For example, the **c** string

```
the house is around the corner
```

is treated identically to:

```
thehouseisaroundthecorner
```

1.3.2 Varying-Length Character Strings

Varying-length character strings are sequences of no more than 2000 ASCII characters. Uppercase and lowercase alphabetic characters within varying-length character strings are accepted literally.

To include a quotation mark within a variable-length character string, you double it, as in:

```
the ''dog'' is black
```

This evaluates to:

```
the 'dog' is black
```

There are two types of varying-length character strings in ULTRIX/SQL: **text** and **varchar**. The preferred varying-length, character-string type is **varchar**. All ASCII characters except the null character are allowed within **text** strings. Null characters are converted to blanks. Strings of type **varchar** may contain any character, including non-printing characters and the null character.

Blanks are significant in comparisons for both **text** and **varchar** data types. For example, the character string

```
the house is around the corner
```

is considered distinct from:

```
thehouseisaroundthecorner
```

There are some differences in the way the two data types handle blanks. In comparing strings of unequal length, **varchar** effectively adds blanks to the end of the shorter string to make it the same length as the longer string. **Text** does not add blanks; it will consider a shorter string as "less than" a longer string if all characters up to the length of the shorter string are equal. As an example of how this affects comparisons, consider the two strings (a) "abcd\001" and (b) "abcd" where "\001" represents one ASCII character, **controlA**. If these are compared as **text**, then (a) > (b). However, if compared as **varchar**, then (a) < (b), since the "blank" character added by **varchar** has a higher ASCII value than "001."

1.3.3 Integer Data Types

Integer values range from -2,147,483,648 to +2,147,483,647, and they contain no fractional part. Integer values that exceed that range are converted to floating-point. If an integer is less than +32,767 and greater than -32,768, it is treated as a two-byte integer. Otherwise it is converted to a four-byte integer.

The three integer data types are **integer1** (1-byte), **smallint** (2-byte), and **integer** (4-byte).

1.3.4 Floating-Point Numeric Data Types

Floating-point values consist of an integer part, a decimal point and a fractional or scientific notation part of the following format:

[+|-] {*dig*} [.*dig*{*dig*}][e|E [+|-] {*dig*}]

where *dig* is a digit. An example is:

```
2.3 e-02
```

A mantissa with a missing exponent has an exponent of one (1) inserted. Floating-point numbers are double-precision quantities with a range of approximately -10**38 to +10**38 and a precision of approximately 16 significant figures.

The character used to indicate the decimal point, by default a period (.), can be changed by means of the **II_DECIMAL** environment variable, described in the *ULTRIX/SQL Operations Guide*.

The approximate numeric data types are **float4** (4-byte) and **float** (8-byte).

1.3.5 Dates

Dates are represented by the abstract data type, **date**.

1.3.5.1 Date Output

ULTRIX/SQL supports date values that constitute either absolute dates and times or time intervals. ULTRIX/SQL outputs such values as strings of 25 characters with trailing blanks inserted.

ULTRIX/SQL uses one of the following output formats for an absolute date or time:

Format	Example
dd-mmm-yyyy	15-nov-1982
dd-mmm-yyyy hh:mm:ss	15-nov-1982 12:32:48

ULTRIX/SQL displays 24-hour times for the current time zone, which is determined when ULTRIX/SQL is initialized. Dates are stored in Greenwich Mean Time and adjusted for your time zone when they are displayed.

For a time interval, ULTRIX/SQL displays the most significant portions of the interval that fit in the 25-character string. If necessary, ULTRIX/SQL inserts trailing blanks to fill out the string. The output format appears as follows:

yy **yrs** *mm* **mos** *dd* **days** *hh* **hrs** *mm* **mins** *ss* **secs**

Significance is a function of the size of any component of the time interval. For instance, consider the following time interval:

```
5 yrs 4 mos 3 days 12 hrs 32 mins 14 secs
```

ULTRIX/SQL displays such an interval as follows:

```
5 yrs 4 mos 3 days 12 hrs
```

1.3.5.2 Date Input

Dates are input as quoted character strings. ULTRIX/SQL accepts the following valid input formats:

Absolute dates (US)—Valid formats for input of November 15, 1982:

Format	Example
'*mm/dd/yy*'	'11/15/82'
'*dd-mmm-yy*'	'15-nov-82'

Format	Example
'*dd-mmm-yyyy*'	'15-nov-1982'
'*mm-dd-yy*'	'11-15-82'
'*yy.mm.dd*'	'82.11.15'
'*mmddyy*'	'111582'
'*mm/dd*'	'11/15'
'*mm-dd*'	'11-15'
'today'	The string 'today' is a legal absolute date with today's date as its value.
'now'	The string 'now' is a legal absolute date and time with today's date and the current time as its value.

The preceding date formats are the default formats, also known as US format. See the following section titled "International Date Formats" for information about changing the date format conventions to accommodate international conventions.

Absolute times—Valid formats for input of 10:30:00:

Format	Example
'*hh:mm:ss*'	'10:30:00'
'*hh:mm:ss xxx*'	'10:30:00 pst'
'*hh:mm*'	'10:30'

Note

ULTRIX/SQL supplies the appropriate time zone designation. Time formats are assumed to be on a 24-hour clock. However, times entered with a designation of "am" or "pm" are automatically converted to 24-hour internal representation. Any such designation must follow the absolute time and must precede the time zone, if included. If you do not specify a date with an absolute time, today's (that is, the current day's) date is supplied.

Absolute date and time—Valid input formats for November 15, 1982, 10:30:00:

Format	Example
'*mm/dd/yy hh:mm:ss*'	'11/15/82 10:30:00'
'*dd-mmm-yy hh:mm:ss*'	'15-nov-82 10:30:00'
'*mm/dd/yy hh:mm:ss xxx*'	'11/15/82 10:30:00 pst'
'*dd-mmm-yy hh:mm:ss xxx*'	'15-nov-82 10:30:00 pst'
'*mm/dd/yy hh:mm*'	'11/15/82 10:30'

Format	Example
'dd-mmm-yy hh:mm'	'15-nov-82 10:30'
'mm/dd/yy hh:mm xxx'	'11/15/82 10:30 pst'
'dd-mmm-yy hh:mm xxx'	'15-nov-82 10:30 pst'

Date intervals—Valid formats for date intervals consist of one or more of the following units, starting with the largest unit and ending with the smallest unit:

yy **yrs** *mm* **mos** *dd* **days**

Note

The input format for **yrs** and **mos** can be spelled out in full or specified as singular or plural abbreviations (for example, **years**, **yr**, and **yrs** are all valid input formats).

Ranges for date interval components are shown below:

Date Interval Component	Range
years (yr, yrs)	-800 to +800
months (mo, mos)	-9611 to +9611
days	-8388608 to +8388607

Examples:

```
'5 years'
'8 months'
'14 days'
'5 yrs 8 mos 14 days'
'5 years 8 months'
'5 years 14 days'
'8 months 14 days'
```

Time intervals—Valid formats for time intervals consist of one or more of the following units, starting with the largest unit and ending with the smallest unit:

hh **hrs** *mm* **mins** *ss* **secs**

Note

The input format for **hrs, mins** and **secs** can be spelled out in full or specified as singular or plural abbreviations (for example, **hours**, **hr**, and **hrs** are all valid input formats).

Ranges for time interval components are as follows:

Time Interval	Range
hours (hr, hrs)	-596 to +596

Time Interval	Range
minutes (min, mins)	-35791 to +35791
seconds (sec, secs)	-2147483 to +2147483

Examples:

```
'23 hours'
'38 minutes'
'53 seconds'
'23 hrs 38 mins 53 secs'
'23 hrs 53 seconds'
'28 hrs 38 mins'
'38 mins 53 secs'
'23:38 hours'
'23:38:53 hours'
```

1.3.5.3 International Date Formats

The database may be set to one of five date formats (modes) for the interpretation of dates. This mode is set on a session basis. The II_DATE_FORMAT environment variable described in the *ULTRIX/SQL Operations Guide* can be used to change the date format conventions to accommodate the international date conventions shown below. The modes are:

MODE	Input	Interpreted as
US	default	(as above)
MULTINATIONAL	*mm/dd/yyyy*	*dd/mm/yyyy*
ISO (Multinational)	*mmddyy*	*yymmdd*
SWEDEN/FINLAND	*mm-dd-yyyy*	*yyyy-mm-dd*
GERMAN	*dmmyy*	*dmmyy*
	ddmmyy	*ddmmyy*
	dmmyyyy	*dmmyyyy*
	ddmmyyyy	*ddmmyyyy*

1.3.6 Money

ULTRIX/SQL stores money values as their actual money amount, significant to exactly two decimal places. Thus, ULTRIX/SQL rounds all money values to their amounts in dollars and cents on input and output. Arithmetic operations on the **money** data type retain two-decimal-place precision.

ULTRIX/SQL supports the following range of money values:

$-99999999999999.99 <= *m* <= $99999999999999.99

ULTRIX/SQL displays money values as strings of 20 characters. The display format is:

$*sdddddddddddddd.dd*

where *s* is the sign (- for negative and no sign for positive) and *d* is a digit from 0 to 9.

ULTRIX/SQL accepts money values on input either as character strings or as numbers, as follows:

- Character-string input

  '$*sdddddddddddddd.dd*'

  The dollar sign ($) is optional. The sign defaults to plus (+) if not specified. A cents value of zero (.00) need not be specified.

- Numeric input

  ULTRIX/SQL accepts any valid integer or floating-point number on input as a money value and converts it to the **money** data type automatically.

Note that several environment variables described in the *ULTRIX/SQL Operations Guide* affect the display of money values. The II_MONEY_FORMAT environment variable can be used to set the currency symbol. As indicated above, the default currency symbol is the dollar sign ($). The II_MONEY_PREC environment variable sets the precision with which money values are displayed. The default precision is two decimal digits.

The II_DECIMAL environment variable sets the character used to indicate the decimal point, by default a period (.).

1.3.7 Storage Formats for Data Types

Every data item in an ULTRIX/SQL database is stored in one of the following storage formats:

Notation	Type	Range
char(1) - char(2000)	character	a string of 1 to 2000 characters
c1 - c2000	character	a string of 1 to 2000 characters
varchar(1) - varchar(2000)	character	a string of 1 to 2000 characters
text(1) - text(2000)	character	a string of 1 to 2000 characters
integer1	1-byte integer	-128 to +127
smallint	2-byte integer	-32,768 to +32,767
integer	4-byte integer	-2,147,483,648 to +2,147,483,647

Notation	Type	Range
float4	4-byte floating-point	-1.0e+38 to +1.0e+38 (7 digit precision)
float	8-byte floating-point	-1.0e+38 to +1.0e+38 (16 digit precision)
date	date (12 bytes)	1-jan-1582 to 31-dec-2382 (for absolute dates) and -800 years to 800 years (for time intervals)
money	money (8 bytes)	$-99999999999999.99 to $99999999999999.99

Note 1

Char and **varchar** are preferred to **c** and **text**.

Note 2

The RISC implementation of ULTRIX supports the IEEE standard for floating-point numbers. The **float** type is accurate to 15 decimal precision and the **money** type is accurate to 14 decimal precision (that is, -$dddddddddddd.dd to +$dddddddddddd.dd). Also, floating-point numbers range from -10**256 to +10**256.

Note 3

The following are valid synonyms for the data types in the preceding table: **character** is equivalent to **char**; **integer2** is equivalent to **smallint**; **int** and **integer4** are equivalent to **integer**; **real** is equivalent to **float4**; **double precision** and **float8** are equivalent to **float**.

1.4 Constants

There are two basic types of constants: string and numeric. In addition, there is a special constant, **null**. ULTRIX/SQL also provides system constants to provide data that can help improve query performance. Constants are also known as literals.

Each type of constant is assigned a default data type, but you can assign them another data type if you wish.

1.4.1 String Constants

String constants are represented by a sequence of characters enclosed in apostrophes. Printing characters are represented literally. To represent a non-printing character you must use the hex constant. (The hex constant is only necessary in the Terminal Monitor; in embedded ULTRIX/SQL, any sequence of characters that can be assigned to a host program variable may be assigned to a character string.)

A hex constant is a special kind of string constant. It is represented by an "X" followed by a string enclosed by apostrophes that contains an even number of characters from the set {[A-F], [a-f], [0-9]}.

For example, the following represents the ASCII string "ABC<carriage return>."

```
X'4142430D'
```

SQL string constants do not support the \ octal representation of ASCII.

The default data type for string constants is **varchar**, but they may be assigned without explicit conversion to any of the character data types or the **money** data type.

1.4.2 Numeric Constants

Numeric constants are represented by a sequence of digits, an optional decimal point, and an optional exponent representation. If no decimal point is specified and if the value of the constant is within the legal range, the default is **integer**. Otherwise the default is **float**. Numeric constants may be assigned without explicit conversion to any of the numeric data types or the **money** data type.

1.4.3 The null Constant

The **null** constant may be assigned to any nullable data type.

1.5 Structured Data

This section discusses ULTRIX/SQL tables, columns, rows, correlation names, and groups. It also introduces the sample database used in examples throughout this manual.

1.5.1 Tables

All data in ULTRIX/SQL is stored as *tables*. A table is a named array of values. The array is composed of *columns* (sometimes called *fields* or *attributes*) and *rows* (sometimes called *records* or *tuples*). Table names may not begin with "ii." An example of a table appears in the following figure.

Figure 1-1: "Candidates" Table

NAME	PARTY	AGE	FUNDING
Robbins	Republican	42	1250000
Capetti	Democrat	52	946000
Greenberg	Citizens	48	766000
Hernandez	Democrat	38	987000
Johnson	Independent	46	854000
Chang	Republican	55	1540000

1.5.2 Columns

Each column of a table has a name, which must be a legal ULTRIX/SQL name. All values in any given column have the same storage format (that is, data type and defined width in bytes). The maximum number of columns in a table is 127.

1.5.3 Rows

A row represents an individual record in a table. All rows in a table are of the same width in bytes and they each maintain the same column types. The maximum length of a row is 2000 bytes.

1.5.4 A Sample Database

A sample database is used for examples throughout this reference manual. The name of the database is "empdata," and its description appears in the following table.

Table Name	Column Name	Data Type
employee	eno	**smallint**
	ename	**char**(10)
	age	**integer1**
	job	**smallint**
	salary	**float4**
	dept	**smallint**
dept	dno	**smallint**
	dname	**char**(10)
	mgr	**smallint**
	floor	**integer1**
job	jid	**smallint**
	jtitle	**char**(10)
	lowsal	**float4**
	highsal	**float4**

1.5.5 Correlation Names

Consider the following **select** statement (Example 1):

```
select   employee.eno, employee.ename
from     employee
where    employee.dept = 23;
```

This statement will retrieve employee numbers and names for all employees in department 23. Its **select** - **from** - **where** structure is typical of retrieval statements in SQL.

Now consider the following alternative formulation of the same query (Example 2):

```
select   e.eno, e.ename
from     employee e
where    e.dept = 23;
```

In this second example, "e" is a *correlation name.* A correlation name is also known as a *range variable* because it is used in an SQL statement to "range over" some table. A correlation name is specified, as shown, by its appearance following the table name in a **from** clause (or an **update** clause, in the case of an **update** statement). At any particular point during execution of the statement in question, the correlation name serves to mark a particular row of the specified table as the current row for processing. Statement execution completes when every row of the table has been marked and processed in this way. Thus in the example above, "e" marks each employee record in turn, and the query is complete when all employee records have been processed.

It is not always necessary to introduce a correlation name explicitly; the formulation shown in Example 1 is perfectly legal SQL. However, the correlation name is still present *implicitly,* even in Example 1. The symbol "employee" in that version is actually being used to play two roles: (1) it serves to identify the employee table; (2) it also serves as a correlation name ranging over that table. Note that it is never wrong, and sometimes it is necessary, to introduce correlation names explicitly.

A correlation name can be any sequence of alphanumeric characters acceptable as a name (see the section entitled "Names" earlier in this chapter).

Finally, it is not always necessary to qualify column names explicitly with the correlation name. An unqualified column name (appearing in, for example, a **select** or a **where** clause) is assumed to be implicitly qualified by a table or correlation name appearing in the **from** clause (or **update** clause) on the same syntactic level as that unqualified reference (see the discussion on subqueries in the section entitled "Search Conditions" later in this chapter). Thus, for example, Example 1 of the query above could be simplified to the following:

```
select   eno, ename
from     employee
where    dept = 23;
```

"Eno," "ename" and "dept" are all implicitly qualified by "employee." Likewise, Example 2 could be simplified to the following:

```
select   eno, ename
from     employee e
where    dept = 23;
```

"Eno," "ename" and "dept" are now all implicitly qualified by "e."

Note that, in order to prevent ambiguity, column names must be qualified explicitly when it isn't clear which table the column comes from.

The maximum number of correlation and table names that can be referenced in a single statement is 30 names. Under certain circumstances, the maximum number may be less than 30.

1.5.6 Groups

It is sometimes convenient to think of the rows of a table as being divided up into groups or partitions by the values of some columns of that table. For example, the candidates table presented in the "Tables" section might be grouped by party, to yield the result shown in the following figure.

Figure 1-2 "Candidates" Table, Grouped by Party

NAME	PARTY	AGE	FUNDING
Greenberg	Citizens	48	766000
Capetti	Democrat	52	946000
Hernandez	Democrat	38	987000
Johnson	Independent	46	854000
Chang	Republican	55	1540000
Robbins	Republican	42	1250000

Note that such grouping is purely conceptual—the table is not really rearranged in the database. The grouping is specified *dynamically* by means of a **group by** clause, as follows:

```
select      . . .
from        candidates
group by    party;
```

The purpose of such grouping is generally to allow some set function to be computed for each group. For example, the statement

```
select      party, avg (funding)
from        candidates
group by    party;
```

will retrieve each party name, together with the average funding for that party, from the candidates table.

1.6 Expressions

Expressions are used in ULTRIX/SQL in many contexts—for example, to denote values to be retrieved (in a **select** clause) or compared (in a **where** or **having** clause). ULTRIX/SQL expressions fall into two broad classes: those that involve set functions and those that do not. Most of the rules for forming expressions apply equally to each of the two classes, with the following exceptions:

- The argument to a set function is an expression, but that expression cannot in turn involve any set functions. In other words, no nesting of set functions is permitted.
- Expressions involving set functions can appear only in certain specific contexts.

Note that constants are considered expressions. The sections titled "Scalar Functions" and "Set Functions" provide more information about functions and expressions.

1.6.1 Columns

A column name, explicitly or implicitly qualified, is an expression. Some examples are:

```
employee.ename
```

```
e.ename
```

```
ename
```

1.6.2 Parentheses

An expression can be enclosed in parentheses, such as ('J. J. Jones'), without affecting its meaning except with respect to precedence, as discussed below.

1.6.3 Arithmetic Operations

1.6.3.1 Operators

Expressions of numeric types can be combined arithmetically to produce other expressions. ULTRIX/SQL supports the following arithmetic operators (in descending order of precedence):

+, -	plus, minus (unary)
**	exponentiation
*, /	multiplication, division
+, -	addition, subtraction (binary)

Unary operators group from right to left, and binary operators group from left to right.

Parentheses can force the desired order of precedence. For example,

```
(job.lowsal + 1000) * 12
```

is an expression in which the plus (+) operator is forced to take precedence over the asterisk (*) operator.

A variety of arithmetic checks, such as integer overflow, integer divide by zero, floating-point underflow, floating-point overflow and floating-point divide by zero, can be enabled by specifying the **-x** flag on the **sql** command line. Refer to the **sql** command in Chapter 5.

The plus (+) operator can also be used to concatenate strings. For example,

```
'This ' + 'is ' + 'a ' + 'test.'
```

gives the value:

```
'This is a test.'
```

When used in this fashion, the plus (+) operator behaves exactly like the **concat** function, described in "Arithmetic Operations on Dates."

1.6.3.2 Arithmetic Operations on Dates

ULTRIX/SQL supports a limited set of arithmetic operations on items of the **date** data type:

Addition:

interval	+	interval	->	interval
interval	+	absolute	->	absolute

Subtraction:

interval	-	interval	->	interval
absolute	-	absolute	->	interval
absolute	-	interval	->	absolute

ULTRIX/SQL does not support multiplication or division of date values.

ULTRIX/SQL also enables you to convert date constants into numbers of days relative to an absolute date. For example, to convert today's date to the number of days since January 1, 1900, use the expression:

```
num_days = int4(interval('days', 'today' - date('1/1/00')))
```

To convert back, use the following:

```
(date('1/1/00') + concat(char(num_days), ' days'))
```

where "num_days" is the number of days added to the date constant.

Note that for comparisons, a blank (default) date is less than any interval date. All interval dates are less than all absolute dates. Intervals are converted to comparable units before they are compared. For instance, **date**('5 hours') is greater than **date**('200 minutes'). Note also that dates are stored internally in an absolute format. For this reason, "5:00 pm pst" compares as equal to "8:00 pm est."

Note also that the expression

```
date('1-feb-89') + '1 month'
```

yields March 1. Adding a month always yields the same date in the next month unless there are fewer days in the next month, in which case it yields the last day of the next month. For instance, adding a month to May 31 yields June 30. Similar rules hold for subtraction. Moreover, similar rules apply for adding and subtracting years.

When adding intervals, each of the units is added. For example,

```
date('6 days') + date('5 hours')
```

yields "6 days 5 hrs," while

```
date('4 years 20 minutes') + date('6 months 80 minutes')
```

yields "4 yrs 6 mos 1 hr 40 mins."

When adding or subtracting intervals, or when subtracting absolute dates, overflow or underflow are propagated upward, except that neither will pass from hours to days. ULTRIX/SQL performs operations on the date component (yrs, mos, days) independently of operations on the time component (hrs, mins, secs) of a date interval.

1.6.4 Type Conversion

This section explains how ULTRIX/SQL converts data types when combining expressions.

1.6.4.1 Default Numeric Type Conversion

When two numeric expressions are combined, ULTRIX/SQL converts as necessary to make the storage formats (that is, data types and widths) identical. Thus, the two parts of the resulting expression have the same storage format.

When ULTRIX/SQL operates on an integer and a floating-point number, the integer is converted to a floating-point number before the operation. When ULTRIX/SQL operates on two integers of different sizes, the smaller is converted to the size of the larger. When operating on two floating-point numbers of different sizes, ULTRIX/SQL converts the larger to the size of the smaller number.

When multiplying or dividing a **money** data item by a non-money item (that is, integer or floating-point), ULTRIX/SQL converts the non-money multiplier or divisor to the **money** type prior to calculation.

The following table summarizes the possible results of numeric combinations.

	Integer1	**Smallint**	**Integer**	**Float4**	**Float**	**Money**
Integer1	integer1	smallint	integer	float4	float	money
Smallint	smallint	smallint	integer	float4	float	money
Integer	integer	integer	integer	float4	float	money
Float4	float4	float4	float4	float4	float4	money
Float	float	float	float	float4	float	money
Money	money	money	money	money	money	money

For example, for the expression

```
(job.lowsal + 1000) * 12
```

the first operator (+) combines a **float4** expression (job.lowsal) with a **smallint** constant (1000) that is converted to a **float4** number. The result is a **float4** expression. The second operator (*) combines this **float4** expression with a **smallint** constant (12) that is converted to a **float4** number, resulting in a **float4** expression.

Note that while

```
(job.lowsal + 1000) * 12
```

produces a **float4** expression, the expression

```
float8((job.lowsal+1000)*12)
```

produces a **float** (**float8**) expression. ULTRIX/SQL also provides specific type conversion functions. These are discussed in the section, "Explicit Type Conversion Functions."

1.6.4.2 Numeric Overflow

Numeric overflow can occur when the results of a computation are larger than can be held by the data type in which the computation is performed. For example, in the following statement the calculation on the right-hand side is done in **integer2** arithmetic. If the **integer2** arithmetic results in a value greater than 32767, the largest possible **integer2** value, then overflow will occur.

```
update emp
set integer4col = integer2col * integer2col;
```

You can avoid many common types of overflow by using a function to convert values to a higher precision before performing the calculation. For example,

```
update emp
set integer4col=int4(integer2col) * int4(integer2col);
```

(For more information on the **int4** function, see "Explicit Type Conversion Functions.")

Numeric overflow, underflow (for floating-point calculations) and division by zero are controlled by the **-x** command line flag for the **isql** and **sql** commands. (The **-x** flag is also valid in embedded SQL for the **connect** statement.) ULTRIX/SQL will either continue as if no error occurred, signal an error and abort the query, or signal a warning and continue, depending on how the **-x** flag is set. See the **isql** or **sql** command in Chapter 5 for more information on this flag.

1.6.4.3 Default Character Type Conversion

Whenever a string of type **c** or **char** is put into a column defined as type **text** or **varchar**, all the string's trailing blanks are removed. Conversely, whenever a string of type **text** or **varchar** is put into a column defined as type **c** or **char**, the string is padded with blanks to fill out the column's defined width, if necessary.

1.6.4.4 Explicit Type Conversion Functions

In addition to ULTRIX/SQL default type conversions, many explicit type conversion functions are available. The following explicit type conversion functions can be used:

Name	Operand Type	Result (Format)	Description
c(*expr*)	any	**c**	Converts any value to **c** string.
char(*expr*)	any	**char**	Converts any value to **char** string.
date(*expr*)	**c, text, char, varchar**	**date**	Converts **c, char, varchar** or **text** string to internal date representation.
dow(*expr*)	**date**	**c**	Converts absolute date into its day of week (for example, 'Mon,' 'Tue')
float4(*expr*)	any except **date**	**float4**	Converts non-date expression to **float4.**
float8(*expr*)	any except **date**	**float**	Converts non-date expression to **float.**
hex(*expr*)	**varchar, char, c, text**	**varchar**	Returns the hex representation of the argument string. The result length is 2 times the input string length, as for example, hex('A') - '61' (ASCII) or 'C1' (EBCDIC).
int1(*expr*)	any except **date**	**integer1**	Converts non-date expression to **integer1.**
int2(*expr*)	any except **date**	**smallint**	Converts non-date expression to **smallint.**
int4(*expr*)	any except **date**	**integer**	Converts non-date expression to **integer.**
money(*expr*)	any except **date**	**money**	Converts non-date expression to internal money representation.
text(*expr*)	any	**text**	Converts any value to a **text** string. This function removes trailing blanks, if any, from **c** or **char** string expressions.
varchar(*expr*)	any	**varchar**	Converts any value to a **varchar** string. This function also removes trailing blanks, if any, in **c** or **char** string expressions.

1.6.5 Scalar Functions

Two kinds of functions are provided: *scalar functions* and *set functions.* Scalar functions take as their argument a single-valued expression (or, in some cases, two such expressions). Set functions take as their argument an entire set of scalar values. The present section is concerned only with scalar functions; set functions are described in a following section.

A scalar function reference consists of the function name, followed by a parenthetical expression (or pair of expressions) representing the function arguments. A scalar function reference is an expression. Scalar function references can be nested to any level.

The explicit type conversion functions discussed earlier are scalar functions; the other available scalar functions are described below.

1.6.5.1 Numeric Functions

In addition to the type conversion functions described above, the following numeric functions are available:

Name	Format (Operand and Result)	Description
abs(*n*)	all numeric types and **money**	absolute value of *n*
atan(*n*)	**float**	arctangent of *n*
cos(*n*)	**float**	cosine of *n*
exp(*n*)	**float**	exponent of *n*
log(*n*)	**float**	natural logarithm of *n*
mod(*n*,*b*)	**integer, smallint, integer1**	*n*, modulo *b*, where *n* and *b* are integers
sin(*n*)	**float**	sine of *n*
sqrt(*n*)	**float**	square root of *n*

For instance, the following example gives the exponential of "job.lowsal" as a **float** expression:

```
exp(job.lowsal)
```

1.6.5.2 String Functions

The following functions operate on **c**, **char**, **text**, or **varchar** data. The expressions *c1* and *c2* represent arguments for the various functions. They can represent any of the string types, except where noted. The expressions *len* and *nshift* represent integer arguments.

Name	Format (Operand and Result)	Description
charextract(*c1*,*n*)	any character data type	Returns the *n*th byte of *c1*. If *n* is larger than the length of the string, the result is a blank character.

Name	Format (Operand and Result)	Description
concat(*c1*,*c2*)	**c, text , varchar**	Concatenates one string to another. The result size is the sum of the sizes of the two arguments. If the result is a **c** or **char** string, it is padded to achieve the proper length. To determine the characteristics of concatenating one string to another, see the following chart on string concatenation.
_date(*s*)	any character data type	Returns a 9-character string giving the date *s* seconds after January 1, 1970. The output format is "*dd-mmm-yy*".
left(*c1*,*len*)	any character data type	Returns the leftmost *len* characters of *c1*. If the result is a fixed-length **c** or **char** string, it is the same length as *c1*, padded with blanks. The result format is the same as *c1*.
length(*c1*)	**smallint**	If *c1* is a fixed-length **c** or **char** string, returns the length of *c1* without trailing blanks. If *c1* is a variable-length string, returns the number of characters actually in *c1*.
locate(*c1*,*c2*)	**smallint**	Returns the location of the first occurrence of *c2* within *c1*, including trailing blanks from *c2*. The location is in the range 1 to **size**(*c1*). If *c2* is not found, the function returns **size**(*c1*) + 1.
lowercase(*c1*)	any character data type	Converts all uppercase characters in *c1* to lowercase.
pad(*c1*)	**text** or **varchar**	Returns *c1* with trailing blanks appended to *c1*. For instance, if *c1* is a **varchar** string that could hold 50 characters but only has two characters, **pad**(*c1*) appends 48 trailing blanks to *c1* to form the result.
right(*c1*,*len*)	any character data type	Returns the rightmost *len* characters of *c1*. Trailing blanks are not removed first. If *c1* is a fixed-length character string, the result is padded to the same length as *c1*. If *c1* is a variable-length character string, no padding occurs. The result format is the same as *c1*.
shift(*c1*,*nshift*)	any character data type	Shifts the string *nshift* places to the right if *nshift* > 0 and to the left if *nshift* < 0. If *c1* is a fixed-length character string, the result is padded with blanks to the length of *c1*. If *c1* is a variable-length character string, no padding occurs. The result format is the same as *c1*.
size(*c1*)	**smallint**	Returns the *declared* size of *c1* without removal of trailing blanks.

Name	Format (Operand and Result)	Description
squeeze(*c1*)	**text** or **varchar**	Compresses white space. White space is defined as any sequence of blanks, null characters, newlines (line feeds), carriage returns, horizontal tabs and form feeds (vertical tabs). Trims white space from the beginning and end of the string and replaces all other white space with single blanks. This function is useful for comparisons. The value for *c1* must be a string of variable-length character string data type (not fixed-length character data type). The result is the same length as the argument.
_time(*s*)	any character data type	Returns a 5-character string giving the time *s* seconds after January 1, 1970. The output format is "*hh:mm*" (seconds are truncated).
trim(*c1*)	**text** or **varchar**	Returns *c1* without trailing blanks. The result has the same length as *c1*.
uppercase(*c1*)	any character data type	Converts all lowercase characters in *c1* to uppercase.

The following table indicates the effects of concatenating one string to another, depending on the particular combination of data types of the two strings.

1st String	2nd String	Trim Blanks from 1st?	Trim Blanks from 2nd?	Result Type
c	**c**	Yes	--	**c**
c	**text**	Yes	--	**c**
c	**char**	Yes	--	**c**
c	**varchar**	Yes	--	**c**
text	**c**	No	--	**c**
char	**c**	Yes	--	**c**
varchar	**c**	No	--	**c**
text	**text**	No	No	**text**
text	**char**	No	Yes	**text**
text	**varchar**	No	No	**text**
char	**text**	Yes	No	**text**
varchar	**text**	No	No	**text**
char	**char**	No	--	**char**
char	**varchar**	No	--	**char**

1st String	2nd String	Trim Blanks from 1st?	Trim Blanks from 2nd?	Result Type
varchar	**char**	No	--	**char**
varchar	**varchar**	No	No	**varchar**

The string functions can be arbitrarily nested to achieve other string functions. For example,

```
left(right(x.name, size(x.name) - 1), 3)
```

returns the substring of "x.name" from character positions 2 through 4.

You can also nest string functions within themselves. For example,

```
concat(concat(x.lastname, ','), x.firstname)
```

concatenates "x.lastname" with a comma and then concatenates "x.firstname" with the first concatenation result. Note, however, that the same result can be achieved with the + operator:

```
x.lastname + ',' + x.firstname
```

1.6.5.3 Date Functions

ULTRIX/SQL supports two functions that derive values from absolute dates and one function that derives a value from interval dates. These functions operate on rows that contain date values. The *unit* expression is a quoted string that represents the part of the date to use in the calculation. Legal values are:

Unit	Expression
Second	**seconds, sec, secs**
Minute	**minutes, min, mins**
Hour	**hours, hr, hrs**
Day	**days**
Week	**weeks, wk, wks**
Month	**months, mo, mos**
Quarter	**quarters, qtr ,qtrs**
Year	**years, yr, yrs**

For **date_trunc** and **date_part**, the *date* expression must be an absolute date. The **interval** function accepts only intervals for *date_interval*.

Name	Format (Result)	Description
date_gmt(*date*)	any character data type	Converts an absolute date into the GMT character equivalent with the format "*yyyy_mm_dd hh:mm:ss* GMT."

date_part(*unit,date*)	**integer**	Returns an integer representing one component of the input date. The *unit* parameter represents the desired component. This function is useful in set functions and in assuring correct ordering in complex date manipulation. For example, suppose *date_field* contains the value "23-oct-1985." Then
		`date_part('month',date(date_field))`
		returns a value of 10, and
		`date_part('day',date(date_field))`
		returns a value of 23.
		
		Months are ordered with January set to month 1. Hours are set to a 24-hour clock. Quarters are numbered 1 through 4. Weeks return a number representing the number of the week since the beginning of the year in which the input date falls. Week 1 begins on the first Monday of the year. Dates before the first Monday of the year are considered to be in week 0.

date_trunc(*unit,date*)	**date**	Returns a date value that represents the input *date* truncated to the level of granularity expressed in the *unit*. By using the **date_trunc** function you can group all the dates within the same month or year, and so forth. A value of "1-oct-1985" is returned by:
		`date_trunc('month',date('23-oct-1985 12:33'))`
		A value of "1-jan-1985" is returned by:
		`date_trunc('year',date('23-oct-1985'))`
		
		All truncation takes place in terms of calendar years and quarters ("1-jan," "1-apr," "1-jun" and "1-oct"). If you need to truncate in terms of a fiscal year, simply offset the calendar date by the number of months between the beginning of your fiscal year and the beginning of the *next* calendar year ('6 mos' for a fiscal year beginning July 1, or '4 mos' for a fiscal year beginning September 1):
		`date_trunc('year',date+'4 mos') - mos`
		
		Monday constitutes the starting day for weeks. Note the beginning of a week for an early January date may fall into the previous year.

Name	Format (Result)	Description
interval(*unit*, *date_interval*)	**float**	Converts a date interval into a floating-point constant expressed in the unit of measurement specified by *unit*. The **interval** function assumes that there are 30.436875 days per month and 362.2425 days per year when using the **mos**, **qtrs** and **yrs** specifications.

1.6.5.4 The Ifnull Function

The **ifnull** function allows users to return a fixed value instead of a null value, when a null is encountered. The **ifnull** function is defined as **ifnull**(*v1*,*v2*). The function takes two character or two numeric input arguments, *v1* and *v2*. If the value of *v1* is **not null**, *v1* is returned. If the value of *v1* is **null**, *v2* is returned.

The data type and length of the result are determined by a comparison of the input argument data types and lengths. The data type of the result is the "larger" of the data types of the input arguments, defined as follows:

float8 > **float4** > **integer4** > **integer2** > **integer1**

varchar > **text** > **char** > **c**

The length of the result is taken from the longest of the two input values.

In the following examples, *i2* and *i4* represent variables with data types of **smallint** and **integer**, respectively. Regardless of whether the first argument is **null** or **not null**, both examples return a value with **integer** data type, because *i4* has the larger data type.

```
ifnull (i2, i4)
```

```
ifnull (i4, i2)
```

In the following example, *varchar(5)* and *c(10)* represent variables of the named data types. This example returns a result with the larger data type, **varchar**, and a length of 10:

```
ifnull (varchar(5), c(10))
```

The result is nullable if either argument is nullable. The *v1* value is not *required* to be nullable, though in most applications it would be nullable.

1.6.6 Dbmsinfo( ) Function

The **dbmsinfo( )** function is used to request information from a database. This function queries the database from SQL.

The **dbmsinfo( )** function takes the place of **_username**. This function has the syntax:

dbmsinfo ('*request_name*')

The following request names can be used with **dbmsinfo( )**.

Request Name	Response Description
transaction_state	'1' means in a transaction; '0' means not in a transaction.
_bintim	Returns the current time and date in an internal format, represented as the number of seconds since January 1, 1970 00:00:00 GMT.
_cpu_ms	cpu time for session, in milliseconds
_et_sec	elapsed time for session, in seconds
_dio_cnt	direct I/O requests for session
_bio_cnt	buffered I/O requests for session
_pfault_cnt	page faults for server
dba	ULTRIX/SQL username of the database owner
username	ULTRIX/SQL username of the user currently running ULTRIX/SQL
_version	ULTRIX/SQL version number
database	database name
terminal	terminal address
query_language	'sql'

These request names are case insensitive, and **dbmsinfo( )** will always return a **varchar(24)** as the result. If **dbmsinfo** is given a request name it does not recognize, it will return an empty string.

The query,

```
select x=dbmsinfo('transaction_state')
```

returns a variable-length string containing the answer (for instance, "1").

1.6.7 Set Functions

A set function is a function that operates on an entire column of values, not just a single value. Consider the following example:

```
select   sum (employee.salary)
from     employee
where    employee.dept = 23;
```

This statement will retrieve the total salary for employees in department 23. The argument to the function is the set (column) of employee salary values where the employee department is equal to 23.

The following set functions are supported:

Name	Input Datatype	Format(Result)	Description
count	(any)	**integer**	Count of occurrences
sum	**integer** **float** **money**	**integer** **float** **money**	Summation
avg	**integer** **float** **money**	**float** **float** **money**	Average (sum/count)
max	(any)	same as argument	Maximum value
min	(any)	same as argument	Minimum value

The general syntax of a set function reference takes the form:

set_fun ([**distinct** | **all**] *expr*)

where *set_fun* denotes a set function, *expr* denotes any expression that does not itself include a set function reference (at *any* level of nesting), and the optional key word **distinct** indicates that duplicate values are to be eliminated from the argument before the set function is performed. The key word **all** indicates the default condition, in which duplicate values are *not* eliminated. Note that it makes no sense to use **distinct** in conjunction with the functions **min** and **max**.

1.6.7.1 The count Function

The **count** function includes a special case. The set function reference **count(*)** may be used to count the number of rows in the result table. For example, the statement:

```
select   count(*)
from     employee
where    dept = 23;
```

counts the number of employees in department 23. The argument "*" cannot be qualified by **all** or **distinct**.

Null values are ignored by the set function. Here again, **count(*)** is the exception, since it counts rows rather than columns. Consider the following table:

Name	Exemptions
Smith	0
Jones	2
Tanghetti	4
Fong	null
Stevens	null

Running **count(c1)** will return the value 3, whereas **count(*)** will return 5.

1.6.7.2 Restrictions on the Use of Set Functions

The following restrictions apply to the use of set functions:

- First, as already mentioned, they cannot be nested.
- Second, set function references, or expressions that include such a reference, such as

```
sum (employee.salary) / 25
```

  are permitted only in the context of a **select** or **having** clause. Furthermore, any column names appearing outside such a set function reference (in such a **select** or **having** clause) must have been specified as one of the operands in a **group by** clause at the same syntactic level as that **select** or **having** clause.

If the argument to a set function evaluates to an empty set, then the value returned is as follows:

Set Function	Value Returned
count	zero
sum, avg	null
max, min	null

The **group by** clause allows set functions to be performed on groups of rows, according to the values in specified columns of the rows. See the discussion on "Groups" in the section "Structured Data" for an example of how grouping is used in conjunction with set functions.

1.6.7.3 ifnull and Set Functions

As stated above, the **sum, avg, max** and **min** set functions can return a null value, when the argument to a set function evaluates to an empty set. This can occur even when the column the set function is operating on is **not null**(able). To assure that a set function will never return a null, use the **ifnull** function. The **ifnull** function returns the normal set function result unless that result is null, in which case it returns the second argument to the **ifnull** function.

The following returns -1 if sum(employee.salary)/25 is null:

```
ifnull ( sum(employee.salary)/25, -1 )
```

The following returns 0 if max(s.empno) is null:

```
ifnull ( max(s.empno), 0 )
```

1.7 Search Conditions

Search conditions are used in where and having clauses to qualify the selection of data. Search conditions are composed of predicates of various kinds, optionally combined together by means of parentheses and the logical operators and, or and not. Thus, any of the following is a legal search condition, where search_condition stands for an arbitrary search condition:

predicate
not *search_condition*
search_condition **or** *search_condition*
search_condition **and** *search_condition*
(*search_condition*)

Of the three logical operators, **not** has the highest precedence, followed by **and**, with **or** having the lowest precedence. They group from left to right. The parentheses may be used to control grouping.

There are seven kinds of predicates, each described in its own section below:

comparison predicate
like predicate
between predicate
in predicate
any-or-all predicate
exists predicate
is null predicate

Predicates evaluate to "true," "false" or "unknown." They evaluate to "unknown" if one or both operands are null (the **is null** predicate is the exception). When predicates are combined using logical operators (**and**, **or**, **not**) to form a search condition, the search condition evaluates to "true," "false" or "unknown," as determined by the following tables.

AND	**true**	**false**	**unknown**
true	true	false	unknown
false	false	false	false
unknown	unknown	false	unknown

OR	**true**	**false**	**unknown**
true	true	true	true
false	true	false	unknown
unknown	true	unknown	unknown

Not(true) is "false," **not**(false) is "true," **not**(unknown) is "unknown."

After all search conditions are evaluated, the value of the **where** or **having** clause is determined. The **where** or **having** clause can be "true" or "false" only; "unknown" values are considered "false."

1.7.1 Subqueries

Nesting of queries is accomplished in ULTRIX/SQL by means of a search condition feature known as the *subquery*. A subquery is a *subselect* used in a predicate of a search condition. (See the Section called "Select" in this chapter for more information about subselects.) The search condition containing the subquery can be part of another subquery, or of any data manipulation statement permitting search conditions. Multiple levels of nesting are permitted. Here is an example of a subquery:

```
select   ename
from     employee
where    dept in
         (select dno
         from     dept
         where    floor = 3);
```

The expression in parentheses is the subquery; it evaluates to the set of department numbers for departments on the third floor. The outer query then retrieves the names of employees whose department number is in that set (that is, names of employees who work on the third floor).

Subqueries often take the place of expressions in predicates. Note that subqueries can be used in place of expressions only in the specific instances outlined in the sections on predicate types below.

The preceding example serves to illustrate the concept of *syntactic* level. Briefly, the **select, from**, and **where** clauses in the subquery are considered to be at a different syntactic level from the **select, from**, and **where** clauses in the outer subselect. More generally, two syntactic units within the same statement are considered to be at the same syntactic level if and only if there exists a subselect within that statement such that the two syntactic units are both immediately contained within that subselect (that is, neither one is contained within a subquery nested within that subselect).

The syntax of the subquery is identical to that of the subselect, except for one restriction—expressions in the **select** clause cannot be assigned result column names.

A subquery may include references to correlation names defined (explicitly or implicitly) outside the subquery. For example, the following statement selects names of employees with a salary greater than the average for their department:

```
select   ename
from     employee empx
where    salary
         (select avg (salary)
         from     employee empy
         where    empy.dept = empx.dept);
```

Here the subquery includes a reference to the correlation name "empx" defined in an outer query—that is, at a different syntactic level. Note that the reference "empx.dept" *must* be explicitly qualified here; otherwise it would be assumed to be *implicitly* qualified by "empy." The overall query is evaluated by letting "empx" take each of its permitted values in turn (that is, letting it range over the employee table), and for each such value of "empx," evaluating the subquery. Note that at least one of the correlation names must be explicit in this example (either "empx" or "empy," but not both, could be allowed to default to simply "employee").

1.7.2 Comparison Predicate

A *comparison* predicate takes the form:

expression_1 comparison_operator expression_2

where *comparison_operator* is one of the following:

= equal to
!= not equal to
> greater than
>= greater than or equal to
< less than
<= less than or equal to

Note that the comparison operator "not equal to" may also be indicated by the <> or ^= character combinations. All comparison operators are of equal precedence.

Note

> If a subquery is in the right-hand argument of a comparison predicate, the subquery may return at most one row. If the subquery returns zero rows, the comparison predicate evaluates to "false."

If there is a null on either or both sides of any comparison operator, the expression will evaluate to "false."

1.7.3 like Predicate

The **like** predicate provides the only pattern-matching capability in ULTRIX/SQL for the character data types (**char**, **varchar**, **c**, and **text**). It takes the following form:

columnname [**not**] **like** *pattern* [**escape** *escape_character*]

where *pattern* is a string constant, not a column. The pattern-match characters are the percent sign (%) to denote 0 or more arbitrary characters, and the underscore (_) to denote exactly one arbitrary character.

If the **escape** clause is specified, entering an escape character changes the meaning of the character immediately following the escape character. The escape character may be used with the following characters:

- Pattern matching characters percent (%) and underscore (_). To enter the percent sign (%) or underscore (_) literally, precede it with the escape character.

- The escape character itself. To enter the escape character literally, type it twice.

- Square brackets ([ ]). When preceded by an escape character, square brackets provide special pattern matching capabilities. You can specify a series of individual characters or a range of characters separated by a hyphen (-).

The following examples illustrate some uses of the **like** predicate's pattern matching capabilities.

- To match any string starting with "a":

  ```
  name LIKE 'a%'
  ```

- To match any string starting with A through Z:

  ```
  name LIKE '\[A-Z\]%' ESCAPE '\'
  ```

- To match any two characters followed by "25%":

  ```
  name LIKE '__25\%' ESCAPE '\'
  ```

- To match a string starting with a backslash:

  ```
  name LIKE '\%'
  ```

  Because there is no escape clause, the backslash is taken literally.

- To match a string starting with a backslash and ending with a percent character:

  ```
  name LIKE '\\%\%' ESCAPE '\'
  ```

- To match any string starting with 0 through 4, followed by an uppercase letter, then a left bracket ([ ), any two characters and a right bracket ( ]):

  ```
  name LIKE '\[01234\]\[A-Z\][__]' ESCAPE '\'
  ```

1.7.4 between Predicate

The operators **between** and **not between** have the following meanings:

Operator	**Meaning**
y **between** *x* **and** *z*	$x <= y,\ y <= z$
y **not between** *x* **and** *z*	**not** (*y* **between** *x* **and** *z*)

In the foregoing, *x*, *y* and *z* are expressions. Subqueries may not be substituted for any of the expressions.

1.7.5 in Predicate

The operators **in** and **not in** (followed by a parenthesized list of expressions) are defined as follows:

Operator	Meaning
y **in** (*x*,, . . ., *z*)	*y* = *x* **or** . . . **or** *y* = *z*
y **not in** (*x*,, . . ., *z*)	**not** (*y* **in** (*x*,, . . ., *z*))

In the preceding table, *x*, *y* and *z* are expressions and may not be subqueries. If there is only one expression in the list, the parentheses are optional.

Another version of the **in** predicate takes the form:

expression [**not**] **in** (*subquery*)

The subquery must contain a reference to exactly one column in its **select** clause.

1.7.6 any-or-all Predicate

An *any-or-all* predicate takes the form:

any-or-all_operator (*subquery*)

The subquery must have exactly one expression in its **select** clause (so that it evaluates to a set of scalar values, not a set of rows). The any-or-all operator is one of the following:

=any	**=all**
!=any	**!=all**
<any	**<all**
<=any	**<=all**
>any	**>all**
>=any	**>=all**

It is permissible to include a space between the comparison operator and the key word **any** or **all**.

Let a dollar sign ($) denote any one of the comparison operators =, !=, < , <=, >, >=. Then the predicate *x* **$any** (*subquery*) evaluates to "true" if, and only if, the comparison predicate *x* **$** *y* is true for at least one value *y* in the set of values represented by *subquery*. If the subquery is empty, the **$any** comparison fails (evaluates to "false").

Likewise, the predicate *x* **$all** (*subquery*) is true if, and only if, the comparison predicate *x* **$** *y* is true for all values *y* in the set of values represented by *subquery*. If the subquery is empty, the **$all** comparison succeeds (evaluates to "true").

The operator =**any** is equivalent to the operator **in**. For example:

```
select   ename
from     employee
where    dept in
         (select  dno
         from     dept
         where    floor = 3);
```

may be rewritten as:

```
select   ename
from     employee
where    dept = any
         (select  dno
         from     dept
         where    floor = 3);
```

The operator **some** is a synonym for the operator **any** and would appear as:

```
select   ename
fromemployee
where    dept = some
         (select  dno
         fromdept
         where    floor = 3);
```

1.7.7 exists Predicate

An **exists** predicate takes the form:

[not] exists (*subquery*)

It evaluates to "true" if, and only if, the set represented by *subquery* is not empty. For example, the following statement selects names of employees who work on the third floor:

```
select   ename
from     employee
where    exists
         (select  *
         from     dept
         where    dno = employee.dept
         and      floor = 3);
```

It is typical, but not required, for the subquery argument to **exists** to be of the form, **select * . . .**

1.7.8 is null Predicate

The **is null** predicate takes the form:

is [not] null

The statement *x* **is null** is true if, and only if, *x* is the null value. Because you can't use the equal sign (=) comparison operator to test for a null result, you must use the **is null** predicate to find out whether an expression is null.

1.8 Data Manipulation Statements

The ULTRIX/SQL data manipulation statements are **select, update, delete** and **insert**.

1.8.1 Select

The general syntax of **select** is:

> *subselect*
> {**union** [**all**] *subselect*}
> [**order by** *result_column* [**asc** | **desc**]{, *result_column* [**asc** | **desc**]}]

where:

- Each *subselect* has the syntax shown following this list.
- The clause *subselect* **union all** *subselect* yields all results that either subselect would yield if run individually. If **all** is not specified in the **union**, duplicate rows are removed from the result.
- Corresponding data types across subselects must be coercible into a common data type—that is, they must be either all character types or all numeric types.
- All subselects in a **select** have the same number of columns in their result table.
- Each *result_column* in the **order by** clause consists of either a result column name or an integer constant in the range 1 - *n*, where *n* is the number of columns in the result table of each of the subselects. If the statement contains a single subselect, the column must be one of the columns of the result table. If the statement contains more than one subselect, the *result_column* name is derived from the first subselect. Because the column name in an **order by** clause refers to a column in the *result* table, it may not be qualified by a table or correlation name.
- The optional key words **asc** and **desc** specify ascending and descending sort sequence, respectively. If neither is specified for a particular column, **asc** is assumed by default.

The syntax for *subselect* is as follows:

select [**all**|**distinct**] *expression* [**as** *result_column*] {, *expression* [**as** *result_column*]}
from *table* [*corr_name*] {, *table* [*corr_name*]}
[**where** *search_condition*]
[**group by** *column* {, *column*}]
[**having** *search_condition*]

The key word **distinct** indicates that duplicate rows are to be eliminated. The key word **all**, the default condition, causes duplicate rows to remain.

The *expressions* in the **select** clause can be any expressions constructed in accordance with the rules for expressions (refer to the section titled "Expressions" in this chapter). They may also take one of the following forms:

*correlation_name.**	All the columns of the table denoted by *correlation_name*
*table.**	All the columns of *table*
*	All the columns of all the tables named in the **from** clause. This cannot be part of a comma-separated list; it must be the only element in the select list.

A *result_column* may be assigned to any *expression* that denotes a single column in the result table (that is, where *expression* does not use the "*" syntax). *Result_column* appears as the column name for the column in the result table resulting from the expression. If *result_column* is not specified and the *expression* is simply one column from the source table, then the result column name is the same as the source table column name used in the expression. If the *expression* is a scalar or aggregate function or involves more than one column, and the *result_column* is not specified, then ULTRIX/SQL provides a default result column name ("col1," "col2," . . .). The result column, whether default or explicit, is also used in the **order by** clause.

The *columns* in the **group by** clause are names of columns from the tables identified in the **from** clause. They may be qualified by a **having** clause.

From a conceptual standpoint, the subselect is evaluated in the following manner. First, the Cartesian product of all tables identified in the **from** clause is formed. (Cartesian products are defined in the section titled "Cartesian Product.") From that product, rows not satisfying the search condition specified in the **where** clause are eliminated. Next, the remaining rows are grouped in accordance with the specifications of the **group by** clause. Groups not satisfying the search condition in the **having** clause are then eliminated. Finally, the expressions specified in the **select** clause are evaluated. If the key word **distinct** has been specified, any duplicate rows are eliminated from the result table.

Note

> Bear in mind that the foregoing explanation is purely conceptual in nature. Actual evaluation normally does *not* proceed in precisely the manner described, but instead uses some more efficient method, as determined by the ULTRIX/SQL optimizer.

If the subselect includes a **group by** clause, each expression in the **select** clause must be *single-valued per group*. That is, the only data items permitted in such an expression are the following:

- constants
- grouping columns
- set function references

As usual, however, such terms can be combined using arithmetic operations, can be the arguments to scalar functions, and so forth.

If the subselect includes a **having** clause, each expression in that clause must also be single-valued per group. If the **group by** clause is omitted in a subselect with a **having** clause, the entire table is considered to be a single group.

The result of a **select** statement is the union of the results of all subselects in that statement, ordered in accordance with the specifications of the optional **order by** clause. Duplicate rows are always eliminated if either **union** or **distinct** is specified. If **order by** is not specified, the rows of the result appear in unpredictable order.

The following is an example of a **select** statement:

```
select   eno
from     employee
where    age > 45
union
select   mgr
from     dept
where    floor = 3
order    by 1;
```

1.8.2 Update

The general syntax of **update** is as follows:

> **update** *table* [*corr_name*]
> **set** *column* = *expression*{, *column* = *expression*}
> [**where** *search_condition*]

Example:

```
update   employee
set      job = 27,
         salary = salary * 1.1
where    job = 25;
```

1.8.3 Delete

The general syntax of **delete** is:

> **delete**
> **from** *table* [*corr_name*]
> [**where** *search_condition*]

Example:

```
delete   from employee
where    job = 0;
```

1.8.4 Insert

The general syntax of **insert** is:

> **insert**
> **into** *table* [(*column* {, *column*})]
> *source*

where *source* either is a subselect or takes the form:

values (*expression* {, *expression*})

Expressions used in the **values** clause can be only constants, scalar functions on constants, or arithmetic operations on constants.

Examples:

```
insert
into      dept (dno, dname, mgr)
values    (38, 'Purchasing', 21458);

insert
into      employee (eno)
          select   mgr
          from     dept
          where    dname = 'newdept';
```

1.9 Relational Concepts

1.9.1 Expressing Relational Operators in SQL

One of the first query languages proposed for use in relational systems was based on the *relational algebra.* Even though no purely algebraic language is in current use, some of the algebraic operators have become a standard part of the terminology of relational systems. The most familiar of these are:

- Projection
- Restriction
- Cartesian product
- Join

This section shows how these operators are expressed in ULTRIX SQL. For illustration, this section uses the tables "employee," "dept" and "job" defined in the earlier section, "A Sample Database."

1.9.1.1 Projection

Projection is an operator that constructs a "vertical section" of an existing table by taking a subset of its columns. For example, the theoretical statement,

```
project employee on (ename, age)
```

specifies a table consisting of the "ename" and "age" columns of the "employee" table.

The **select** clause in SQL corresponds to projection. For example, the statement "project employee on (ename, age)" is expressed in SQL as:

```
select    ename, age
from      employee;
```

1.9.1.2 Restriction

Restriction constructs a "horizontal section" of a table by taking those rows that satisfy a specified condition. For example, the theoretical statement,

```
restrict employee on (age > 40)
```

defines a table consisting of all rows in "employee" for which the value in "age" is greater than 40.

The **where** clause of an SQL statement corresponds to restriction. For example, "restrict employee on (age > 40)" is expressed in SQL as:

```
select   *
from     employee
where    age > 40;
```

1.9.1.3 Cartesian Product

The Cartesian product of two tables (for instance, A and B) is a table (denoted by A*B) consisting of all concatenations of rows from A with rows from B. That is, each row *t* in A*B is of the form:

$$t = ab$$

where *a* is a row from A, and *b* a row from B, and every distinct pair (*a*,*b*) produces a row in A*B.

For example, "employee*job" is a table consisting of all concatenations *ej*, where *e* is a row from "employee" and *j* is a row from "job."

The Cartesian product is easily expressed in SQL with the **select** statement. For example, the theoretical "employee*job" is expressed in SQL as:

```
select   *
from     employee, job;
```

1.9.1.4 Join

The join operator constructs a table out of two existing tables by collecting all pairs of rows such that each pair satisfies some condition. When the condition is equality between columns from the rows, the operator is called an *equijoin.* For example, the following theoretical statement would be an equijoin:

```
join employee with job on (job of employee = jid of job)
```

By contrast, the following theoretical statement is a join, but not an equijoin:

```
join employee with job on (100*(age of employee) > lowsal of job)
```

A join is equivalent to a combination of Cartesian product followed by a restriction. For example, the second join in the previous paragraph is equivalent to a theoretical formulation:

```
restrict (employee*job) on (100*(age of employee) > lowsal of job)
```

Because SQL allows Cartesian product and restriction to be combined in a single query, joins are easily expressed. The two theoretical examples of joins given above are expressed in SQL as follows:

```
select   employee.*, job.*
from     employee, job
where    employee.job = job.jid;

select   employee.*, job.*
from     employee, job
where    100*employee.age > job.lowsal;
```

1.9.2 Nulls and Defaults

Null represents an unknown or absent value. ULTRIX/SQL gives you the option of having null assigned automatically in a given column when no value is specifically assigned. Null is *not* a value such as zero, a blank, or an empty string.

Nulls are useful if you want to take an aggregate on a column, but don't want unknown or inapplicable values to affect the aggregate. For example, if there is a column "age" in the "employee" table, and if you want to run an aggregate on that column to determine the average age of the employees, you want to make sure that any ages that have not been entered do not count as zeros. If ages that have not been entered are given the value null rather than zero, they will not be counted when the aggregate is run.

If you choose not to allow a column to contain the null value, ULTRIX/SQL also lets you choose whether you want a default value (zero, blank, or empty) assigned to that column. If you do not allow either a null or a default value to be assigned, then the user will be forced to enter a value in the column to avoid an error message. Disallowing nulls and defaults is a good way to make sure that all columns are filled in, in cases where this is appropriate.

Set functions, with the exception of **count()**, return null for an aggregate over an empty set, even when the aggregate includes columns which are not nullable. (Note that **count()** returns 0.) In the following example, **select** returns null, since there are no rows in **tbl**.

```
create table tbl (coll integer NOT NULL);
select max(coll) as x from tbl;
```

To eliminate this condition, you could use the **ifnull** function. For example,

```
select IFNULL(max(coll),0) as x from tbl;
```

will return zero (0).

You determine whether to allow nulls and defaults in a column at the time you create the table with the **create table** command.

1.10 Transactions

A *transaction* in ULTRIX/SQL is defined as one or more SQL statements that are to be processed as a single, indivisible database action. Transactions are atomic units of consistency and concurrency in the ULTRIX/SQL multi-user database environment. None of the effects on a database of one user's transaction is visible to other users' transactions until the transaction is committed. When the transaction is committed, all of its effects are written permanently to the database, and the effects become available to the transactions of other users.

Concurrency control in ULTRIX/SQL insures that simultaneously executing transactions do not interfere with each other in ways that could compromise the atomic status of a transaction. Deadlock is a possible consequence of transaction concurrency control, and deadlock is handled by the ULTRIX/SQL transaction processing system. (For a definition of deadlock, see the discussion titled "Transaction Rollback" later in this section.)

Transactions are committed or rolled back under user control. Transactions can also be rolled back under system control in cases of deadlock. Single statements, both inside and outside a transaction, can be rolled back under system control in cases of deadlock, timeout or error conditions (for instance, a replace operation that generates a duplicate key in a table that has unique keys). Single statements can be rolled back under user control in the case of interrupts.

1.10.1 Transaction Control Statements

The following transaction-controlling statements are available:

- The **commit** statement ends a transaction block and commits the transaction's effects to the database.
- The **rollback** statement terminates a transaction in progress and undoes the effects of all processed statements.

1.10.2 Committing Transactions

A transaction is *committed* when its updates to the database are written. Committing a transaction occurs at the end of the transaction. Before ULTRIX/SQL commits a transaction, none of its updates to the database are available to other users, and the transaction can be rolled back without causing inconsistency or propagating undesirable rollbacks of other transactions. After the transaction is committed, however, its effects in the database are considered permanent and are visible to other transactions.

A transaction is committed explicitly with the **commit** statement. If a user **rollback** command or system-generated rollback on deadlock terminates the transaction before a **commit** command is processed, then the transaction is rolled back, and all its effects on the database are backed out.

1.10.3 Transaction Rollback

At any time before a **rollback** statement commits a transaction, the transaction can be rolled back under user or system control. All effects of the transaction on the database are undone, and no other transactions in progress are adversely affected.

Transactions can be rolled back in either of the following ways:

- User rollback—The statement **rollback** causes immediate termination of a transaction in progress.
- System abort—Deadlock is a situation that may arise during concurrent execution of transactions, when each of two transactions is attempting to update a part of the database that the other transaction is currently using. Each transaction must wait for the other to release a part of the database (for instance, a table or a data page) before it can perform its own updates. Each transaction requires what the other transaction owns, and neither transaction will release the part of the database it currently has until it gets the other part it needs. Because of this standoff, neither transaction can proceed.

  ULTRIX/SQL detects this situation when it occurs and chooses one transaction to roll back in order to end the deadlock. An error number or status code (4700) is returned to the user to indicate rollback on deadlock. The user may then restart the transaction, if desired.

1.10.4 Interrupt and Timeout Handling in Transactions

The transaction processing system in ULTRIX/SQL recognizes the interrupt signal **Control-C**. This has a distinct effect on transaction processing.

A **Control-C** received by the Terminal Monitor during multi-statement transaction processing causes ULTRIX/SQL to abort automatically the latest statement of the transaction. The transaction remains uncommitted and can be continued in normal fashion. This action can take place *only once for a given transaction*; subsequent **Control-C** characters are ignored unless a new statement is added to the multi-statement transaction since the last **Control-C**. The transaction must eventually be terminated in normal fashion, either with a **commit** or **rollback**.

A timeout condition detected while waiting for a lock (see **set lockmode** command) causes an error status (4702) to be returned to the user, and otherwise behaves as if a **Control-C** had been received from the application.

1.10.5 SQL Transaction Semantics

Every SQL database query either begins or is added to an existing multi-query transaction. An SQL transaction is started at the execution of the first SQL statement. Subsequent statements (for instance, **select, insert, update, delete**) accumulate as part of that transaction. The transaction is not committed until a **commit** statement is issued. Statements which cannot be issued within a transaction, such as the **set lockmode** statement, can be executed if no other SQL statements have been executed since the last **commit**.

Queries issued between **commits** will accumulate as part of the transaction and locks on data touched by each query will be held until the next **commit** statement. Even read locks, associated with **select** statements, will accumulate and be held until **commit** time.

1.11 Database Procedures

Database procedures are a collection of statements managed as objects by ULTRIX/SQL as part of the database definition. Procedures provide strong benefits for the user. They enhance performance by reducing the amount of communication between the application and the database management system. They provide the Database Administrator (DBA) with an extra level of control over data access and modification. Also, one procedure can be used in many applications in a database, which reduces coding time.

1.11.1 Using Database Procedures

Procedures can be created or dropped in the ULTRIX/SQL Terminal Monitor or within embedded ULTRIX/SQL. Procedures can only be executed from within embedded ULTRIX/SQL.

A procedure may include data manipulation statements (such as **select** or **insert**) as well as control flow statements (such as **if** and **while**) and the status statements, **message** and **return**.

When you create and use database procedures, there are several considerations to remember:

- Within a database procedure, all object references are resolved when a procedure is created. This means that if a procedure references a public table when it is created, the procedure will always use that table, even if executed by a user having a private table with an identical name.

- All referenced objects must exist at the time the procedure is created and when it is executed. Between the time of creation and the time of execution, you can modify, reorder, or drop and recreate objects such as tables and columns without affecting the procedure definition. However, if an object is redefined in a way that invalidates the procedure definition, then the definition must be dropped and recreated. An example of this is a column whose data type is changed from numeric to string.

- The procedure's query execution plan is created when the procedure is created. If the procedure is modified in a way that invalidates the plan, then the plan is recreated at the next invocation of the procedure.

The following is an example of a database procedure called "move_emp." This example accepts an employee ID number as input. The employee matching that ID is moved from the "employee" table and added to the "emptrans" table. Both tables are inaccessible to users except through the procedure. When the procedure is invoked, the executing application passes a single integer parameter.

```
CREATE PROCEDURE move_emp (id INTEGER NOT NULL) AS
BEGIN
    INSERT INTO emptrans
        SELECT *
            FROM employee
            WHERE id = :id;
    DELETE FROM employee
        WHERE id = :id;
END;
```

1.11.1.1 Permissions on Procedures

A procedure is owned by the person who creates it. If the creator is the Database Administrator (DBA), then the procedure is public and available to any user having the DBA's permission. A procedure created by any other user is private to that user. If the DBA and a user have identically named procedures, the user has access only to the private procedure.

Procedures provide the DBA with greater control over database access. The DBA can grant a user permission to execute a procedure even if the user has no direct access to the underlying tables. In this way, the DBA controls exactly what operations a user can perform on a database.

The DBA uses the following statement to grant permissions to users:

> **grant execute**
> **on procedure** *procedure_name* **to** *user_list*

1.11.1.2 Error Handling

Unless the procedure programmer provides explicit error handling mechanisms, either within the procedure itself or within the calling application, the default action is to continue to the next statement when an error occurs.

Database procedures make use of the control flow statements, **if** and **while**, and two built-in variables, **iirowcount** and **iierrornumber**, to process errors. An application that invokes a database procedure must use the SQL Communications Area (SQLCA) to process errors occurring inside the database procedure. The variables **iirowcount** and **iierrornumber** are only available within the database procedure. (Refer to the *ULTRIX/SQL Reference Guide to Embedded SQL* for information about using the SQLCA.)

The variable **iirowcount** is an integer that indicates the number of rows affected by the last executed SQL statement. If the statement was not a statement that affects rows or if an error occurred, then **iirowcount** is set to -1. If the statement was a row-affecting statement, but no rows were affected, then the value of **iirowcount** is set to 0. The initial value of **iirowcount** is 0.

The variable **iierrornumber** is an integer that holds the error number associated with an error occurring during the execution of a statement. If no error occurs, the value of **iierrornumber** is set to 0. The error number is a positive number and its initial value is 0.

The execution of each statement sets the value of **iierrornumber** either to zero (no errors) or an error number. In order to check the execution status of any particular statement, **iierrornumber** must be examined immediately after the statement's execution.

Errors occurring in **if, while, message,** and **return** statements do not set **iierrornumber**. However, any errors that occur during the evaluation of the condition of an **if** or **while** statement terminate the procedure and return control to the calling application.

1.11.1.3 Message Handling

Database procedures use the **message** statement to display text on the screen while executing. It is possible to provide alternative instructions for message processing using the **whenever** statement within embedded SQL. Refer to the *ULTRIX/SQL Reference Guide to Embedded SQL* for information about using the **whenever** statement and processing procedure messages.

1.11.2 Creating and Executing a Procedure

1.11.2.1 Creating a Procedure

A database procedure can be created within embedded ULTRIX/SQL. The syntax for the statement is:

> [**create**] **procedure** *proc_name*
>   [(*param_name* [=] *param_type* { , *param_name* [=] *param_type*})]
>   =|**as**
>   [*declare_section*]
> **begin**
>   *statement_list*
> **end;**

The parameters in this syntax have the following definitions:

- *proc_name* is the name of the procedure to be created
- *param_name* is the name of the procedure parameter
- *param_type* is the procedure parameter's type

Procedure parameters are treated as local variables in the procedure body, although they have an initial value assigned when the procedure is invoked. You can also assign values to procedure parameters within the body of the procedure. (Local variables are discussed below.)

All parameter types may have the **null** or **default** clauses. For example, the following procedure fragment accepts three parameters: a non-null integer, a varying-length string and a date:

```
CREATE PROCEDURE eval_emp (id INTEGER NOT NULL,
    comment VARCHAR(100),
    meeting DATE NOT NULL) AS . . .
```

The *declare_section* declares a list of local variables that can be referenced within the procedure. The syntax for this statement is:

declare
var_name { ,*var_name*} [=] *var_type*;
{*var_name* { ,*var_name*} [=] *var_type*;}

The parameters in this syntax have the following definitions:

- *var_name* is the name of the local variable.
- *var_type* is the type of the variable.

Variable names must be unique within the procedure. If a variable is nullable, it is initialized to null. If a variable is not nullable, it is initialized to the default value.

You can substitute local variables and procedure parameters for any constant value in statements in the procedure body. A preceding colon (:) is only necessary if the referenced name could be misinterpreted as an SQL column name. For example, if a procedure parameter and a referenced column (in a procedure statement) have the same name, the parameter must be preceded by a colon. The following example illustrates this rule.

In this example, the procedure retrieves the name of an employee who matches an employee ID. Both the employee ID column and the procedure parameter are named "id." The colon in the **where** clause distinguishes the column from the parameter.

```
CREATE PROCEDURE name_of_emp (id INTEGER NOT NULL) AS
DECLARE
    name CHAR(50) NOT NULL;
BEGIN
    SELECT fname + ' ' + lname
        INTO :name
        FROM employee
        WHERE id = :id;
    MESSAGE :name;
END;
```

The *statement_list* may include local variable assignments and any of the following statements:

insert	**rollback**	**while**
delete	**select**	**return**
update	**if**	**message**
commit		

You cannot issue any data definition statements, such as **create table**, from inside a database procedure.

Refer to the statement summary in Chapter 2 for detailed information about the syntax of the **create procedure** statement.

1.11.2.2 Executing a Procedure

Procedures are invoked from within an embedded ULTRIX/SQL application. The statement that invokes a procedure is **execute procedure**. You can execute a procedure dynamically by specifying the **using** clause in an **execute procedure** statement. However, you cannot use either of the dynamic ULTRIX/SQL statements, **execute** or **execute immediate**, to execute a database procedure. Nor can you invoke a procedure interactively or from inside another procedure. Refer to the *ULTRIX/SQL Reference Guide to Embedded SQL* for information about executing a database procedure.

1.11.3 Dropping a Procedure

Dropping a procedure removes the procedure's definition from the database. You must be the owner of a procedure to **drop** a procedure. Procedures may be dropped within an embedded ULTRIX/SQL application. You cannot drop a procedure from inside another procedure.

The syntax of the statement is:

drop procedure *proc_name*

The parameter *proc_name* is the name of the procedure you want to drop.

The statement takes effect immediately. Executions of the procedure in progress, invoked by other users, continue until they are completed. However, no additional references to the procedure are allowed.

1.12 Multi-File System Databases

In order to accommodate large databases within a finite computer system, ULTRIX/SQL enables users to locate the user tables of a single database on more than one file system. Merely by establishing names for discrete areas of a given disk, an ULTRIX/SQL system administrator can preserve the usefulness of an ULTRIX/SQL database, even when it becomes extremely large.

1.12.1 ULTRIX/SQL Locationnames and Areas

A *locationname* is a label that denotes an ULTRIX/SQL directory. These labels are independent of the directory structure. In the ULTRIX operating system, an area would be defined as a directory or sub-directory (for instance, /usr/cormac/new or . . . /mydb/other).

Each *locationname* maps to exactly one area; however, many different *locationnames* can map to the same area. *Locationnames* follow the ULTRIX/SQL naming convention: they must begin with a letter or an underscore (_), and the maximum length is 32 characters. The area designation can be up to 255 characters and must follow the ULTRIX syntax for directory names. Areas and *locationnames* are specified with the **accessdb** command, described in the *ULTRIX/SQL Operations Guide.*

You can use *locationnames* in the **createdb** and **finddbs** utilities, as well as in the **create table, create index** and **modify** commands. If a *locationname* is not specified in a utility or SQL command, then the appropriate default is used. Host language programs using SQL can be written in a manner independent of the system configuration, because all references to database directories can be *locationnames*. Each installation has a set of default *locationnames*. These are **ii_database, ii_journal** and **ii_checkpoint**. These *locationnames* map to the environment variables II_DATABASE, II_JOURNAL and II_CHECKPOINT, respectively.

1.12.2 Assigning Database Tables to Single Areas

As mentioned above, ULTRIX/SQL assigns a table or index in a database to a default *locationname* unless it is otherwise specified on the **create table** or **create index** statement. However, if disk space on the default file system that stores the database becomes too full, the table can be relocated to another file system.

1.12.2.1 Relocating the Database User Tables

The process of relocating a database's user tables to a different device requires four steps:

1. Make sure a valid ULTRIX directory exists for the new database locations.

2. The second step is executed by the ULTRIX/SQL System Administrator. The System Administrator uses the **accessdb** command (described in the *ULTRIX/SQL Operations Guide* and *ULTRIX/SQL Database Administrator's Guide*) to create the new *locationnames*, creating the mapping to directories within or outside the ULTRIX/SQL installation area.

3. The ULTRIX/SQL System Administrator uses the **accessdb** command to *extend* the database to the additional directories by assigning the requisite *locationnames* to tables and indexes.

4. The fourth step is executed by the ULTRIX/SQL user who is the table's owner. Use the following form of the **modify** command (described in Chapter 2) to relocate the user table to a new location:

    modify *tablename* **to reorganize with**
    location = (*locationname* {, *locationname*})

1.12.2.2 Multi-Location Tables

Tables and indexes may also be physically partitioned across multiple areas. A table may be assigned to multiple locations when it is created (using the **create table** or **create index** statement) by way of the **with location** = (*location-list*) clause. For example:

```
create table large (wide varchar (2000))
with location = (location1, location2, location3);
```

The specified locations must already exist and be mapped to directories. (See the **accessdb** description in the *ULTRIX/SQL Operations Guide*.)

Alternatively, a table may be spread over several locations, using the **modify to reorganize** the statement:

```
modify large to reorganize with location = (location1, location2,
    location3);
```

A table, or part of a table, may be relocated to a corresponding location or set of locations by using either of the following **modify to relocate** statements:

```
modify large to relocate
    with     oldlocation = (location1, location2, location3),
             newlocation = (location4, location5, location6);

modify small to relocate
    with     oldlocation = (location1),
             newlocation = (location2);
```

The difference between **modify . . . to relocate** and **modify . . . to reorganize** is that with the **relocate** option, the data from each area in the old location list is moved "as is" to the corresponding location in the *newlocation* list. For example:

```
modify medium to relocate with
    oldlocation = (location1, location2),
    newlocation = (location3, location4);
```

The data for table **medium** in location1 is moved to location3, and the data in location2 is moved to location4. The number of locations in the *oldlocation* list must be equal to the number of locations in the *newlocation* list.

A portion of a table may be relocated by specifying only certain locations in the location lists. For the following example, assume that table "large" is currently assigned to location1, location2 and location3. Then,

```
modify large to relocate with
    oldlocation = (location3),
    newlocation = (location5);
```

will only relocate the table data that resides in location3, leaving location1 and location2 unchanged.

Modify . . . to relocate with only one location named in the location lists is analogous to the **relocate** statement.

With the **reorganize** option, the table is not only moved, but is also reorganized. That is, a table that is spread across three locations can be reorganized to be spread across only two or five locations. You do not need to specify the old locations for the **reorganize** form of **modify**. The entire table is reorganized. The only parameter in the **with** clause that is accepted is the **location** = (*locationname* [,*locationname* . . .]) clause.

The algorithm for spreading a table or index across multiple locations is very simple (that is, efficient) from an internal standpoint, but may be a bit confusing from an external point of view.

If a table is to be spread over three locations, as in the following example,

```
create table large (wide varchar(2000),
    with location = (location1, location2, location3);
```

then as rows are added to the table, they will be added to each location in 16-page (approximately 32-Kilobyte) chunks. When the first 16 blocks are filled in location1, the following 16 pages of data are put in location2 and the next 16 pages are put in location3. Then the pattern starts over again with location1.

If it is not possible to allocate 16 full pages in an location when it is that location's turn to be filled, the table is determined to be out of space, even if there is plenty of room in the table's other locations.

ULTRIX/SQL Statements 2

2.1 Introduction

The ULTRIX/SQL structured query language (SQL) consists of statements that perform a range of functions for data definition, data manipulation and database administration. This chapter presents these statements individually, describing each statement's purpose, syntax and use.

This and other chapters of the *ULTRIX/SQL Reference Manual* provide the definitive description of ULTRIX/SQL functions for those readers who have been referred from another manual. Chapter 3 describes how to use these functions in interactive mode, using the ULTRIX/SQL Terminal Monitor. For complete information about the use of ULTRIX/SQL within a host language program, consult the documentation for your ULTRIX/SQL preprocessor.

2.2 commit

2.2.1 Purpose

Commit the current transaction.

2.2.2 Syntax

commit [**work**]

2.2.3 Description

This statement commits the current transaction. Once committed, the transaction cannot be aborted, and all changes to the database become visible to other users through use of the **select** statement. Once executed, the current transaction is terminated; a new one is automatically started on execution of the next SQL statement. Any open cursors are closed and all locks are released.

The optional word **work** has no effect. It is included for compatibility with other implementations of SQL.

2.3 copy

2.3.1 Purpose

Copy data into/from a table from/into a file.

2.3.2 Syntax

copy [**table**] *tablename* (*columnname* = *format* [**with null** [(*value*)]]
{, *columnname* = *format* [**with null**[(*value*)]]}) **into** | **from** '*filename*'
[**with** *with_options_list*]

The *with_options_list* consists a comma-separated list of any of the following items:

on_error = terminate | continue
error_count = *n*
rollback = enabled | disabled
[**log** = '*filename*']

2.3.3 Description

The **copy** statement moves data between ULTRIX/SQL tables and standard files. **Table** is a key word and must be typed as shown when used. *Tablename* is the name of an existing table. In general, *columnname* identifies a column in the table. *Format* indicates the storage format for the column's values in the file.

The file specified by *filename* does not accept a null as valid data. The **with null** clause is provided so that you can specify a substitute value for nulls when copying a table that contains nulls. ULTRIX/SQL substitutes the specified *value* in the file whenever it encounters a null in the table. In reverse, if you are copying from a file to a table, ULTRIX/SQL substitutes a null in the table whenever it encounters the specified *value* in the file. (Be careful to choose a value for null that does not occur as part of the data in your table or file.)

The value specified as the substitute for null must be compatible with the format of the field in the file. Character formats require quoted values and numeric formats require unquoted numeric values. Do not use a null character, quoted or unquoted, for a numeric format. For a numeric field, the file does not accept an actual null character, nor will it accept the "null" character string.

If you specify **with null** but do not specify a value, you get an ULTRIX/SQL binary data value. It will have non-printable characters as part of the data representation because every data value has a trailing byte specifying whether the value is **null**. Therefore, you must specify the value in a **with null** clause when using the **c0**, **text(0)**, **char(0)**, and **varchar(0)** format specifications.

If you do specify (*value*) in the **with null** clause, null values are represented by the value specified and there is no byte to represent the null.

To write a file, use the **into** *filename* form of the **copy** statement. To copy data from a file to an ULTRIX/SQL table, use the **from** *filename* form of the statement. *Filename* must be enclosed in single quotation marks. Unless the full pathname is specified, *filename* will be assumed to be in the current directory of the process running the ULTRIX/SQL utility or embedded SQL program.

The **with on_error** clause lets you specify that **copy** should not be terminated due to an error processing a row. If **continue** is set, ULTRIX/SQL does not terminate the **copy** if it encounters errors converting between row and file format. However, the **copy** will be terminated on errors that occur when reading or writing the copy file and on errors that signify a problem with **copy** processing in general rather than a problem confined to a single row. If **terminate** is set, **copy** terminates at the first conversion error. **Terminate** is the default.

If an error is encountered while **on_error** is set to **continue**, a warning message corresponding to the type of error is printed and that row is skipped. When the **copy** is finished, the following message is displayed:

```
COPY: Warning: Copy completed with %d warnings. %d rows
     successfully copied.
```

The **error_count** = *n* clause instructs **copy** to terminate after *n* errors instead of just one. This clause is meaningful only if **on_error** = **terminate** is set. It is an error to specify an **error_count** if **on_error** = **continue** is specified. The default **error_count** is 1.

The **with rollback** clause lets you specify whether rows appended to the database during a **copy** should be backed out if the **copy** is terminated due to an error. This option is meaningful only with **copy from**, because rows are never backed out of the copy file if **copy into** is terminated.

If **rollback** = **enabled** is specified, all rows added to the database during a **copy** statement are backed out if the **copy** is terminated abnormally. This is the default setting.

The **rollback=disabled** option does *not* mean that a transaction cannot be rolled back. Data manager internal errors that may indicate data corruption will still cause back out, and rows are still not committed until the transaction is complete. This option means only that rows will not *automatically* be backed out if an error occurs.

There are two error messages that indicate that **copy** has been interrupted abnormally due to an error or interrupt. If you are running **copy from** and either **rollback** = **enabled** is set or the termination is due to a data manager error, you will get the following error message:

```
COPY: Copy has been aborted
```

Any other abnormal termination will produce the following error message:

```
COPY: Copy terminated abnormally, %d rows successfully copied.
```

The **with log** = *'filename'* clause lets you send rows that **copy** cannot process to the file specified. In a **copy from**, rows written to the log file will be exactly the same as they were in the copy file. In a **copy into**, they will be in the format of the rows in the database.

The **with log** option is especially useful in a **copy from** statement when the **on_error = continue** option is set. In this case, the **copy** will continue to completion even though there may be rows in the copy file which cannot be processed. Warnings are given for each row that cannot be appended to the database, and those rows are written to the log file. You can then edit the log file and fix up the rows in order to load them into the database.

If an error occurs opening the log file, the **copy** will halt. The log file will be opened prior to the start of data transfer, so the copy will halt immediately.

If an error occurs writing to the log file, a warning is given and the **copy** continues.

If the specified log file already exists, it is overwritten with the new values or truncated if the new **copy** statement produces no bad rows.

On a **copy from** a file to a table, the table can have an index, but performance will be much slower than for the same table without an index. Before you can copy into a table, "all to all" permission must be defined on that table. Updates must be allowed on the table; it cannot be an index or system table. You cannot use the **copy** statement to add data through a view. If you **copy** to add rows to a table that has integrity constraints, the integrity constraints are ignored.

To execute a **copy into** a file, either you must be the owner of the table, or the table must have retrieve permission for all users (or all permissions for all users).

The syntax formats for **copy from** are:

Format of Fields in Data File	Storage Format in Table
integer1, smallint, integer	Values are stored as integers of 1-byte, 2-byte or 4-byte length in the file.
float4, float	Values are stored as floating point numbers (either single or double precision) in the file.
c1,...,2000 **char(1),...,char(2000)**	Values are stored as fixed-length strings of type c or **char**.
text(1),...,text(2000) **varchar(1),...,varchar(2000)**	Values are stored as fixed-length strings of type **text** or **varchar**.
c0, char(0)	The value is a variable-length character string (any data type).
text(0), varchar(0)	The value is a variable-length **text** string (any data type).
d0,d1,...,d255	The value is a dummy column of variable (d0) or fixed (d1,...,d255) length (contains no data).
date	Values are stored as internal ULTRIX/SQL dates.
money	Values are stored as internal ULTRIX/SQL money values.

Corresponding columns in the table and their entries in the file need not be of the same type or length. For example, most applications read and write numeric data from files stored in character format and therefore primarily use the **c** or **text** type format. The **copy** statement converts as necessary. When converting anything except character to character, copy mode checks for overflow. When converting from character to character, the **copy** statement pads character strings with blanks or truncates strings on the right, as necessary.

The column names should be ordered according to how they are to appear in the file. Columns are matched according to name. Thus the order of the columns in the table and the file need not be the same.

The **copy** statement provides for variable-length strings and dummy columns. The action taken depends on whether it is a **copy into** or a **copy from** statement. Delimiters for variable-length strings and dummy columns can be selected from the following list:

Delimiter	Description
nl	newline character
tab	tab character
sp	space
nul or **null**	null character
comma	comma
colon	colon
dash	dash
lparen	left parenthesis
rparen	right parenthesis
x	any single character *x* (excluding digits—that is, 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9)

In the file, the special meaning of any delimiter can be suspended by preceding the delimiter with a backslash (\) unless the field format is **text(0)***delim*, where *delim* stands for a delimiter.

2.3.4 Copying from a File into a Table

When copying data from a file into a table, ULTRIX/SQL assigns either a null or a default value to any columns in the ULTRIX/SQL table that are not assigned values from the file. If the column was created as a nullable column, ULTRIX/SQL assigns a null. Otherwise, ULTRIX/SQL assigns a default value of zero for a numeric column or blanks for a character column. If the column was created **not null not default**, and no value is assigned from the file, ULTRIX/SQL returns an error. When copying data from a file into a table, the following special meanings apply:

<table>
<tr><th>Field Format</th><th colspan="2">Description</th></tr>
<tr><td rowspan="8">c0delim, text(0)delim or char(0)delim</td><td colspan="2">Values in the file are variable-length character strings terminated by the delim delimiter. If delim is not specified, the first comma, tab or newline (<nl>) encountered terminates the string. The delimiter is not copied. For example:</td></tr>
<tr><td>pnum=c0</td><td>String ending in comma, tab or <nl>.</td></tr>
<tr><td>pnum=text(0)nl</td><td>String ending in <nl>.</td></tr>
<tr><td>pnum=c0nl</td><td>String ending in <nl>.</td></tr>
<tr><td>pnum=c0sp</td><td>String ending in a space.</td></tr>
<tr><td>pnum='c0Z'</td><td>String ending in the "Z" character.</td></tr>
<tr><td>pnum=text(0)'Z'</td><td>String ending in the "Z" character.</td></tr>
<tr><td>pnum='c0%'</td><td>String ending in the "%" character. A string in the file can contain the delimiter by preceding it with a backslash character (\), but only if the format is c0delim. For example, when using <code>name = c0,</code> the string "Blow\, Joe" is accepted into the column as "Blow, Joe."</td></tr>
<tr><td>d0delim</td><td colspan="2">Values in the file are variable-length character strings delimited by delim. Each string is read and discarded. The delimiter rules are identical for c0 and d0. The column name is ignored.</td></tr>
<tr><td>d1,d2,...,d255</td><td colspan="2">Values in the file are fixed-length byte strings. The specified number of bytes is read and discarded. The column name is ignored.</td></tr>
<tr><td>text(1),...,text(2000)</td><td colspan="2">Values in the file are fixed-length text strings. The fields must be padded with null characters to the given length.</td></tr>
<tr><td>varchar(0)</td><td colspan="2">Values in the file are variable-length varchar strings preceded by a two-byte length specifier.</td></tr>
</table>

When copying from a fixed format file, be sure to take into account the newline (<nl>) characters at the end of each line because there is no requirement that the *rows* you read from the file correspond to the *records* in the file. For example, suppose you have a table called "employee" containing the columns "name," "age" and "department," and a text file containing employee data in a fixed format, as follows, where a caret (^) is a blank space:

```
Jones,J.^^^^^32^^^Anytown,USA^^^toy
Smith,P.^^^^^41^^^New York,NY^^^admin
```

A valid **copy** statement for the preceding file would be something like the following:

```
copy table employee (name=c12, age=c3, xxx=d17,
    department=c0nl) from ... ;
```

Note that the dummy column name "xxx," which is not in the table, is an arbitrary name for the skipped field from the file. The name itself has no particular meaning. The last field in the fixed field file, in this case "department," is most conveniently specified as **c0nl.** This instructs ULTRIX/SQL to read the remainder of the line into the "department" column of the table.

Note that the format indicators in the **copy from** statement should describe how values are represented *in the file*. This is not necessarily the same format as the corresponding table column. For example, the file record might contain a numeric field holding a string of ASCII characters, such as 1927.63, which would be converted on input and stored in the ULTRIX/SQL table in a column of type **float.** In this case, the **copy** statement should describe the field as a **c** format item, not **float.**

Finally, note that copying from a file into an empty, non-journaled table without indexes runs significantly faster than copying into a table that contains one or more rows, is journaled, or has indexes. The copy is fastest when the table is in the **heap** storage structure.

2.3.5 Copying Data from a Table into a File

When the direction is **into, copy** transfers data *into* the file from the table. If the file already exists, it is overwritten, if allowed by the ULTRIX permissions. When copying in this direction, the following special meanings apply:

Field Format	Description
c0, char(0)	The column is converted to a fixed-length character string and written into the file. For character columns, the length is the same as the column length. For numeric columns, the standard ULTRIX/SQL conversions take place as specified by the **-i, -f** and **-c** flags. (See the **sql** command in Chapter 4.)
varchar(0)	The column is converted to **varchar** and written as a variable-length string preceded by a two-byte length specifier.
c0*delim*, **char(0)***delim*	The column is converted according to the rules for **c0** above. The one-character delimiter is inserted immediately after the column. (Note that for numeric columns, **c0sp** and **char(0)sp** are not meaningful and will result in input errors because the data is converted by right justification and blankfill.)
text(0)*delim*	The column is converted to a **text** string and written into the file. The one-character delimiter is inserted immediately after the column. For **c** and **char** columns, the length is the same as the column length. For **text** and **varchar** columns, the length varies according to the number of characters in each text value. For numeric columns, the standard ULTRIX/SQL conversions take place as specified by the **-i** and **-f** flags. (See the **sql** command in Chapter 4.) Note that for numeric columns, **text(0)sp** is not meaningful and will result in input errors because the data is converted by right justification and blankfill.

Field Format	Description
text(1),...,text(2000)	The column is converted to a **text** string and written into the file, according to the rules for **text(0)** *delim* above. No delimiter is used. If necessary, the column is padded with null characters to the given length.
d1,d2,...,d255	The column name is taken to be the name of the delimiter. It is written into the file once for d1, twice for d2, etc.
d0	This format is ignored on a **copy into** statement.
d0*delim*	The *delim* is written into the file. The column name is ignored.
varchar(0)	The variable-length-only specifier and data are written to the file.

Note that arbitrary delimiters can be specified independently of columns on a **copy into** statement. If you want to specify a newline character at the end of a line, include nl=d1 at the end of the list of columns, where nl ("n" followed by lowercase L) stands for newline, and d1 ("d" followed by the number one) instructs ULTRIX/SQL to add one (d1) newline (nl) character. Do not confuse "l" (lowercase L) and "1" (number one).

If no columns appear in the **copy** statement (that is, **copy table** *tablename* () **into|from** *filename*), then the **copy** statement automatically performs a *bulk copy* of all columns, using the order and formats of the columns in the table. This is provided as a convenient shorthand notation for copying and restoring entire tables.

2.3.6 Performance Issues in Copying from a File into a Table

Copying from a file into a non-journaled **heap** table without secondary indexes runs significantly faster than copying into a **btree, isam** or **hash** table, or a table that is journaled or has secondary indexes.

For example, consider the following two queries. The first will run more slowly because the table's **btree** index must be dynamically maintained as data is copied from the external file into the table:

```
CREATE TABLE employee (name text(12), age integer2,
    department text(8) ) ;
MODIFY employee TO btree ON name ;
COPY TABLE employee (name=c12, age=c3, xxx=d17,
    department=c0nl) FROM ... ;
```

The following query, on the other hand, will run more quickly because the "employee" table is a **heap** while data is copied and ULTRIX, therefore, does not need to maintain an index structure for the table during the copy operation. After the **copy** statement is complete, the table is modified to **btree:**

```
CREATE TABLE employee (name text(12), age integer2,
    department text(8) ) ;
COPY TABLE employee (name=c12, age=c3, xxx=d17,
    department=c0nl) FROM ... ;
MODIFY employee TO btree ON name ;
```

Depending on the initial size of the database table and the amount of data to be copied from the external file, it may be faster to modify the database table to **heap** before copying data into it. For example, if "departments" is an existing table that is **btree** on the "department" column, it may be faster to copy with the first of the following two scripts:

```
MODIFY departments TO heap ;
COPY TABLE departments (department=c8, ... ) FROM ... ;
MODIFY departments TO btree ON department ;   /* restore original
    structure */
```

This second copy script, below, may run more slowly because it requires ULTRIX/SQL to maintain the index structure on the "departments" table during the copy operation:

```
COPY TABLE departments (department=c8, ... ) FROM ... ;
```

As a general rule, if the external file contains more rows than the database table, then you may get better performance by modifying to **heap** before doing the **copy** and then modifying to the correct structure when the **copy** is complete. Note that if the database table is empty, it is nearly always better to modify to **heap** before doing the copy.

For more information, see the *Ultrix/SQL Database Administrator's Guide.*

2.3.7 Effect of Table Structure on copy from Performance

Isam Avoid copying large amounts of data into a database table that has an **isam** structure. You should modify the table to **heap** first, as described above. It is particularly inefficient, from a performance standpoint, to copy into an empty **isam** table.

The index structure of an **isam** table is fixed (as opposed to **btree**, whose index structure grows dynamically as data is added to the table) and therefore will not grow as you add data. The net result will be overflow chains that can significantly degrade database management system performance.

Hash You can get good performance copying data into a **hash** structure if the **hash** structure has been pre-allocated with enough space for the new data and if many rows do not **hash** to the same page and produce overflow. The **minpages** parameter is used to pre-allocate space in **hash** tables.

Copying into a **heap** structure gives the best performance. Use of the **btree** structure will not be as fast as **heap** because the index structure must be maintained, but should be faster than **isam** because of the lack of long overflow chains. **Hash** will be fast if the table has enough empty space to hold the new data.

2.3.8 Examples

The first two examples in this section illustrate different ways of representing numeric data in a file. In the first example, several fields are represented in 2-byte integer format, and "sal" is represented as a 4-byte floating point item. These items would not be readable as characters with the text editor. The **copy** statement loads them into ULTRIX/SQL table columns, which may or may not have the same format as the file data.

The second example copies some of the same data out of the "employee" table into a file. This time, all items are written as character data. This means, for instance, that "sal" would be converted from its format in the ULTRIX/SQL table (say, **float4** or **float**) to ASCII characters in the result file.

Copy data into the "employee" table.

```
copy table employee (eno=integer2, ename=c10, age=integer2,
    job=integer2, sal=float4, dept=integer2, xxx=d1)
    from '/usr/mydir/files/myfile.in';
```

Copy employee names, numbers and salaries into a file, inserting commas and newline characters so that the file can be printed or edited (either of the following statements).

```
copy table employee (ename=c0, comma=d1, eno=c0, comma=d1,
    sal=c0, nl=d1)
    into '/usr/mydir/files/mfile.out';

copy table employee (ename=c0comma, eno=c0comma, sal= c0nl)
    into '/usr/mydir/files/mfile.out';
```

Bulk copy the "employee" table into a file.

```
copy table employee ()
    into '/usr/mydir/files/ourfile.dat';
```

Bulk copy the "employee" table from a file.

```
copy table employee ()
    from '/usr/mydir/another.fil';
```

2.4 create index

2.4.1 Purpose

Create an index on an existing base table.

2.4.2 Syntax

create [**unique**] **index** *indexname* **on** *tablename*
    (*columnname* {,*columnname*})
        [**with** *with_options_list*]

The *with_options_list* consists a comma-separated list of any of the following items:

fillfactor = *n*
key = (*columnlist*)
leaffill = *n*
location = (*locationname* ...)
maxpages = *n*
minpages = *n*
nonleaffill = *n*
structure = **cbtree** | **btree** | **cisam** | **isam** | **chash** | **hash**

2.4.3 Description

The **create index** statement creates an index on an existing base table. The index key is constructed of columns from the specified table in the order given. A maximum of 32 *columnnames* may be specified per index, but you can build any number of indexes for a table. Only the owner of a table is allowed to create indexes on that table.

Rows are returned in ascending order, by default.

The **key**=(*columnlist*) option enables you to create an index with more attributes (columns) than you want the key to contain. This can improve performance, since ULTRIX/SQL does not have to return to the base table if the information needed to satisfy a query is in the index. If you use this option, the columns in *columnlist* must be an ordered subset of the columns specified in the index definition. In addition, they must be the leading columns in the index definition. For example, an index defined on columns *a*, *b*, *c* and *d* may be keyed on *a*, or *ab*, or *abc* or *abcd.* (The default is *abcd* if the **key** clause is omitted.)

Fillfactor specifies the percentage (from 1 to 100) of each primary data page that should be filled with rows, under ideal conditions. **Fillfactor** may be used with **isam, cisam, hash, chash, btree** and **cbtree**. When creating a table with storage structure **btree** or **cbtree, nonleaffill** determines the percentage of each index page to fill. Care should be taken when specifying large fillfactors because a non-uniform distribution of key values could later result in overflow pages and thus degrade access performance for the table.

Minpages specifies the minimum number of primary pages a **hash** or **chash** table must have. **Maxpages** specifies the maximum number of primary pages a **hash** or **chash** table may have. **Minpages** and **maxpages** must be at least one. If both **minpages** and **maxpages** are specified in a **create index** statement, **minpages** cannot exceed **maxpages.**

The default values for **fillfactor, minpages** and **maxpages** are as follows:

Structure	Fillfactor	Minpages	Maxpages
hash	50	16	no limit
chash	75	1	no limit
isam	80	---	---
cisam	100	---	---
btree	80	---	---
cbtree	100	---	---

The **leaffill** parameter of the **create index** statement applies only to tables stored in **btree** and **cbtree** structures. The **leaffill** parameter specifies percentages to fill each index page for a **btree** or **cbtree** table.

The **leaffill** specifies a percentage *n*, where *n* ranges from 1 to 100, and its percentage specifies how much each index page should be filled *at the time the table is modified to* **btree** *or* **cbtree**. This parameter contrasts with the **fillfactor** parameter, which specifies the percentage occupancy of data pages (not index pages) when a table is converted to **btree** or **cbtree.**

The **leaffill** parameter allows you to control locking contention in **btree** and **cbtree** index pages. By retaining a percentage of open space on these index pages, more concurrent users can access the **btree** without contention while their queries descend the index tree. Note, however, that you must strike a balance between preserving space in index pages and creating a greater number of index pages; more levels of index pages require more I/O to locate a data row.

The default value for **leaffill** is 70 (percent). This default applies to both **btree** and **cbtree** indexes.

The parameter *locationname* refers to the location(s) on which the new index will be created. The *locationname* must be defined on the system and the database must have been extended to the corresponding location. If no *locationname* is specified, the default location for the database is assumed. If multiple *locationnames* are specified, the index is physically partitioned across the locations. (See Chapter 1 for more information about ULTRIX/SQL *locationnames*, locations and multi-location tables.)

In order to maintain the integrity of the index, users are *not* permitted to update indexes directly. However, whenever a table is changed, its indexes are automatically updated by the system. Indexes may be modified to increase even further the access efficiency of the table. When an index is first created, it is automatically modified to an **isam** storage structure on all its columns. If this structure is undesirable, you may override the default structure with the **-n** flag (see the **sql** command in Chapter 4) by entering a **modify** statement directly, or by specifying the modify parameters in the **with** clause of the **create index** statement.

Once created, an index improves query processing "silently." That is, if you retrieve data from a table based on an indexed column, you need not indicate to ULTRIX/SQL that it should consult the index. ULTRIX/SQL automatically uses indexes to accelerate query processing once the indexes are created.

If a **modify** or **drop** statement is used on a table, all indexes on that table are destroyed. Note also that the **modify** and **drop** statements can be executed directly on an index.

You are not allowed to create indexes on system tables.

2.4.4 Examples

Create an index called "x" for the columns "ename" and "age" on table "employee."

```
create index x on employee (ename, age);
```

Create an index called "ename" and have it located on the location referred to by the *locationname* "remote."

```
create index ename on employee (ename, age)
    with location = (remote);
```

2.4.5 Usage Notes

No more than 32 columns may appear in the index key.

2.5 create integrity

2.5.1 Purpose

Define integrity constraints on a base table.

2.5.2 Syntax

create integrity on *tablename* [*corr_name*] **is** *search_condition*

2.5.3 Description

The **create integrity** statement creates an integrity constraint for the specified base table. After the constraint is defined, all updates to the table must satisfy the specified search condition. The search condition must be true for every existing row in the table when the **create integrity** statement is issued; if it is not true, a diagnostic is issued, and the integrity constraint is rejected. Note that the column may contain **null** values; ULTRIX/SQL ignores **null** values when evaluating a column at the time a **create integrity** is issued against the column.

In the current implementation, integrity constraints that are violated are not specifically flagged. Updates that violate any integrity constraints are simply not performed.

The search condition must not involve any tables (or their correlation names) other than the one specified in the **on** clause. The search condition must also not contain a subselect.

The **create integrity** statement may be issued only by the table owner.

2.5.4 Examples

Make sure that all employee salaries are equal to or greater than 6000.

```
create integrity on employee is salary >= 6000;
```

2.6 create procedure

2.6.1 Purpose

Create a named database procedure definition.

2.6.2 Syntax

```
[create] procedure proc_name
        [(param_name [=] param_type {,param_name [=] param_type})]
        = | as
    [declare_section]
begin
    statement {; statement}[;]
end
```

2.6.3 Description

The **create procedure** statement creates a named database procedure definition that is managed as a named object by ULTRIX/SQL as part of a database. Database procedures are executed by an **execute procedure** statement that you embed in a host language program (see the *ULTRIX/SQL Reference Guide to Embedded SQL* for details).

The parameter *proc_name* is the name of the procedure. The name must be a legal ULTRIX/SQL name (see Chapter 1).

The parameter *param_name* is the formal name of the procedure parameter.

The parameter *param_type* is the procedure parameter's type. It can be any of the ULTRIX/SQL types (see Chapter 1). All types may have the **null** or **default** clauses.

The *declare_section* declares a list of local variables that you can reference in the procedure body. The syntax for this section is:

```
declare
    var_name { , var_name} [=] var_type;
    {var_name { , var_name} [=] var_type;}
```

Refer to the summary of the **declare** statement in this manual for full information about this syntax.

The parameter *statement* may include local variable assignments and any of the following:

commit	**insert**	**rollback**	**while**
delete	**message**	**select**	
if	**return**	**update**	

Some of the statements in the above list (for instance, **if, while, message** and **return**) can be coded only in the procedure definition, but are discussed in separate statement sections within this chapter.

A procedure cannot contain any data definition statements, such as **create table**, nor may it **create, drop**, or **execute** another procedure. Additionally, unlike the embedded ULTRIX/SQL versions of some of these statements, you cannot use the **repeat** clause in a statement in the procedure body. (Using the procedure itself provides the same performance benefits as the **repeat** clause.)

Select statements inside a procedure must assign their results to local variables. Also, they can return only a single row of data. If more rows are returned, no error is issued, but only the first row retrieved is in the result variables.

Both procedure parameters and local variables can be used in place of any constant value in statements in the procedure body. Procedure parameters are treated as local variables inside the procedure body, although they have an initial value assigned when the procedure is invoked. Preceding colons (:) are only necessary if the referenced name could be interpreted to refer to more than one object.

Local variable assignments use the equal sign (=) or colon and equal sign (:=) as the assignment operator.

All statements, except a statement preceding an **end, endif**, or **endwhile**, must be terminated by a semicolon.

You can replace the keywords **begin** and **end** with braces ({}) but the terminating semicolon must follow the closing brace if another statement is entered in interactive mode after the **create procedure** statement and before committing the transactions.

2.6.4 Examples

In this example, the "mark_emp" database procedure accepts as input an employee ID number and a label string. The employee matching that ID is labeled and an indication is returned.

```
CREATE PROCEDURE mark_emp
    (id INTEGER NOT NULL, label VARCHAR(100)) AS
BEGIN
    UPDATE employee
        SET comment = :label
        WHERE id = :id;
    IF iirowcount =1 THEN
        MESSAGE 'Employee was marked';
        COMMIT;
        RETURN 1;
    ELSE
        MESSAGE 'Employee was not marked - record error';
        ROLLBACK;
        RETURN 0;
    ENDIF;
END;
```

In this example, the "add_n_rows" database procedure accepts as input a label, a base number, and a number of rows. The procedure inserts the specified number of rows into the table "blocks," starting from the base number. If an error occurs, then the procedure terminates and the current row number is returned.

```
CREATE PROCEDURE add_n_rows
     (base INTEGER, n_INTEGER, label VARCHAR(100)) AS
DECLARE
     limit INTEGER;
     err INTEGER;
BEGIN
     limit = base + n;
     err = 0;
     WHILE (base < limit) AND (err = 0) DO
          insert into blocks VALUES (:label, :base);
          IF iierornumber > 0 THEN
               err = 1;
          ELSE
               base = base + 1;
          ENDIF;
     ENDWHILE;
     RETURN :base;
END;
```

2.7 create table

2.7.1 Purpose

Create a new base table.

2.7.2 Syntax

create table *tablename*
(*columnname format* {, *columnname format*})
[*with-clause*];

create table *tablename*
[(*columnname* {, *columnname*})]
as *subselect*
[**with** *with_options_list*];

A *with_options_list* consists of a comma-separated list of any number of the following items:

location = (*locationname* {, *locationname*})
[**no**]**journaling**
[**no**]**duplicates**
structure = *storage_structure* [, **key** = (*columnname*)]

For the syntax of *subselect*, see the **select** section later in this chapter.

2.7.3 Description

The **create table** statement creates a new base table owned by the user who issues the statement. The parameter *tablename* specifies the name of the table. The name and data type of each column in the new table are specified by the *columnname* and *format* arguments. If these arguments are not included in the **create table** statement, then you must include the **as** clause. The new table will then take its column names and formats from the results of the **select** clause of the *subselect* in the **as** clause.

When the **create table** statement includes an **as** clause, specifying column names is optional unless two or more columns of the table would otherwise have the same name. If that is the case, you must specify the column names.

You cannot specify a column format if your **create table** statement includes an **as** clause. ULTRIX/SQL derives the column format from the source table in the following manner:

- The result column data type is the same as the source column data type.

- If a column in the source table was created **with null**, the format of the result column in the new table will also be **with null.**

- If a column in the source table was created **not null with default,** or **not null not default,** or **not null**, the format of the result column in the new table will be **not null not default.**

The parameter *columnname* can be any valid ULTRIX/SQL name.

The parameter *format* has the following syntax:

datatype [**not null** [**with default** | **not default**] | **with null**]

The parameter *datatype* can be any valid ULTRIX/SQL data type and length. See Chapter 1 for a discussion of valid data types and formats.

The **with null** | **not null** clause determines what happens during an **insert** to a field for which no value is specified. There are three possible settings for this clause:

with null
not null with default
not null not default

The clause **with null** means that a field into which no data has been inserted is marked as having no value. The clause **not null with default** means the default value (0 for numeric formats and spaces for character formats) is stored in the column on insert when no value is supplied by the **insert** statement. The clause **not null not default** means an error condition is created when the insert is attempted without a value.

If no **with null** | **not null** clause is specified, **with null** is assumed. If **not null** alone is specified, **not null not default** is assumed.

A table can have a maximum of 127 columns and can be a maximum of 2000 bytes wide. Note that a **varchar** or **text** column requires two more bytes than the value specified in the format to indicate the precise length of the stored value. For example, **varchar**(20) stores values up to 20 characters long, but requires 22 bytes in storage. A nullable column requires one more byte than the value specified in the format. A table cannot be defined to have a name beginning with "ii."

Tables are created with no expiration date. If you want to impose an expiration date on a table, use the **save** statement.

If an **as** clause is specified, the table is populated with the set of rows resulting from execution of the specified subselect; otherwise the table is created empty. If **as** is specified, the new table is created with the storage structure defined by the most recent **set result_structure** statement within the session (see the **modify** statement); the default is compressed heap. If **as** is not specified, the new table is created as heap.

The parameter *locationname* refers to the location(s) (see Chapter 1) on which the new table will be created. The *locationname(s)* must be defined on the system and the database must have been extended to the corresponding location(s). If no *locationname* is specified, the default location for the database is assumed. If multiple *locationnames* are specified, the table is physically partitioned across the locations, as described in the section of Chapter 1 titled "Multiple-Location Tables." (Please see the chapter on **accessdb** in the *ULTRIX/SQL Operations Guide*.)

If **with journaling** is set, journaling will occur for the table only if journaling is enabled for the database as a whole using the **ckpdb** statement (see Chapter 5). Enabling journaling causes ULTRIX/SQL to keep a record of all changes to the table (**inserts**, **updates** and **deletes**) in the journal for the containing database, and thus allows the ULTRIX/SQL recovery system to reconstruct the table after a disk crash. Journaling also allows an audit trail to be built for the table, which is useful for monitoring updates or for maintaining change histories.

It is not necessary to enable journaling to recover from operating system or ULTRIX/SQL failures. Recovery from such a failure is a standard function of transaction processing.

The **with duplicates | no duplicates** option does not affect a table created as a heap. This type of storage structure allows duplicate rows regardless of the setting of this option. The **with duplicates | no duplicates** setting affects only those tables created as or later modified to be structures other than **heap**. Additionally, this setting can be overridden by specifying a unique key for a table using the **modify** statement. (See the **modify** statement for more information about table structures with unique keys.)

2.7.4 Examples

Create the "employee" table with columns "eno," "ename," "age," "job," "salary" and "dept," with journaling enabled.

```
create table employee
    (eno     smallint,
    ename    varchar(20) not null with default,
    age      integer1,
    job      smallint,
    salary   float4,
    dept     smallint)
    with     journaling;
```

Create a table with some other data types.

```
create table debts
    (acct    varchar(20) not null not default,
    owes     money,
    due      date not null with default);
```

Create a table listing employee numbers for employees who make more than the average salary.

```
create table highincome
    as select    eno
    from         employee
    where        salary  all
                 (select avg (salary)
                 from    employee);
```

Create a table which will span two locations.

```
create table emp as
    select eno from employee
        where location = (location1, location2);
```

2.8 create view

2.8.1 Purpose

Define a virtual table.

2.8.2 Syntax

create view *viewname* [(*columnname* {, *columnname*})] **as** *subselect*
    [**with check option**]

The syntax of *subselect* is described in the **select** statement summary in this chapter.

2.8.3 Description

The syntax of the **create view** statement is very similar to that of the **as** form of **create table**. However, data is not retrieved when a view is created. Instead, the definition is stored. When *viewname* is later used in an ULTRIX/SQL statement, the statement operates on the associated base tables, which are tables that store data.

All selects on views are fully supported. Simply use a *viewname* in place of a *tablename* in any selects. However, updates, inserts, and deletes on views are subject to the following rules:

- Updates, inserts and deletes are not allowed if the view was created from more than one table or from a non-updatable view.
- The ability to **update** a view or **insert** a new row depends on whether the **with check option** is specified, as explained below.

If you specify the **with check option**, you cannot update that view if the result of the statement removes a row from the view. Nor can you **update** columns that are part of the view's qualification or whose source is not a simple column (that is, columns that result from an expression or set function).

If you do not specify the **with check option**, you can **update** any row in the view, even if the update results in a row that is no longer a part of the view.

For example, consider the following two statements:

```
create view      v
    as select    *
    from         t
    where        c > 10

update       v
set          c = 5
```

Once *c* is set to the value 5, then when *t* is updated, the updated rows are no longer in the view. If the view had been created **with check option**, the update would not be allowed.

Inserts are only allowed if you do not specify the **with check option**. In addition, all columns in the underlying table that were declared as **not null not default** must be present in the view. If they are not, then the **insert** operation is not allowed.

By default, **with check option** is not set.

Although a person who defines a view need not own all tables upon which a view is based, use of the view is restricted to those who have all necessary permissions to the base tables. Permissions on the base tables or on views owned by the Database Administrator (DBA) may be granted by the DBA using the **grant** statement.

When a table used in the definition of a view is dropped, the view is also dropped.

2.8.4 Example

Define a view of employee data including names, salaries and managers' names.

```
create view      empdpt (ename, sal, dname)
    as select    employee.name, employee.salary,
                     dept.name
    from         employee, dept
    where        employee.mgr = dept.mgr;
```

2.9 declare

2.9.1 Purpose

Declare a list of local variables for use in a database procedure.

2.9.2 Syntax

declare
var_name {, *var_name*} [=] *var_type*[**not null** [**with default** | **not default**] | **with null**];
{*var_name* {, *var_name*} [=] *var_type*[**not null** [**with default** | **not default**] | **with null**];}

2.9.3 Description

This statement is used only in a database procedure definition, to declare a list of local variables for use in the procedure. The statement is optional and, if used, is placed before the **begin** clause.

The parameter *var_name* is the name of the local variable. Variable names must be unique within the procedure body.

The parameter *var_type* is the type of the variable. A local variable type may be any of the ULTRIX/SQL data types. Nullable variables are initialized to null; non-nullable variables are initialized to the default value. For example, a non-nullable floating point variable is initialized to 0.0 by default. Any non-nullable variables declared without an explicit default value are initialized to the ULTRIX/SQL default value.

2.9.4 Example

This procedure fragment demonstrates some declarations and uses of local variables. Note that some of these statements will cause an error.

```
CREATE PROCEDURE variables (vmny MONEY NOT NULL) AS
DECLARE
    vi4     INTEGER NOT NULL;
    vf8     FLOAT;
    vc11    CHAR(11) NOT NULL;
    vdt     DATE;
BEGIN
    vi4 = 1234;
    vf8 = NULL;
    vc11 = '26-jun-1957';
    SELECT DATE(:vc11) INTO :vdt;
    RETURN :vi4;
END;
```

2.10 delete

2.10.1 Purpose

Delete rows from a table.

2.10.2 Syntax

delete from *tablename* [*corr_name*]
[**where** *search_condition*]

2.10.3 Description

The **delete** statement removes rows that satisfy *search_condition* from the specified table. If the **where** clause is omitted, the statement deletes all rows in the table. The result is a valid but empty table.

Note that **delete** does not automatically recover the space in a table left by the deleted rows. However, if you add new rows later, the empty space may be reused. If you delete many rows from a table, you may want to run the **modify to merge** statement to recover the lost space. You can specify any storage structure and still recover the empty space. In particular, if you want to delete all rows from a table, you can use the special **modify** *tablename* **to truncated** to delete all rows and recover the space at one time. (See the **modify** statement in this chapter for more information.)

To delete rows from a table, you must either be its owner or have **select** and **delete** permission on the table.

2.10.4 Example

Remove all employees who make over $35,000.

```
delete from employee where salary > 35000;
```

2.11 drop

2.11.1 Purpose

Destroy (remove) one or more tables, indexes or views.

2.11.2 Syntax

drop *tablename* | *indexname* | *viewname* {, *tablename* | *indexname* | *viewname*}

Alternate forms:

drop table *tablename* {,*tablename*}
drop index *indexname* {,*indexname*}
drop view *viewname* {,*viewname*}

2.11.3 Description

The **drop** statement removes the specified table(s), indexes and views from the database. Only the owner of a view or table is allowed to drop it. Likewise, only the owner of an indexed table is allowed to drop an index.

If a table is dropped, any indexes and views defined on that table are automatically dropped too.

If **drop table, drop view**, or **drop index** is used, the object name is checked to be sure it is the correct type. For instance, **drop table** *viewname* is not permitted. Similarly, **drop table** *tablename*, *viewname* will drop the table and not the view.

If a **drop** statement is used without any of the keywords **table, index** or **view**, the object names can be any mixture of the three types, since object names must be unique within the database.

2.11.4 Example

Drop the "employee" and "dept" tables.

```
drop employee, dept;
```

2.12 drop integrity

2.12.1 Purpose

Destroy (remove) one or more integrity constraints.

2.12.2 Syntax

drop integrity on *tablename integer* {, *integer*}

The key word **all** can appear in place of the list of integers.

2.12.3 Description

The **drop integrity** statement removes the specified integrity constraints from the database. The constraints are specified by integers whose values can be obtained using the **help integrity** statement. Alternatively, the key word **all** can be specified, meaning all integrity constraints currently defined for the table in question.

Only the owner of the table to which a given constraint applies is allowed to drop that constraint.

2.12.4 Example

Drop integrity constraints 0, 4, and 5 on "job."

```
drop integrity on job 0, 4, 5;
```

2.13 drop permit

2.13.1 Purpose

Destroy (remove) one or more permissions.

2.13.2 Syntax

For tables and views:

drop permit on *tablename integer* {, *integer*}

For procedures:

drop permit on procedure *proc_name*
integer | **all**

2.13.3 Description

The **drop permit** statement removes specified permissions from the table, view or procedure. The permissions are specified by integers whose values can be obtained using the **help permit** statement. Alternatively, the key word **all** can be specified, meaning all permissions currently defined for the table, view or procedure in question.

Only the owner of the table to which a given permission applies is allowed to drop that permission.

2.13.4 Examples

Drop all permissions on "job."

```
drop permit on job all;
```

Drop the second permission on procedure "AddEmp."

```
drop permit on procedure AddEmp 2;
```

2.14 drop procedure

2.14.1 Purpose

Remove a procedure definition from the database.

2.14.2 Syntax

drop procedure *proc_name*

2.14.3 Description

This statement removes a database procedure definition from the database. When executed, it takes effect immediately. Executions in progress, invoked by other users, are allowed to continue until they are completed. A procedure can only be dropped by its owner.

The parameter *proc_name* is the name of the procedure to be removed.

2.14.4 Example

This statement removes the procedure named "salupdt."

```
drop procedure salupdt
```

2.15 grant

2.15.1 Purpose

Grant privileges on a table, view, or procedure.

2.15.2 Syntax

grant all [**privileges**] **on** [**table**] *tablename* {, *tablename*} **to public**

grant all [**privileges**] **on** [**table**] *tablename* {, *tablename*}
  to *username* {, *username*}

grant *priv* {, *priv*} **on** [**table**] *tablename* {, *tablename*} **to public**

grant *priv* {, *priv*} **on** [**table**] *tablename* {, *tablename*} **to** *username* {, *username*}

grant *priv* **on procedure** *proc_name* { , *proc_name*}
  to public | *username* {, *username*}

2.15.3 Description

The parameter *priv* represents one of the following privileges:

- **select**
- **insert**
- **delete**
- **update** (*columnname* {, *columnname*})
- execute

The **grant** statement grants one or more of these privileges to any set of users on the tables, views, or procedures specified. The privileges **select, insert, update,** and **delete** can only be granted on tables or views. The privilege **execute** can only be granted on procedures.

A **grant** statement must be issued by the Database Administrator (DBA) of the current database, who must own all the tables, views and procedures specified. If a non-DBA issues a **grant** statement, an error is returned. If the DBA issues a **grant** statement that includes tables, views or procedures that the DBA does not own, processing will continue on all the tables, views or procedures that the DBA does own.

If the DBA issues a **grant** statement to allow a user to use a view or procedure, then the user can do so without permission(s) on the underlying tables or views.

The optional words **privileges** and **table** have no effect. They are included for compatibility with other versions of SQL.

2.16 help

2.16.1 Purpose

Get information about SQL, or about tables in the database.

2.16.2 Syntax

help [*]

help *tablename* | *viewname* | *indexname*
{, *tablename* | *viewname* | *indexname*}

help table *tablename* {, *tablename*}

help view *viewname* {, *viewname*}

help index *indexname* {, *indexname*}

help permit|integrity *tablename* {, *tablename*}

help procedure *procedure_name* {, *procedure_name*}

help help

help sql

help *sql_statement*

2.16.3 Description

The **help** statement may be used to display information about ULTRIX/SQL features, definitions of views, protections or permissions, or information about the contents of the database and specific tables in the database. In addition, **help** may be used at the Terminal Monitor to obtain information regarding ULTRIX/SQL, including such features as the syntax of ULTRIX/SQL statements and the available data types. The legal forms are as follows:

help	Lists all user (not system) tables, views, and indexes that exist in the current database.
help *	Gives information about the makeup of all user-defined (not system) tables, views, and indexes in the database.
help *tablename* {, *tablename*}	Provides the name, owner, creation date and time, and the database management system version under which the table was created. It also provides the following information for each column in the table: name, data type, length, whether nullable or not, whether a default value will be provided, and the key sequence.
help *viewname* {, *viewname*}	Provides information for views similar to that provided for tables by **help table**.

help *indexname* {, *indexname*}	Provides information for indexes similar to that provided for tables by **help table**.
help table *tablename* {, *tablename*}	Gives the same information as **help** *tablename* and additional information such as the number of pages in the table, whether journaling is on, any optimizer statistics, permissions, secondary indexes, location, row width, and number of columns.
help view *viewname* {, *viewname*}	Displays the text of the view, the view name, owner, and the state of the check option.
help index *indexname* {, *indexname*}	Displays the name, owner, creation date and time, database management system version under which it was created, and, for each column, the index name and sort direction.
help permit *tablename* {, *tablename*}	Prints the permission text for the specified tables. You may also use **help permit** to display the permission text for indexes. **Help permit** will not, however, provide help for permissions on procedures.
help integrity *tablename* {, *tablename*}	Prints current integrity constraints on the specified tables. You can also use this statement to print current integrity constraints on indexes.
help procedure *procedure_name* {,*procedure_name*}	Displays the procedure name, definition, and the creation date and time. Note that if you are not the procedure's creator, then the definition is not shown. If permissions are defined on the procedure, **help procedure** lists them also.
help help	Prints a list of ULTRIX/SQL features for which help is available.
help sql	Prints general information about ULTRIX/SQL.
help *sql_statement*	Prints information on the specified *sql_statement*.

You can use the asterisk (*) as a pattern-matching character when specifying an object name. For example, if you type "help table emp*" you receive help on all tables in the database whose names begin with "emp." If you put the asterisk in front of "emp" (as in "help table *emp"), you would receive help on all the tables whose names ended with "emp."

When you are using the asterisk as a pattern matching character, do not use the percent sign (%) or, in particular, the underscore (_) at the same time in the object name.

When the asterisk is used by itself with **help** (that is, **help ***) it is the equivalent of the word **all**.

The **permit** and **integrity** forms of the **help** statement print out unique integer identifiers for each constraint. The **drop permit** and **drop integrity** statements use these identifiers to remove individual constraints. (See the sections in this chapter on **drop integrity** and **drop permit**.)

2.16.4 Examples

Retrieve a list of all tables in the database.

```
help;
```

Retrieve help about the "employee" table.

```
help employee;
```

Retrieve help about the "employee" and "dept" tables.

```
help employee, dept;
```

Retrieve the definition of the "highpay" view.

```
help view highpay;
```

List all permits issued on the "job" and "employee" tables.

```
help permit job, employee;
```

List all integrity constraints issued on the "dept" and "employee" tables.

```
help integrity dept, employee;
```

List information on the **select** statement.

```
help select;
```

2.17 if-then-else

2.17.1 Purpose

Choose between alternative paths of execution inside a database procedure.

2.17.2 Syntax

if *boolean_expr* **then** *statement*; {*statement*;}
    {**elseif** *boolean_expr* **then** *statement*; {*statement*;}}
    [**else** *statement*;{*statement*;}]
endif

2.17.3 Description

In ULTRIX/SQL, this statement can only be specified as part of a database procedure. (See the **create procedure** statement for details about issuing statements within procedures.)

A boolean expression (*boolean_expr*) must always evaluate to "true" or "false." As discussed in Chapter 1, a boolean expression can include comparison operators (=, >, and so on) and the logical operators **and**, **or** and **not**. Boolean expressions processing null values frequently evaluate to "unknown." Any boolean expression whose result is "unknown" will behave exactly as if it evaluated to "false."

The simplest variant of the **if** statement performs an action only if the boolean expression evaluates to "true." The syntax for this variant is as follows:

if *boolean_expr* **then**
    statement; {*statement*;}
endif

If the boolean expression evaluates to "true," the list of statements is executed. If the expression evaluates to "false" (or "unknown"), the statement list is not executed, and control passes directly to the statement following the **endif** terminator.

The second variant of the **if** statement includes the **else** construct. Its simplest form is as follows:

if *boolean_expr* **then**
    statement; {*statement*;}
else
    statement; {*statement*;}
endif

In this variant, if the boolean expression is true, the statements immediately following the key word **then** are executed. If the expression is false (or "unknown"), the statements following the key word **else** are executed. In either case, after the appropriate statement list is executed, control passes to the statement immediately following **endif**.

The third **if** variant involves the **elseif** construct. The **elseif** construct allows the running application to test a series of conditions in a prescribed order. The statement list corresponding to the first true condition found is executed; all other statement lists connected to conditions are skipped. The **elseif** construct can be used with or without an **else** construct, which must follow all the **elseif** constructs. If an **else** construct is included, one statement list is guaranteed to be executed, because the statement list connected to the **else** is executed if all the specified conditions evaluate to "false."

The simplest form of this variant is the following:

> **if** *boolean_expr* **then**
>   *statement*; {*statement*;}
> **elseif** *boolean_expr* **then**
>   *statement*; {*statement*;}
> **endif**

If the first boolean expression evaluates to "true," the statements immediately following the first **then** key word are executed. In such a case, the value of the second boolean expression is irrelevant. If the first boolean expression proves false, however, the next boolean expression is tested. If the second expression is true, the statements following the second **then** key word are executed. If both boolean expressions test false, neither statement list is executed.

A more complex example of the **elseif** construct is as follows:

> **if** *boolean_expr* **then**
>   *statement*; {*statement*;}
> **elseif** *boolean_expr* **then**
>   *statement*; {*statement*;}
> **elseif** *boolean_expr* **then**
>   *statement*; {*statement*;}
> **else**
>   *statement*; {*statement*;}
> **endif**

In this case, the first statement list is executed if the first boolean expression evaluates to "true." The second statement list is executed if the first boolean expression is false and the second true. The third statement list is executed only if the first and second boolean expressions are false and the third evaluates to "true." Finally, if none of the boolean expressions is true, then the fourth statement list is executed. After any of the statement lists is executed, control passes to the statement following the **endif**.

Two or more **if** statements can be nested. In such cases, each **if** statement must be closed with its own **endif**.

If an error occurs during the evaluation of an **if** statement condition, the database procedure terminates and control returns to the calling application. This is true even if the statement is nested.

2.17.4 Example

This **if** statement performs a delete or an insert and checks to make sure the statement succeeded.

```
IF (id > 0) AND (id <= maxid) THEN
    DELETE FROM emp WHERE id = :id;
    IF iierrornumber > 0 THEN
        MESSAGE 'Error deleting specified row';
        RETURN 1;
    ELSEIF iirowcount = 0 THEN
        MESSAGE 'Specified row does not exist';
        RETURN 2;
    ENDIF;
ELSEIF (id < maxid) THEN
    INSERT INTO emp VALUES (:name, :id, :status);
    IF iierrornumber > 0 THEN
        MESSAGE 'Error inserting specified row';
        RETURN 3;
    ENDIF;
ELSE
    MESSAGE 'Invalid row specification';
    RETURN 4;
ENDIF;
```

2.18 insert

2.18.1 Purpose

Insert rows into a table.

2.18.2 Syntax

insert into *tablename* [(*column* {, *column* })]
    [**values** (*expr*{, *expr*})] | [*subselect*]

Either the **values** clause or the *subselect* must appear. See the **select** statement description for the subselect syntax.

2.18.3 Description

The **insert** statement inserts new rows into the specified table. In the **values** form, a single row is inserted; in the subselect form, all rows that result from evaluating the subselect are inserted.

The *n*th expression in the **values** list, or the *n*th expression in the **select** clause of the subselect, corresponds to the *n*th column in the list of column names. That is, the **values** list must have a value for each column explicitly or implicitly specified by the **into** clause. The values must be listed in an order corresponding to the order of the columns in the table which the value is being stored. Omitting the list of column names is allowed when a subselect is used and the column names in the subselect match column names in the table, or if the **values** list corresponds exactly to the columns in the table.

What happens in columns not specified in the column list depends on the format used when the table was created with **create table**. If the column was set **with null**, null is assigned. If the column is set **not null with default**, the appropriate default value (0 for numeric formats and spaces for character formats) is assigned. Otherwise, an error code is returned and the insert is not executed.

Expressions used in the **values** clause can only be constants (including the **null** constant), scalar functions on constants or arithmetic operations on constants.

An **insert** statement may be issued only by the owner of the table or by a user with **insert** permission on the table.

Inserted data must be appropriate (valid) given the data type of the target column. For example, both must be numeric types or both must be character types.

Some common errors to watch for are:

- Use of a numeric expression to set the value of a string column or use of a string expression to set the value of a numeric column.
- Failure to specify a value for a column that is set to **not null not default**.
- An attempt to insert the **null** constant into a non-nullable column.

2.18.4 Examples

Add a row to an existing table, "emp."

```
insert into emp (name, sal, bdate)
    values ('Jones, Bill', 10000, 1944);
```

Insert into the "job" table all rows from the "newjob" table where the job title is not "Janitor."

```
insert into job (jid, jtitle, lowsal, highsal)
select   job_no, title, lowsal, highsal
from     newjob
where    title != 'Janitor';
```

Add a row to an existing table, using the default columns.

```
insert into emp
    values ('Jones, Bill', 10000, 1944)
```

2.19 message

2.19.1 Purpose

Return a message number, message text, or both to the executing application from a database procedure.

2.19.2 Syntax

message *message_text* | *message_number* | *message_number message_text*

2.19.3 Description

This statement can only be specified as part of a database procedure. (See the **create procedure** statement for details about issuing statements within procedures.)

The parameter *message_text* can be a string literal or a non-null local character variable or parameter. The parameter *message_number* can be an integer or a non-null local integer variable or parameter. Neither *message_text* nor *message_number* can be an expression. Both the *message_text* and the *message_number* are supplied by the database procedure programmer; they do not correspond in any way to the ULTRIX/SQL error codes and associated messages.

When a **message** statement is issued, the default behavior is to display the arguments (*message_number, message_text*) on the screen, which is similar to using a **printf** C language statement. However, if a **message** statement is issued without any *message_text* and the *message_number* is zero (either a literal 0 or a local variable whose value is 0), a blank line is displayed.

An application may override the default behavior by using the embedded SQL **whenever** statement. If you are using a forms system or forms interface with your embedded SQL program (for example, "Curses"), you will need to override the default behavior. Consult the *ULTRIX/SQL Reference Guide to Embedded SQL* for more details about the **whenever** statement and processing procedure messages.

2.19.4 Examples

This fragment returns trace text to the application.

```
MESSAGE 'Inserting new row';
INSERT INTO tab VALUES (:val);
MESSAGE 'About to commit change';
COMMIT;
MESSAGE 'Deleting newly inserted row';
DELETE FROM tab WHERE tabval = :val;
MESSAGE 'Returning with pending change';
RETURN;
```

This example returns a message number to the application. The application can then extract the international message text out of a message file.

```
IF iierrornumber > 0 THEN
    MESSAGE 58001;
ELSEIF iirowcount != 1 THEN
    MESSAGE 58002;
ENDIF;
```

2.20 modify

2.20.1 Purpose

Convert the storage structure of a table or index. Also used to relocate and reorganize data in locations.

2.20.2 Syntax

modify *tablename* | *indexname* **to** *storage_structure* | *verb* [**unique**]
[**on** *columnname* [**asc** | **desc**]{, *columnname* [**asc** | **desc**]}]
[**with** *with_options_list*]

A *with_options_list* consists of a comma-separated list of any number of the following items:

fillfactor=*n*
minpages=*n*
maxpages=*n*
leaffill=*n*
nonleaffill=*n*
newlocation=(*loc1*[, *loc2*[, *loc3* ...]]),
oldlocation=(*loc1*[, *loc2*[, *loc3* ...]]),
location=(*loc1*[, *loc2*[, *loc3* ...]]),

2.20.3 Description

The **modify** statement changes *tablename* or *indexname* to the specified storage structure, reorganizes a **btree** index, or moves a table to two or more different locations. (See the section entitled "Multi-File System Databases" in Chapter 1 for information on the **to reorganize** and **to relocate** options of the **modify** statement.) This statement is used to accelerate performance of queries that access the table, particularly when the table is large or frequently referenced. Only the owner of a table can modify that table.

Any **modify** statement that involves sorting requires additional temporary disk space to execute. For instance, **modify to btree** can require up to three times the space occupied by the original table or index while executing.

The parameter *storage_structure* can be any of the following:

isam — Indexed sequential access method structure. Duplicate rows are allowed unless the **with noduplicates** clause is specified when the table is created.

cisam — Compressed **isam**. Duplicate rows are allowed unless the **with noduplicates** clause is specified when the table is created.

hash — Random **hash** storage structure. Duplicate rows are allowed unless the **with noduplicates** clause is specified when the table is created.

chash	Compressed **hash**. Duplicate rows are allowed unless the **with noduplicates** clause is specified when the table is created.
heap	Unkeyed and unstructured. Duplicate rows are allowed, even if the **with noduplicates** clause is specified when the table is created.
cheap	Compressed **heap**. Duplicate rows are allowed, even if the **with noduplicates** clause is specified when the table is created.
heapsort	**Heap** with rows sorted. Duplicate rows are allowed unless the **with noduplicates** clause is specified when the table is created. (The sort order is not retained if rows are added or replaced.)
cheapsort	Compressed **heapsort**. Duplicate rows are allowed unless the **with noduplicates** clause is specified when the table is created. (The sort order is not retained if rows are added or replaced.)
btree	Dynamic tree-structured organization. Duplicate rows are allowed unless the **with noduplicates** clause is specified when the table is created.
cbtree	Compressed **btree**. Duplicate rows are allowed unless the **with noduplicates** clause is specified when the table is created.

Verb can be any of the following:

merge	Special form of **modify** for **btree** and **cbtree** storage structures. Modifies the tree structure only, merging adjacent pages whenever possible and deleting empty pages.
relocate	Moves a table or portion of a table from the locationss listed in the *oldlocation* list to the locations specified in the *newlocation* list.
reorganize	Spreads the contents of the table over the locations in the location list.
truncated	Special form to delete all rows quickly and release all file space back to the operating system. Automatically converts structure to **heap**.

The current compression algorithm suppresses trailing blanks in columns of the **c** data type.

The key word **unique** may be used with the following storage structures:

isam	**hash**	**btree**
cisam	**chash**	**cbtree**

The **unique** key word has the effect of requiring each key value in the table to be unique. (A key value is the concatenation of all key columns in a row.) If you try to use the **unique** key word for a table containing non-unique keys, ULTRIX/SQL returns an error message and does not change the storage structure.

Keys, whether unique or not, may be defined on nullable columns. For determining "uniqueness" on a key, on a column or on a whole row, null values are considered equal to other null values. Therefore, if you define keys on nullable columns, use **btree**, since the duplicate null values will create overflow chains in **isam** and **hash** tables, making them very inefficient.

For determining the ordering of values in a column, null values are considered "greater than" all non-null values.

If the **on** phrase is omitted when modifying to **isam, cisam, hash, chash, btree** or **cbtree**, the table will automatically be keyed on the first column. When modifying to **heap** or **cheap**, the **on** phrase is meaningless and must be omitted. When modifying to **heapsort** or **cheapsort**, the **on** phrase is optional.

When a table is sorted (**isam, cisam, heapsort, cheapsort, btree** and **cbtree**) the primary sort keys are those specified in the **on** phrase (if any). The first key (*columnname*) after the **on** phrase is the most significant sort key, and each successive *columnname* specified is the next most significant sort key. Any columns not specified in the **on** phrase will be used as least significant sort keys in column number sequence.

When a table is modified to **heapsort** or **cheapsort**, the sort order can be specified as **asc** (ascending) or **desc** (descending). The default sort order is ascending.

Fillfactor specifies the percentage (from 1 to 100) of each primary data page that should be filled with rows, under ideal conditions. **Fillfactor** may be used with **isam, cisam, hash, chash, btree** and **cbtree**. When modifying to **btree** or **cbtree, nonleaffill** determines the percentage of each index page to fill. Care should be taken when specifying large fill percentages for primary data pages, because a non-uniform distribution of key values could later result in overflow pages and thus degrade access performance for the table.

Minpages specifies the minimum number of primary pages a **hash** or **chash** table must have. **Maxpages** specifies the maximum number of primary pages a **hash** or **chash** table may have. **Minpages** and **maxpages** must be at least one. If both **minpages** and **maxpages** are specified in a **modify** statement, **minpages** cannot exceed **maxpages**.

Default values for **fillfactor, minpages** and **maxpages** are as follows:

Structure	Fillfactor	Minpages	Maxpages
hash	50	16	no limit
chash	75	1	no limit
isam	80	---	---
cisam	100	---	---
btree	80	---	---
cbtree	100	---	---

The **leaffill** parameter of the **modify** statement applies only to tables stored in **btree** and **cbtree** structures. The **leaffill** parameter specifies the percentage to fill each index page for a **btree** or **cbtree** table.

The **leaffill** specifies a percentage *n*, where *n* ranges from 1 to 100, and its percentage specifies how much each *index page* should be filled at the time the table is modified to **btree** or **cbtree**. This parameter contrasts with the **fillfactor** parameter, which specifies the percentage occupancy of primary *data pages* (not *index pages*) when a table is converted to **btree** or **cbtree**.

The **leaffill** parameter allows you to control locking contention in **btree** and **cbtree** index pages. By retaining a percentage of open space on these index pages, more concurrent users can access the **btree** without contention while their queries descend the index tree. Note, however, that you must strike a balance between preserving space in index pages and creating a greater number of index pages; more levels of index pages require more I/O to locate a data row.

The default value for **leaffill** is 70 (percent). This default applies to both **btree** and **cbtree** indexes.

Newlocation and **oldlocation** are used only when the **relocate** verb is used. If you use **reorganize**, you must use the **location** option. Chapter 1 discusses the **reorganize** and **relocate** verbs in the section "Multi-Location Tables." Refer to that section for more information about these verbs and **with** clause options.

ULTRIX/SQL storage structures use existing data to build the index (for **isam** and **cisam**), the **hash** function (for **hash** and **chash**) or for sorting (**heapsort** and **cheapsort**). Therefore, it is pointless to modify a table to any of these six structures *before* adding data to the tables. You are strongly encouraged to add all data to a table as a **heap** *before* modifying a table to these structures. Then, after the table contains its data, run **modify** to optimize storage for retrievals. If you add, delete or change the data in the table significantly (affecting, for instance, 20% of the data), run **modify** again to re-optimize storage. If the table is dynamically used as part of an ongoing application, periodically re-optimize it with the **modify** statement. If the table is merely a static repository for data, this maintenance procedure is not needed.

When data is added to a table stored as a **btree** or **cbtree**, the **btree** index automatically expands, so there should be no need to remodify a growing **btree** index. (See the *ULTRIX/SQL Database Administrator's Guide* for more information on automatic expansion.) However, a **btree** index does not shrink when rows are deleted from the **btree** table.

A special form of **modify**—**modify** *tablename* **to merge**—can be used to shrink a **btree** index after you have deleted a significant number of rows from the **btree** table. Because this form of **modify** affects only the index, it usually runs a good deal faster than a normal **modify** statement. This form of **modify** does not require any temporary disk space to execute.

When **modify** is run on a table, any indexes created for the table are destroyed and must be recreated (except for **modify to merge**). For more information on indexes, refer to the **create index** statement description.

2.20.4 Examples

Modify the "employee" table to an indexed sequential storage structure with "eno" as the keyed column.

```
modify employee to isam on eno;
```

If "eno" is the first column of the "employee" table, the same result can be achieved by the following statement.

```
modify employee to isam;
```

Perform the same **modify** statement, but request a 60% occupancy on all primary pages.

```
modify employee to isam on eno with fillfactor = 60;
```

Modify the "job" table to compressed hash storage structure with "jid" and "salary" as keyed columns.

```
modify job to chash on jid, salary;
```

Perform the same **modify** statement, but also request 75% occupancy on all primary pages, a minimum of seven primary pages and a maximum of 43 primary pages.

```
modify job to chash on jid, salary with fillfactor = 75,
    minpages = 7, maxpages = 43;
```

Perform the same **modify** statement again, but only request a minimum of 16 primary pages.

```
modify job to chash on jid, salary with fillfactor = 75,
    minpages = 16;
```

Modify the "dept" table to a heap storage structure and move it to a new location.

```
modify dept to heap;
modify to relocate
    with oldlocation = (loc_1),
         newlocation = (loc_2);
```

Modify the "dept" table to a heap again, but have rows sorted on the "dno" column.

```
modify dept to heapsort on dno;
```

Modify the "employee" table to heapsort in ascending order by "ename," descending order by "age," and have any duplicate rows removed.

```
modify employee to heapsort on ename, age desc;
```

Modify the "employee" table to **btree** on "ename" so that data pages are 50% full and index pages are initially 40% full.

```
modify employee to btree on ename
    with fillfactor = 50, leaffill = 40;
```

Modify to reorganize the "employee" table from a single-location table to a multi-location table, spread over three locations.

```
modify dept to reorganize
    with location = (loc_2, loc_3, loc_4);
```

2.21 return

2.21.1 Purpose

Terminate a currently executing database procedure and return control to the calling application, and optionally, return a value.

2.21.2 Syntax

return [*return_status*]

2.21.3 Description

This statement can only be specified as part of a database procedure. (See the **create procedure** statement for details about issuing statements within procedures.) In a database procedure, the **return** statement terminates the procedure and returns control to the application. The calling application resumes execution at the statement following **execute procedure.**

The **return** statement can return a value to the application that executed the procedure using the *return_status*. The *return_status* must be a non-null integer constant, variable or parameter, whose data type is compatible with the data type of the field to which its value will be assigned upon the return. If the *return_status* is not specified or if a **return** statement is not executed, then the 0 value is returned to the calling application.

The **into** clause of the **execute procedure** statement allows the calling application to retrieve the *return_status* once the procedure has finished executing. Consult the *ULTRIX/SQL Reference Guide to Embedded SQL* for information about the **execute procedure** statement.

2.21.4 Example

This fragment of a database procedure returns a passed parameter to the calling application.

```
CREATE PROCEDURE CHECK (okval INTEGER, failval INTEGER) AS
BEGIN
     ...
     IF (iierrornumber = 0) THEN
          COMMIT;
          RETURN :okval;
     ELSE
          ROLLBACK;
          RETURN :failval;
     ENDIF;
END;
```

2.22 rollback

2.22.1 Purpose

Roll back the current transaction.

2.22.2 Syntax

rollback [**work**]

2.22.3 Description

This statement erases all of the current transaction. The optional word **work** has no effect. It is included for compatibility with other versions of SQL.

2.23 save

2.23.1 Purpose

Indicate an expiration date for a base table.

2.23.2 Syntax

save *tablename* [**until** *month day year*]

2.23.3 Description

The **save** statement is used to indicate an expiration date for a table. Only the owner of a table can specify an expiration date for a table. User tables, when created, default to "no expiration date."

The *month* parameter can be an integer from 1 through 12, or the name of the month, either abbreviated or spelled out. The *day* parameter is simply the day of the month, and the *year* parameter is the fully specified year (that is, 1982 or 1999).

If the optional **until** clause is omitted, the expiration date is set to "no expiration date," which is the same as the default for table creation.

Tables are not automatically destroyed after their expiration date, and are still accessible. At this time there is no means by which expired tables can be destroyed automatically as a group. However, they can be destroyed individually with the **drop** statement.

System tables have no expiration dates.

2.23.4 Example

Specify February 28, 1991 as the expiration date for the "employee" table.

```
save employee until feb 28 1989;
```

2.24 select

2.24.1 Purpose

Retrieve values from one or more tables.

2.24.2 Syntax

subselect
{**union** [**all**] (*subselect*)}
[**order by** *order_column* [**asc** | **desc**] {, *order_column* [**asc** | **desc**]}]

The *subselect* clause has the syntax:

select [**all**|**distinct**] *expression* [**as** *result_column*]
    {, *expression* [**as** *result_column*]}
[**from** *table* [*corr_name*] {, *table* [*corr_name*]}]
[**where** *search_condition*]
[**group by** *column* {, *column*}]
[**having** *search_condition*]

2.24.3 Description

The result of a **select** statement is the union of the results of all subselects in that statement, ordered in accordance with the specifications of the optional **order by** clause.

Duplicate rows are always eliminated if **union** is specified. But if you say **union all**, duplicates are not removed. If you say **union all** once, you must say it for all **unions** within one statement. If **order by** is not specified, the rows of the result table appear in unpredictable order.

Note that all subselects in a **select** statement with **union** must have the same number of columns in their result tables. Also, columns of numeric type cannot be matched with columns of character type.

Each *order_column* in the **order by** clause must consist of either a result column name or an integer constant in the range 1 - *n*, where *n* is the number of columns in the result table of each of the subselects. The *order_column* designations are taken only from the first subselect in a set of subselects that are affected by **union**. The optional key words **asc** and **desc** specify ascending and descending sort sequence, respectively. If neither is specified for a particular column, **asc** is assumed by default.

The key word **distinct**, used in a subselect, indicates that duplicate rows are to be eliminated. If **distinct** is not specified, the subselect defaults to **all**, in which case duplicate rows are *not* eliminated.

The *expressions* in the **select** clause of the subselect can be any expressions constructed in accordance with the rules set forth in Chapter 1. They may also take one of the following forms:

*correlation_name.**	All the columns of the table denoted by *correlation_name*
*table.**	All the columns of *table*

Note that the asterisk (*) is considered a wild card character.

Additionally, you can specify **select * from** *tablenames*, which will return all the columns from all the tables named in the **from** clause.

A *result_column* may be assigned to any *expression* that denotes a single column in the result table (that is, where *expression* does not use the "*" syntax for specifying columns). The specified result column will then appear in the result table as the column heading for the expression. The ability to assign a result column name to an expression is of particular benefit when the expression is not simply a column from a database table. If the expression is such a column, the column heading in the result table will be by default the name of that column. However, when the expression is, for example, a scalar or set function or involves a computation, ULTRIX/SQL will return blanks for the column heading. To override this default, assign the expression an appropriate result column name. The result column name, whether default or explicit, may also be used in the **order by** clause.

The **from** clause is used to specify the tables from which rows are to be selected. An optional correlation name (*corr_name*) may be chosen for each table specified (see Chapter 1 for information about correlation names). If the **from** clause includes more than one table and a column name in the **select** list appears in more than one of the tables in the **from** clause, column names in the **select** statement must be qualified explicitly by a table name or a correlation name. This eliminates ambiguity as to which table a column belongs.

The **from** clause may be omitted if the statement consists only of a **select** clause of a constant expression (see the "Examples" section following this description).

The **where** clause qualifies the selection of rows. Only those rows that satisfy the *search_condition* are selected.

The *columns* in the **group by** clause of the subselect are names of columns from the tables identified in the **from** clause. The groups may be qualified by a **having** clause.

From a conceptual standpoint, the subselect is evaluated in the following manner. First, the Cartesian product of all tables identified in the **from** clause is formed. From that product, rows not satisfying the search condition specified in the **where** clause are eliminated. Next, the remaining rows are grouped in accordance with the specifications of the **group by** clause. Groups not satisfying the search condition in the **having** clause are then eliminated. Finally, the expressions specified in the **select** clause are evaluated. If the key word **distinct** has been specified, any duplicate rows are eliminated from the result table.

If the subselect includes a **group by** clause, each expression in the **select** clause must be *single-valued per group*. That is, the only data items permitted in such an expression are the following:

- grouping columns
- set function references

As usual, however, such terms can be combined using arithmetic operations, or they can be the arguments to scalar functions, and so forth.

If the subselect includes a **having** clause, each expression in that clause must also be single-valued per group. If the **group by** clause is omitted in a subselect with a **having** clause, the entire table is considered to be a single group.

Note

> When **select** is used to display varying-length character columns, two features should be noted. First, the **select** statement pads unused bytes with blanks. Second, nonprinting characters and control characters are displayed as blanks. **Select** assumes that each varying-length character column will require a width of *n* characters on the screen, where *n* is the width specified when the column was created.

Only the table's owner or a user with **select** permission on the table may issue a **select** statement on that table.

2.24.4 Examples

Find all employees who make more than their managers.

```
select   e.ename
from     employee e, dept, employee m
where    e.dept = dept.dno
and      dept.mgr = m.eno
and      e.salary > m.salary;
```

Retrieve all columns for those employees who make more than the average salary.

```
select   *
from     employee
where    salary
         (select avg (salary)
         from employee);
```

Retrieve employee information sorted, with duplicate rows removed.

```
select   distinct e.ename, d.dname
from     employee e, dept d
where    e.dept = d.dno
order    by dname desc, ename;
```

Select lab samples from production and archive tables that were analyzed by lab #12.

```
select   *
from     samples s
where    s.lab = 12
union
select   *
from     archive_samples a
where    a.lab = 12 ;
```

Select the current user name.

```
select dbmsinfo (username);
```

Select a data conversion operation.

```
select dow(date('today') + date('3 days'));
```

2.25 set

2.25.1 Purpose

Set an ULTRIX/SQL session option.

2.25.2 Syntax

set journaling | **nojournaling** [on *tablename*]

set result_structure
  'heap | **cheap** | **heapsort** | **cheapsort** | **hash** | **chash** | **isam** | **cisam** | **btree** | **cbtree'**

set lockmode session | **on** *tablename*
  where [**level** = **page** | **table** | **session** | **system**]
  [, **readlock** = **nolock** | **shared** | **exclusive** | **session** | **system**]
  [, **maxlocks** = *n* | **session** | **system**]
  [, **timeout** = *n* | **session** | **system**]

set [no]**printqry**

set [no]**qep**

set joinop [no]**timeout**

2.25.3 Description

The **set** statement specifies an ULTRIX/SQL run-time option for a single ULTRIX/SQL session. The selected run-time option remains in effect until the end of the ULTRIX/SQL session, using either the ULTRIX/SQL Terminal Monitor or a database invocation within an embedded ULTRIX/SQL program. Alternatively, another **set** statement can change the value of a current run-time option established by a previous **set** statement.

Note

See your *ULTRIX/SQL Operations Guide* for information about changing environment variables.

2.25.4 set journaling | nojournaling

The **set journaling** statement causes all tables created within a session to be logged with the ULTRIX/SQL journaling system. Note, however, that journaling does not take effect until journaling is enabled for the entire database with the **ckpdb** statement. (Refer to Chapter 5 for information about **ckpdb**.)

When you issue the **set journaling** statement, it is not necessary to explicitly specify the **with journaling** clause in the **create table** statement. Also, tables created using the **as** clause of the **create table** statement are logged to the journal. If the **set nojournaling** statement (which is the default) is set, tables are created without logging to the journal, unless the explicit **with journaling** clause appears in the **create table** statement. The **set journaling** statement, when used with an optional table name, causes journaling to begin *at the next checkpoint* for the named table.

2.25.5 set result_structure

The **set result_structure** statement sets the default storage structure for tables created with the **as** clause of the **create table** statement. If the value of **heap** or **cheap** is selected as the default, tables are created exactly as specified in the **select** statement, which may result in duplicate rows. However, performance of the **create table as** statement is best with the **heap** or **cheap** option specified. You can optionally set the default structure of tables created by **create table as** to any of the structures described in the **modify** statement—that is, **heap, cheap, heapsort, cheapsort, hash, chash, btree, cbtree, isam** or **cisam**. For instance, the following two sets of statements do the same thing:

```
set result_structure hash;
create temp as select id ... ;
insert into temp ... ;

create temp as select id ... ;
insert into temp ... ;
modify temp to hash;
```

Both sequences result in the "temp" table being stored in a **hash** structure, hashed on the first column (in this case, "id"); however, the second sequence is much preferred to provide best performance for both creating and loading the table. For **hash, chash, isam** and **cisam** the newly created table is automatically indexed on the first column.

If you do not execute a **set result_structure** statement, the default storage structure for a table created by the **create table as** statement is **cheap**.

2.25.6 set lockmode

You can use the **set lockmode** statement to determine how the ULTRIX/SQL locking system will operate when ULTRIX/SQL accesses data in a table. The **set lockmode** statement allows you to establish a number of different types and levels of locks.

ULTRIX/SQL provides a default strategy for locking in statement processing. (See the *ULTRIX/SQL Database Administrator's Guide* for a more detailed discussion of locking.) If you have no interest in overriding this default, you need not make use of the **set lockmode** statement. The **set lockmode** statement is provided to allow you to optimize performance or enforce stricter validation and/or concurrency controls.

The **set lockmode** statement acknowledges three basic types of locking:

- Locking provided by default by the ULTRIX/SQL system
- Locking instituted for an ULTRIX/SQL session
- Locking specified in an individual instance for a particular purpose

You can switch among any of these three types of locking at any time in your ULTRIX/SQL session, except where specifically disallowed.

The **set lockmode** statement provides four different parameters to govern the nature of locking in an ULTRIX/SQL session:

- **level**
- **readlock**
- **maxlocks**
- **timeout**

The **level** parameter refers to the level of granularity desired when the table is accessed. You can specify any of the following locking levels:

page Specifies locking at the level of the data page (subject to escalation criteria, discussed below under **maxlocks**).

table Specifies table-level locking in the database.

session Specifies the current default for your ULTRIX/SQL session.

system Specifies that ULTRIX/SQL will start with page level locking, unless it estimates that more than *maxlocks* pages will be referenced, in which case table-level locking will be used.

The **readlock** parameter refers to locking in situations where table access is for reading of data only (as opposed to updates of data). You can specify any of the following readlock modes:

nolock Specifies no locking when reading data.

shared Specifies the default mode of locking when reading data.

exclusive Specifies exclusive locking when reading data (useful in "select-for-update" processing within a transaction).

session Specifies the current readlock default for your ULTRIX/SQL session.

system Specifies the general readlock default for the ULTRIX/SQL system.

The **maxlocks** parameter refers to an escalation factor; that is, the number of locks on data pages at which locking escalates from page level to table level. The number of locks available to you is dependent upon your system configuration. You can specify the following maxlocks escalation factors:

n	Specifies a specific (integer) number of page locks to allow before escalating to table level locking. The default is 10. The *n* specified must be greater than 0.
session	Specifies the current maxlocks default for your ULTRIX/SQL session.
system	Specifies the general maxlocks default for the ULTRIX/SQL system. Note that if you specify page level locking and the number of locks granted during a query exceeds the system-wide lock limit, or if the operating system's locking resources are depleted, locking escalates to table level. This escalation occurs automatically and is independent of the user.

The **timeout** parameter refers to a time limit, expressed in seconds, for which a lock request should remain pending. If ULTRIX/SQL cannot grant the lock request within the specified time, then the query that requested the lock aborts. You can specify the following timeout characteristics:

n	Specifies a specific (integer) number of seconds to wait for a lock (setting *n* to 0 requires ULTRIX/SQL to wait indefinitely for the lock).
session	Specifies the current timeout default for your ULTRIX/SQL session (which is also the ULTRIX/SQL default).
system	Specifies the general timeout default for the ULTRIX/SQL system.

Against the backdrop of these **set lockmode** parameters and options are the following ULTRIX/SQL system defaults for each of the parameters:

Parameter	Default
level	dynamically determined by ULTRIX/SQL
readlock	shared
maxlocks	10
timeout	0 (no timeout)

If you select the **system** option for any of the **set lockmode** parameters, the values in the preceding table are automatically supplied. When you begin your ULTRIX/SQL session, the ULTRIX/SQL system defaults are in effect. If you override them with other values using the **set lockmode** statement, you can revert back to the system defaults easily by specifying another **set lockmode** statement that uses the **system** option.

Similarly, if you set session parameters (that is, locking behavior for all user tables accessed by queries in your ULTRIX/SQL session), you can later override those parameters temporarily for individual tables for a specific purpose. After setting the locking behavior for an individual table, you can return the parameters to either the session defaults or the ULTRIX/SQL system defaults.

2.25.7 set [no]printqry

The **set printqry** statement displays each query and its parameters as it is passed to the ULTRIX/SQL database management system for processing. The **set noprintqry** statement disables this feature.

2.25.8 set [no]qep

The **set qep** statement displays a summary of the query execution plan chosen for each query by the optimizer. To disable this option, use **set noqep**.

2.25.9 set joinop [no]timeout

This statement turns the optimizer's timeout feature on and off. With **set joinop [no]timeout** in effect, when the optimizer is checking query execution plans, it stops when it believes that the best plan that it has found would take less time to execute than the amount of time already spent searching for a plan. If you issue a **set joinop notimeout** statement, the optimizer will continue searching query plans, no matter how long it takes. This statement is often used with the **set qep** statement to ensure that the optimizer is picking the best possible query plan.

To return to the default behavior, issue a **set joinop timeout** statement.

2.25.10 Examples

Within an ULTRIX/SQL session, create three tables with journal logging enabled and one without.

```
set journaling;
create table withlog1 ( ... );
create table withlog2 ( ... );
set nojournaling;
create table withlog3 ( ... ) with journaling;
create nolog1 ( ... );
```

Create a few tables with different structures.

```
create table a as ...;            /* heap */
set result_structure 'hash';
create table b as select id ...;  /* hash on 'id' */
set result_structure 'heap';
create table d as select id ...;  /* heap */
```

Set lockmode parameters for your ULTRIX/SQL session to the desired values. Tables accessed after executing this statement are governed by these locking behavior characteristics.

```
set lockmode session where level = page, readlock = nolock,
    maxlocks = 50, timeout = 10;
```

Set the lockmode parameters explicitly for a particular table.

```
set lockmode on employee
    where level = table, readlock = exclusive,
    maxlocks = session, timeout = 0;
```

Reset your ULTRIX/SQL session default locking characteristics to the ULTRIX/SQL system defaults.

```
set lockmode session where level = system, readlock = system,
    maxlocks = system, timeout = system;
```

2.26 update

2.26.1 Purpose

Update values of columns in a table.

2.26.2 Syntax

update *tablename* [*corr_name*]
set *columnname* = *expression* {, *columnname* = *expression*}
[**where** *search_condition*]

2.26.3 Description

The **update** statement replaces the values of the specified columns by the values of the specified expressions for all rows of the table that satisfy the *search_condition*. The expressions in the **set** clause may only use constants or columns from the table specified by *tablename*.

Only the owner of the table or a user with **update** permission on the table is allowed to **update** a table. If a given row update would violate an integrity constraint on the table, that row remains unchanged.

Numeric columns may be updated by values of any numeric type. Update values are converted to the data type of the columns being updated. Character-string columns may be updated by values of any character-string data type. You cannot use a string expression to set the value of a numeric column, nor a numeric expression to set the value of a character-string column. Nullable columns may be set to null by using the **null** constant.

Note

If the table was created with no duplicates allowed, be careful not to issue an update statement that creates duplicate rows. ULTRIX/SQL returns an error in such cases.

2.26.4 Examples

Give all employees who work for Smith a 10% raise.

```
update   emp
set      salary = 1.1 * salary
where    dept in
         (select  dno
         from     dept
         where    mgr in
              (select  eno
              from     emp
              where    ename = '*Smith'));
```

Set all salaried for people who work for Smith to null.

```
update   emp
set      salary = null
where    dept in
         (select  dno
         from     dept
         where    mgr in
             (select  eno
             from     emp
             where    ename = '*Smith'));
```

2.27 while - endwhile

2.27.1 Purpose

Repeat a series of statements while a specified condition is true.

2.27.2 Syntax

```
[label:]   while boolean_expr do
                statement; {statement;}
           endwhile
```

2.27.3 Description

This statement can be specified only as part of a database procedure. (See the **create procedure** statement for details about issuing statements within procedures.)

A boolean expression (*boolean_expr*) must always evaluate to "true" or "false." A boolean expression can include comparison operators (=, >, and so on) and the logical operators **and**, **or** and **not**.

The statement list may include any series of legal database procedure statements, including another **while** statement.

As long as the condition represented by the boolean expression remains true, the series of statements between **do** and **endwhile** is executed. The condition is tested only at the start of each loop. If values change inside the body of the loop so as to make the condition false, execution will still continue through the current iteration of the statement list, unless an **endloop** statement is encountered, as shown by the following formats.

The **endloop** statement may be used to break out of a **while** loop before the **endwhile** statement is encountered. When **endloop** is executed, the loop is immediately closed, and procedure execution continues with the first statement following **endwhile**.

```
while condition_1 do
     statement_list_1
     if condition_2 then
          endloop;
     endif;
     statement_list_2
endwhile;
```

In this case, if *condition_2* is true, *statement_list_2* is not executed in that pass through the loop, and the entire loop is closed. Execution resumes at the statement following the **endwhile** statement.

A **while** statement may also be labeled to allow an **endloop** to break out of a nested series of **while** statements to a specified level. The label precedes **while** and is specified by a unique alphanumeric identifier followed by a colon, as in:

```
A:  while
```

The label must be a legal ULTRIX/SQL name (see Chapter 1). The **endloop** statement uses the label to indicate which level of nesting to break out of. One possible way to use labels in nested **while** statements is:

```
label_1:  while condition_1 do
              statement_list_1
label_2:      while condition_2 do
                  statement_list_2
                  if condition_3 then
                      endloop label_1;
                  elseif condition_4 then
                      endloop label_2;
                  endif;
                  statement_list_3
              endwhile;
              statement_list_4
          endwhile;
```

In this example, there are two possible breaks out of the inner loop. If *condition_3* is true, both loops are closed, and control resumes at the statement following the outer loop. If *condition_3* is false but *condition_4* is true, the inner loop is exited and control resumes at *statement_list_4*.

If no label is specified after an **endloop**, only the innermost loop that is currently active will be closed.

If an error occurs during the evaluation of a **while** statement, the database procedure terminates and control returns to the calling application.

2.27.4 Example

The "delete_n_rows" database procedure accepts as input a base number and a number of rows. The specified rows are deleted from the table "tab," starting from the base number. If an error occurs, then the loop terminates.

```
CREATE PROCEDURE delete_n_rows
    (base INTEGER, n INTEGER) AS
DECLARE
    limit INTEGER;
    err INTEGER;
BEGIN
    limit = base + n;
    err = 0;
    WHILE (base < limit) DO
        DELETE FROM tab WHERE val = :base;
        IF iierrornumber > 0 THEN
            err = 1;
            ENDLOOP;
        ENDIF;
        base = base + 1;
    ENDWHILE;
    RETURN :err;
END;
```

Terminal Monitor Command Line Interface to ULTRIX/SQL 3

3.1 Introduction

There are two versions of the Terminal Monitor user interface to interactive ULTRIX/SQL–a command line interface (**sql**) and a forms-based interface (**isql**). This chapter discusses primarily the command line interface, while Chapter 4 discusses the forms-based interface. In this and other chapters, assume that a discussion of Terminal Monitor capabilities applies both to the **sql** and **isql** interface systems. The interface implementation differs in the two environments, but the database operations are the same.

The **sql** command line interface allows you to create, store, print, edit, and execute a query by entering special commands preceded by a backslash (\ ) on the **sql** command line. After executing the query, you can either enter a new query or edit the existing query with a text editor if minor changes are to be made. You can also read or write files containing statements and execute operating system level commands from within the Terminal Monitor environment.

The command line interface to ULTRIX/SQL is invoked by typing the system level command **sql** at your terminal. (See the **sql** command description in Chapter 5 for details). You can then type a single query, type **\g** (for go) to process the query, and see the results of the query at your terminal. By typing additional queries, followed by **\g**, any of the capabilities of ULTRIX/SQL can be invoked. To exit the ULTRIX/SQL command line interface, type **\q** (for quit).

3.2 Messages, Prompts and Diagnostics

The Terminal Monitor gives a variety of messages, prompts and diagnostics to keep the user informed of the status of the monitor and the query buffer, as summarized below.

Message	Description
login message	Typically provides the version number and login time when a user logs onto the Terminal Monitor. This is followed by the contents of the dayfile, which provides other pertinent information.
go	The Terminal Monitor is empty and ready to accept input. (You may begin a new query.)

Message	Description
continue	The previous query is finished and you are back in the monitor.
*	An asterisk is printed at the beginning of each line as the prompt character.
Executing . . .	The query is being processed by ULTRIX/SQL.
>editor	You have entered the text editor.
Non-printing character *nnn* converted to blank	ULTRIX/SQL maps non-printing ASCII characters into blanks. This message indicates that one such conversion has been made.

When the Terminal Monitor query buffer is empty and ready to accept input, the message **go** is printed. The message **continue** means there is something in the query buffer. After a **\go** command the query buffer is cleared if another query is typed in, unless a command that affects the query buffer is typed first. Commands that retain the query buffer contents are:

\a or **\append**
\e or **\edit**
\p or **\print**
\bell
\nobell

For example,

```
help parts
\go
print parts
```

results in the query buffer containing:

```
print parts
```

However,

```
help parts
\go
\print
print parts
```

results in the query buffer containing:

```
help parts
print parts
```

3.3 Terminal Monitor Commands

A number of commands may be entered by the user to manipulate either the contents of the query buffer or the user's environment. They are all preceded by a backslash (\) and all are executed immediately (rather than at query execution time).

Some commands may take a file name. In such commands, the file name is designated by a string beginning with the first significant character following the command, and ending at the end of the line. No other commands may be entered on a line with a command that contains a file name. Commands that *do not* take a file name may be concatenated on a single line. The following example, for instance, returns the time both before and after execution of the current query buffer.

```
\date\go\date
```

The following table provides a summary of the Terminal Monitor commands:

Command	Description
\a or **\append**	Append to the query buffer. Typing **\append** after completion of a query overrides the auto-clear feature and guarantees that the query buffer will not be reset until executed again.
\bell and **\nobell**	Tell the Terminal Monitor to include or *not* to include a bell (that is, **Control-G**) with the **continue** or **go** prompt. The default is **\nobell.**
\cd or **\chdir** *dir_name*	Change the working directory of the monitor to the named directory.
\date or **\time**	Display the current date and time.
\e or **\ed** or **\edit** or **\editor** [*filename*]	Enter the operating system's text editor (designated by the ULTRIX/SQL startup file). Use the appropriate editor command to return to the ULTRIX/SQL monitor. If no file name is given, the current contents of the query buffer are sent to the editor, and upon return, the query buffer is replaced with the edited query. If a file name is given, the query buffer is written to that file. On exit from the editor, the file contains the edited query, but the query buffer remains unchanged.
\g or **\go**	Process the current query. The contents of the buffer are transmitted to ULTRIX/SQL and the query is executed.
\i or **\include** or **\read** *filename*	Read the named file into the query buffer. Backslash characters in the file are processed as they are read.
\p or **\print**	Print the current query. The contents of the buffer are displayed on the user's terminal.
\q or **\quit**	Exit the ULTRIX/SQL Terminal Monitor.
\r or **\reset**	Erase the entire query (reset the query buffer). The former contents of the buffer are lost and cannot be retrieved.
\s or **\sh** or **\shell**	Access the shell (ULTRIX command line interpreter). Pressing **Control-D** causes you to exit the shell and return to the Terminal Monitor command line.

Command	Description
\script [*filename*]	Write or stop writing the subsequent SQL statements and their results to the specified file. If no file name is supplied with the **\script** command, output is logged to a file called "script.ing" in the current directory. The **\script** command toggles between logging and not logging your ULTRIX/SQL session to a file. If you supply a *filename* on the **\script** command that terminates logging to a file, the *filename* is ignored. You can use this command to save result tables from SQL statements for output. The **\script** command in no way impedes the terminal output of your session.
\w or \write *filename*	Write the contents of the query buffer to the named file.
any-other-character	Ignore any possible special meaning of a character following the backslash (\). This command allows the backslash itself to be inserted as a literal character. (See also Chapter 1 on character strings).

3.4 Flags

Certain flags may be included on the **sql** command line. These flags affect the operation of the Terminal Monitor. Among the most useful of these flags are:

Flag	Description
-a	Disables the autoclear function. This means that the query buffer is never automatically cleared; it is as though the **\append** command were inserted after every **\go**. Note that this flag requires the user to clear the query buffer explicitly
-d	Turns off printing the dayfile.
-s	Turns off printing of all messages (except errors) from the monitor, including the login and logout messages, as well as the dayfile and prompts. It is used for executing "canned queries"—that is, queries redirected from files.

For a complete list of flags available with the **sql** command, consult Chapter 5.

Forms-Based Interface to ULTRIX/SQL 4

4.1 Overview

There are two versions of the Terminal Monitor user interface to interactive ULTRIX/SQL–a command line interface (**sql**) and a forms-based interface (**isql**). This chapter discusses the forms-based interface. (See Chapter 3 for a discussion of the **sql** command line interface.)

The **isql** forms-based interface allows you to enter, edit, save and execute queries by selecting menu options and entering text on a form displayed on the screen. You enter the database query in the displayed form, using ULTRIX/SQL statements, and then press a function key or make a choice from a menu on the form to execute the query.

The ULTRIX/SQL forms-based interface includes a fullscreen editor for entering and editing ULTRIX/SQL statements. When you execute a statement, ULTRIX/SQL immediately displays the result on the screen. If the statement cannot be executed, a detailed error message appears. Context-sensitive Help screens are also available.

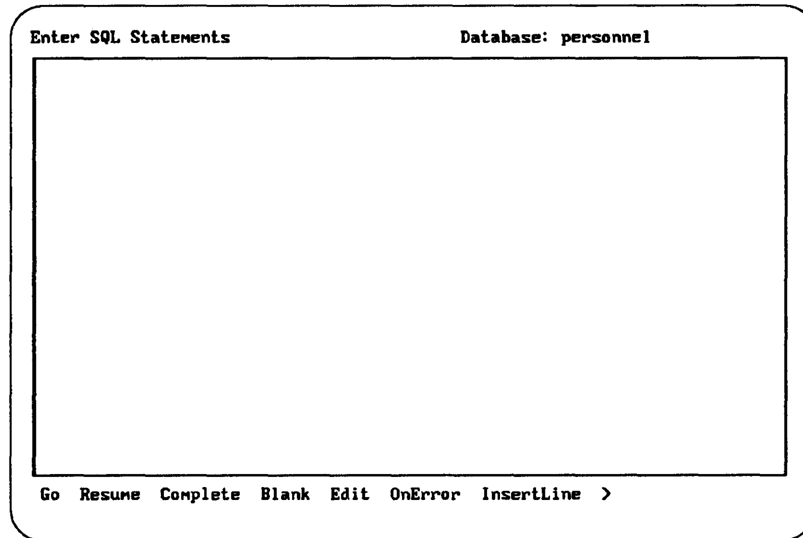
To access ULTRIX/SQL using the forms-based interface, you enter the **isql** command at the operating system prompt. (For details about command syntax, see Chapter 5.)

Before you can use the forms-based interface to interactive ULTRIX/SQL, you must define your terminal to ULTRIX/SQL. For information on how to do this, refer to Appendix D, "Defining Your Terminal."

4.2 Entering ULTRIX/SQL Statements

The interactive ULTRIX/SQL (ISQL) forms system frame (Figure 4-1) consists of a blank window and a menu of operations. You enter and edit SQL commands in the window, which is your "work space." The cursor is initially positioned in the window, ready for you to begin typing an SQL statement, when the **isql** display frame first appears.

Figure 4-1 The isql Display Frame



4.2.1 isql Menu Items

The **isql** operations menu contains more menu items than can be shown on the display frame at one time. The complete list of **isql** menu operations is shown below. For details on how to use menus and other features of a forms-based utility, refer to Appendix C.

Operation	Function
Go	Executes the SQL statements and immediately begins displaying the results.
Resume	Resumes the display of your current statements at your last work position.
Complete	Executes the statements in the work space, but does not display the results until all processing is completed. This command displays the end of the query's output.
Blank	Clears the work space of any statements you have entered.
Edit	Edits the statements in the work space with the standard system editor.
File	Calls the submenu of file operations either to write the information that is on the screen to a file or to read information from an existing file.
OnError	Indicates whether SQL statement processing continues or terminates on errors, and lets you change the setting.
InsertLine	Inserts a blank line above the current cursor position in the input screen.
DeleteLine	Deletes the line at the current cursor position in the input screen. If there is no current line or only one line, no line is deleted.

Operation	Function
Help	Gets help about this frame, including help about the syntax and usage of the interactive query language.
Quit	Leaves interactive SQL.

4.2.2 Help

The **Help** operation includes complete summaries of how to use SQL statements. After selecting **Help,** you can search for database language key words to find helpful hints on syntax and usage. For information on searching for text in a help screen, see Appendix C, "Using Forms-Based Applications."

4.3 Input/Output Screens

When you start up the forms-based interface to interactive ULTRIX/SQL, the interactive database environment gets its instructions from an *input* screen. When you execute SQL statements, **isql** displays the results on an *output* screen. Each of these screens is described below.

4.3.1 Input Screen

The input screen is for entering SQL statements. When cursor appears in the upper left corner of the work space, you can begin typing SQL statements. Because the work space is a single-column table field, all the cursor movement key strokes and forms-based operations and functions are available here.

You can enter statements directly into the work space. The input screen retains all the statements you enter in the work space unless you explicitly edit or clear them. You can also use the default editor on your computer system by selecting the **Edit** operation from the main menu.

The **Edit** operation writes the work space contents to a temporary file and invokes your text editor on that temporary file. When you finish editing the temporary file and exit from the editor, the newly edited work space is restored to **isql**. You can then execute the SQL statements or continue typing additional ones.

You are not limited to SQL statements that you enter directly into the work space by typing. You can also enter SQL statements from text files.

- If you have written a series of statements into a file that you wish to load into the work space, select the **File** operation and then select the **Read** operation.

- To save the script you have written in the work space, select the **File** operation and then select the **Write** operation.

4.3.1.1 Loading a File (Read)

To load a file:

1. Select the **Read** operation.

   Then **isql** prompts you for the filename:

   ```
   Enter name of file to read:
   ```

2. Type the filename. If the file to be read is not in the current working directory, you must include the full pathname designation.

Isql reads in the contents of the file you specified. If you have already entered database language commands into the work space, the file you read in is added at the bottom of the current work space contents.

4.3.1.2 Writing to a File (Write)

To write the contents of your work space into a file:

1. Select the **Write** operation.

   A prompt appears:

   ```
   Enter name of file to be written:
   ```

2. Type the name of the file to which you want your work saved.

Isql writes the work space contents to the specified file while preserving the work space contents.

4.3.1.3 Clearing the Work Space (Blank)

You can clear the work space of its contents by selecting the **Blank** operation. The table field is erased and you can enter new requests or leave the interactive database language environment.

4.3.2 Output Screen

After specifying SQL statements in the work space, you can execute them. Requests that are executed result in output.

Assume that your SQL statement retrieves all the rows from the "emp" table in the "personnel" database. The SQL statement would be:

```
select * from emp
```

When you enter the preceding statement in the work space of the input screen and select the **Go** or **Complete** command, **isql** displays the message:

```
Run the request
```

Then the output screen is displayed, as in Figure 4-2.

Figure 4-2 isql Output Mode

```
Start of Output                                              Column 1/80  Line 1

  1> select * from emp
```

name	title	hourly_rate	manager
Alcott, Scott	Sr Programmer	$50.00	Wolfe, Neal
Applegate, Donald	Analyst	$51.00	Wolfe, Neal
Bee, Charles	Sr Programmer	$43.00	Fielding, Wallace
Belter, Kris	Programmer	$33.00	Alcott, Scott
Beringer, Tom	Programmer	$41.00	King, Richard
Beveridge, Fern	Project Leader	$57.00	Wolfe, Neal
Bluff, Clarence	Programmer	$24.00	Jones, Ashley
Bridges, Debra	Sr Programmer	$48.00	Parsons, Carol
Chung, Arthur	Programmer	$21.00	Ortega, Julio
Downing, Susan	Programmer	$29.00	Bee, Charles
Fielding, Wallace	Project Leader	$47.00	Jones, Betty
Fine, Laurence	Sr Programmer	$42.00	Jones, Betty
Hilton, Connie	Programmer	$37.00	Bridges, Debra
Jones, Ashley	Sr Programmer	$49.00	Turner, Russell
Jones, Betty	Project Leader	$66.00	
King, Richard	Sr Programmer	$39.00	Beveridge, Fern

```
  Top  Bottom  File  Help  End
```

4.3.2.1 Output Frame

The following line appears at the top of the output frame:

```
Start of Output  Column 1/80  Line 1
```

The phrase "Start of Output" indicates that the output extends over more than one screen.

The column indicator shows the width of the output, not the number of columns in the table.

Isql may not know how many lines of text are returned when it begins to process the request. Thus, it does not display a range of lines, but rather, the current line ("Line 1" in the above example).

When in the output screen, you can scroll the rows of output up and down or left and right using the terminal-specific keystrokes described in Appendix E.

You reach the end of the output by choosing the **Bottom** operation. The resulting frame is shown in Figure 4-3:

Figure 4-3 The End of Output Frame

```
End of Output                                        Column 1/80  Line 22/41
 King, Richard        Sr Programmer            $39.00 Beveridge, Fern
 Lorenzo, Sue         Consultant               $52.00 Parsons, Carol
 Moore, Holly         Programmer               $36.00 Thompson, Howard
 Noonan, Brad         Programmer               $25.00 Jones, Ashley
 O'Foote, Suzanne     Programmer               $40.00 Bridges, Debra
 Ortega, Julio        Sr Programmer            $50.00 Wolfe, Neal
 Parsons, Carol       Project Leader           $55.00 Wolfe, Neal
 Peterson, Jean       Analyst                  $32.00 Alcott, Scott
 Randall, David       Programmer               $34.00 Alcott, Scott
 Rolls, Richard       Programmer               $28.00 King, Richard
 Smith, Chester       Programmer               $22.00 Bee, Charles
 Smith, Peggy         Consultant               $32.00 Thompson, Howard
 Stein, Frank         Programmer               $27.00 Thompson, Howard
 Thompson, Howard     Sr Programmer            $45.00 Jones, Betty
 Turner, Russell      Project Leader           $53.00 Jones, Betty
 Walters, Lindsay     Analyst                  $44.00 Fine, Laurence
 Wolfe, Neal          Project Leader           $65.00

(32 rows)
End of Request

  Top  Bottom  File  Help  End
```

The "End of Output" message also appears when you:

- Execute the **Go** operation for a request whose output can be displayed in a single screen
- Execute the **Complete** operation, which runs the request to completion and displays the last portion of the output

If the query runs to completion, the output displays the current position of the cursor and the total number of lines in the output in the upper right corner of the screen. For example, in Figure 4-3 "Line 22/41" indicates that the cursor is on line 22 and that there are 41 lines of output including trim, blank lines, and explanatory text. The output screen also indicates that 32 rows were returned.

The output itself contains:

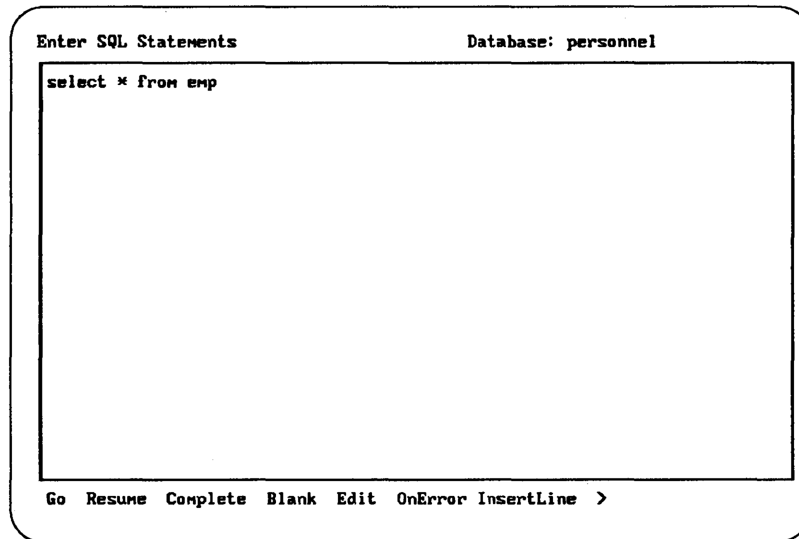
- The SQL statements, with each line preceded with a number and a "greater than" sign (>). In Figure 4-2, the SQL statement is preceded by "1>."
- The data and messages returned by the request.

You can also invoke the **File** command with any key you have assigned to the **printscreen** function (described in Appendix E). If you select this function by pressing its key, the **printscreen** function takes effect. However, when you use the **printscreen** function, the *entire* query results are written to a file, not just the displayed form. As in the **File** command, ULTRIX/SQL prompts you for a filename into which the screen's contents can be written for later output on a suitable device.

4.3.2.2 Returning to the Input Frame

When you return to the input screen (with the **End** command), your original query is displayed as shown in Figure 4-4.

Figure 4-4 Return to Input Mode Frame



You can edit the current request or execute it again with the **Go** or **Complete** commands.

You can also return to the point at which you last inspected the current output with the **Resume** command.

Alternatively, you can clear the work space with the **Blank** command and enter a new request.

4.3.3 Error Messages

If an SQL statement contains errors, **isql** displays an error message. The error message includes information on statement syntax. The erroneous part of the statement may be pointed out as shown in Figure 4-5.

Figure 4-5 Input Error Message

```
End of Output                                                      Line 1/7

  1> slect * from emp
E_US09C4 Syntax error on line 1.  Last symbol read was: 'slect'.
   (28-AUG-1988 18:07:15)

End of Request

---------------------------------------------------------------------------
  Top  Bottom  File  Help  End
```

In Figure 4-5, the message indicates that the **select** statement in line one was typed incorrectly.

If the **OnError** setting was set to Continue, then processing of the remaining statements in the query continues after the error message appears. If the **OnError** setting was set to Terminate, then statement processing will halt when the error message appears, and none of the remaining statements in the query will be processed. (You can change the **OnError** setting, as described below.) Choose **End** to return to the input screen and edit your erroneous entry, or clear (**Blank**) the frame and correctly retype your request.

The default setting for **OnError** is Terminate. You can change this default setting as follows:

1. Select the **OnError** operation from the **isql** main menu.

   A pop-up window offers the following choices and indicates through highlighting whether Continue or Terminate is the current setting:

   ```
   Go           Set options as selected via the highlighted
                row, and return to the ISQL input
                screen.
   Terminate    Set option to terminate and return to the
                ISQL input screen.
   Continue     Set option to continue and return to the
                ISQL input screen.
   Help         Gives Help on this option.
   End          Return to the ISQL input screen without
                changing the option.
   ```

2. Select Continue if you want queries to continue after an error message appears.

   Select Terminate if you want queries to terminate when an error message appears.

You can also set the environment variable II_TM_ON_ERROR to specify the **OnError** setting. For example, in the C shell, you can type either of the following commands at the system prompt (or include one of the commands in your .cshrc file) :

```
setenv II_TM_ON_ERROR "continue"

setenv II_TM_ON_ERROR "terminate"
```

ULTRIX/SQL Operating System Commands 5

5.1 Introduction

A number of ULTRIX/SQL commands are entered at the level of the computer's operating system. These "utility" commands control the overall database organization, its creation, backup, maintenance and the like. Unlike the SQL statements or the Terminal Monitor commands, these commands do not affect the data in the database, but rather act on the database as a whole.

Parameter name conventions in the syntax of these commands are:

dbname — The ULTRIX/SQL database name, which must begin with an alphabetic character and be a maximum of 24 alphanumeric characters, including the underscore (_).

flags — The set of flags used to select special options to the command. Flags are one letter names, preceded by a sign (+ or -) and optionally followed by a parameter value. If the flag name is preceded by both a plus sign (+) and a minus sign (-), as shown below, the option has two settings. The plus sign (+) means to turn the option on; the minus sign (-) means to turn it off.

+ | -x

or

[+x | -x]

If only a minus sign (-) is shown before the flag name, then specification of the minus sign (-) will turn the option on. For example, if a flag is described as **-x**, specifying **-x** invokes the option.

tablename — The name of a table in the database.

username — The login user name for a valid ULTRIX/SQL user.

No command terminator is required by ULTRIX. For this reason, no semicolon is included in the syntax description for ULTRIX/SQL operating system commands.

5.2 accessdb

5.2.1 Purpose

Authorize access to a database and modify database locations, extensions and user authorization.

5.2.2 Syntax

accessdb

5.2.3 Description

The **accessdb** utility allows ULTRIX/SQL superusers to list and modify information about a database, the *locationnames* known to the system, the extensions allowed for databases, and user authorization. It is a full-screen, forms-based, menu-driven utility.

The initial display is a main menu of operations. When you select an operation from the main menu, one of the following appears:

- A form with a submenu
- Another menu with menu items appropriate to the selected operation

To continue, you fill in the form or select another operation. The **Quit** operation exits the utility and returns you to the shell. The **End** operation returns you to the main menu screen. Other operations may also return you to the main menu when the initiated task has been completed. All menus within **accessdb** contain a **Help** operation that provides a brief description of each currently available menu item.

5.2.4 Restrictions

The **accessdb** utility uses a forms-based interface and must be run on a supported video terminal or in a window emulating a supported terminal. The terminal type is made known to ULTRIX/SQL by way of the environmental variable TERM or TERM_INGRES, which must be set to one of the terminal types defined in the $II_SYSTEM/sql/files/termcap file.

The **accessdb** utility can only be used by the ULTRIX/SQL System Administrator or an ULTRIX/SQL superuser. Other ULTRIX/SQL users cannot use **accessdb**, but may obtain similar read-only information by using the **catalogdb** command.

See the *ULTRIX/SQL Operations Guide* and the *ULTRIX/SQL Database Administrator's Guide* for a more complete description of this utility.

5.3 auditdb

5.3.1 Purpose

Audit a database.

5.3.2 Syntax

auditdb [**-b***dd-mmm-yyyy:hh:mm:ss*] [**-e***dd-mmm-yyyy:hh:mm:ss*] [**-f**] [**-i***username*] [**-s**] [**-t***tablename*] [**-u***username*]{*dbname*}

5.3.3 Description

The **auditdb** command allows the user to print selected portions of the journal for a database or to create an ULTRIX/SQL readable audit trail of the changes made to a particular table. The **auditdb** command operates on all journal entries that have been moved to the journal files. The flags are interpreted as follows:

Flag	Description
-b	Print journal entries for ULTRIX/SQL transactions committed after the time specified with the **-b** flag.
-e	Print journal entries for ULTRIX/SQL transactions committed before the time specified with the **-e** flag.
-f	Create a file named audit.trl in your current directory. (Note that you can only use this flag if your table has less than 120 columns and less than 1948 bytes per row.) For more information see the discussion on audit.trl following this table.
-i	Print journal entries for actions taken by the specified user only.
-s	Invoke ULTRIX/SQL superuser status for system-wide access to any database.
-t	Print the journal entries for the table specified with the **-t** flag.
-u	Print the journal, with specified options, for databases owned by the specified user.

The audit.trl file is in binary (bulk copy) format and contains rows appended to, deleted from, or copied into the table specified with the **-t** flag. You can copy this file into a database table that has been created as follows, with **not null** specified in the audit table's seven control columns.

```
create table auditrel
    (date           date not null with default,
    usrname         char(24) not null with default,
    operation       char(8) not null with default,
    tranid1         integer not null with default,
    tranid2         integer not null with default,
    table_id1       integer not null with default,
    table_id2       integer not null with default,
    { columns of tablename} )
```

To copy the file audit.trl into the table "auditrel," use the following command:

```
copy table auditrel () from '/usr/dir/audit.trl'
```

When the copy is finished, "auditrel" will have a row for each operation against the specified table. The values in each row, corresponding to the columns in the table, are:

date	The date and time of the beginning of the multi-query transaction that contained the operation.
username	The ULTRIX username of the user who performed the operation.
operation	One of the following: insert, update, delete.
transaction ID	An 8-byte value composed of two 4-byte integers concatenated together that uniquely identifies the transaction. The column "tranid1" holds the high order 4 bytes and the column "tranid2" holds the low order 4 bytes of the transaction id.
table ID	The identifiers for the table. Table_id1 and table_id2 are two 4-byte integers whose values correspond to the values in the columns "table_reltid" and "table_reltidx," respectively, from the **iitables** standard catalog for the table specified.

Only the Database Administrator who created the database or the ULTRIX/SQL System Administrator (if the **-s** flag is specified) may run the **auditdb** command on a database.

Note that **auditdb** does not necessarily give you a complete list of all transactions since the last checkpoint. There are two reasons for this:

- Since **auditdb** does not exclusively lock the database, other users may complete a transaction while **auditdb** is running.
- In some cases, a completed transaction might not yet have been moved to the journal files.

If you need an absolutely accurate list of transactions since the last checkpoint, make sure all users exit the database before you run **auditdb**.

Some possible diagnostic messages you may receive and their causes are:

Message	Description
You are not a valid ULTRIX/SQL user	The current username is not entered in the ULTRIX/SQL users file.
You may not use the -s flag	You have tried to use the -s flag, but you do not have ULTRIX/SQL System Administrator or superuser privileges.
You are not the dba for *dbname*	You have tried to audit a database for which you are not the Database Administrator.
Cannot enter *dbname*	The database does not exist.

5.3.4 Examples

Audit the "empdata" database.

```
auditdb empdata
```

Audit "empdata," creating an ULTRIX/SQL-readable audit trail for the "employee" table; then copy this into the **sql** monitor environment.

```
%auditdb -temployee -f empdata
%sql empdata

create table empaudit (date date,
    usrname c24, oper c8, tranid1 i4, tranid2 i4,
    tbl_base i4, tbl_index i4, eno i2,
    ename c10, age i1, job i2, salary money, dept i2);
copy table empaudit () from '/usr/directory/audit.trl'
```

5.4 catalogdb

5.4.1 Purpose

List databases that you own.

5.4.2 Syntax

catalogdb [-u*username*]

5.4.3 Description

The **catalogdb** utility allows you to list your databases, the databases that you may access, the *locationnames* known to the system, and the extensions made to your databases. You may also use **catalogdb** to view your user capabilities. For information on how to modify these attributes, see the discussion on **accessdb** in this chapter and in the *ULTRIX/SQL Operations Guide*.

The optional flag for **catalogdb** and its purpose is:

Flag	Description
-u	Allows the System Administrator to use **catalogdb** as the user specified by *username*.

The **catalogdb** utility uses a forms-based interface and must be run on a supported video terminal or in a window emulating a supported terminal. The terminal type is made known to ULTRIX/SQL by way of the environmental variable TERM or TERM_INGRES, which must be set to one of the terminal types defined in the $II_SYSTEM/sql/files/termcap file (see Appendix D, "Defining Your Terminal," for more information).

When you invoke the **catalogdb** command, the main menu appears, offering the following options:

```
Catalog  Database  User  Help  Quit :
```

You can use the **Help** operation to obtain a full description of the main menu items. In summary, these are:

Menu Item	Function
Catalog	Submenu of additional operations (see below)
Database	Detailed information on one database
User	Summary information about your ULTRIX/SQL status
Help	Help information
Quit	Exit the **catalogdb** program

After you select one of these operations, the screen clears and a new display appears. Each menu item evokes a different display. Each display includes its own menu, including a **Help** operation, which provides help information, and an **End** operation, which returns you to the main menu.

You can only browse through the **catalogdb** displays; you cannot change the data. To change any values displayed, you must run the **accessdb** utility, described earlier in this chapter, and in the *ULTRIX/SQL Database Administrator's Guide* and *ULTRIX/SQL Operations Guide.* (You must be the ULTRIX/SQL System Administrator or an ULTRIX/SQL superuser to run the **accessdb** utility.)

The **Catalog** operation calls a submenu offering the following options:

```
Databases  DbExtensions  LocationNames  Help  End
```

In summary, these options perform the following functions:

Menu Item	Function
Databases	Table of all your databases
DbExtensions	Table of all your database extensions
LocationNames	Table of all *locationname*/area mappings on the system
Help	Help information
End	Return to **catalogdb** menu

The **Database** operation on the **catalogdb** main menu prompts you for the name of one of the databases that you own. Enter the full name of the database to invoke the display with information about that database.

The **User** operation on the **catalogdb** main menu displays a summary of information about your username. It lists the permissions accorded to your account, the databases that you own and the private databases to which you have access. The two database lists (the databases you own and others you may access) may contain more entries than can be shown at one time. You can scroll among the entries using the techniques described in Appendix C, "Using Forms-Based Applications."

To leave the display invoked by the **User** operation, type **e** or **end** to select the **End** operation. You will be returned to the main menu for **catalogdb**. You may then select another main menu item.

5.4.4 Examples

Browse through data on your own account and databases.

```
catalogdb
```

As System Administrator, browse the data for another user.

```
catalogdb -uPeter
```

5.5 ckpdb

5.5.1 Purpose

Checkpoint a database.

5.5.2 Syntax

ckpdb [**-d**] [**+j** | **-j**] [**-m***device*] [**-u***username*] [**-s**] [**+w** | **-w**] {*dbname*}

5.5.3 Description

The **ckpdb** command creates a new checkpoint for the named databases and marks all journal entries up to this checkpoint as expired. Because there is a new checkpoint, previous journal entries are no longer needed. Command line flags have the following interpretations:

Flag	Description
-d	Destroy the most recently expired checkpoint and journal files.
+j | **-j**	Enable/disable journaling for a database. When this flag is not specified, the current journaling status of the database is maintained.
-m	Place the new checkpoint onto the specified tape device rather than on disk.
-s	Invoke ULTRIX/SQL superuser (System Administrator) status for system-wide access to any database. You must be the System Administrator.
-u	Execute the **ckpdb** command on specified or all databases owned by the indicated user.
+w | -w	Wait/do not wait for the database to be "free." Note that this flag can be used only in interactive sessions and not in batch mode. The default is **-w**.

Only the Database Administrator who created the database or the ULTRIX/SQL System Administrator (if the **-s** flag is specified) may run the **ckpdb** command on a database. If neither **+j** nor **-j** is specified, the current status of journaling for the database as a whole is maintained.

If you wish, you can write the checkpoint to a specified tape device instead of to disk. Note that you can write only one checkpoint per tape.

If no databases are specified, all databases for which you are the Database Administrator are affected. All databases can have new checkpoints created if the ULTRIX/SQL System Administrator uses the **-s** flag.

The **ckpdb** command locks the database because errors can occur if the database is active while the **ckpdb** command is running. If a database is busy, the **ckpdb** command reports this and proceeds to the next database, if any. If the **-w** flag is specified, the **ckpdb** command does not wait, regardless of standard input. The +w flag ensures that the **ckpdb** command waits.

5.5.4 Examples

Checkpoint “empdata” and initiate journaling on “empdata.”

```
ckpdb +j empdata
```

Checkpoint all databases for which you are DBA, retaining only the newest checkpoints.

```
ckpdb -d
```

Checkpoint “empdata” to tape.

```
ckpdb -m/dev/rmt0 empdata
```

5.6 copydb

5.6.1 Purpose

Creates command files to copy a database and restore it.

5.6.2 Syntax

copydb [**-u***username*] [**-c**] [**-d***pathname*] *dbname* {*tablename*}

5.6.3 Description

The **copydb** command creates two ULTRIX/SQL command files in the current directory.

- The **copy.out** file contains ULTRIX/SQL instructions to copy all tables owned by the user into files in the named directory.
- The **copy.in** file contains ULTRIX/SQL instructions to copy the files into tables, create indexes and perform modifications.

The **copydb** command does not copy the database but creates ULTRIX/SQL commands that do the copying. You must run **sql** using the commands in the **copy.in** and **copy.out** files to copy the database (see the examples).

The name of a file created by **copy.out** consists of the table name, truncated to eight characters if necessary, followed by an extension made up of the first three letters of the owner's login name. If the filename already exists, a unique digit replaces the last character of the table name segment. The directory must *not* be the same as the database's actual directory, nor should it be an ULTRIX/SQL installation directory.

The optional flags have the following purposes:

Flag	Description
-u	Runs **copydb** with the user identification specified by *username*. This flag may only be used by the Database Administrator or an ULTRIX/SQL superuser. The fact that the **copydb** command creates the copy files does not necessarily mean that the user can access the specified table in the **copy.out** script. If table names are specified, only those tables are included in the copy files.
-c	Causes the copy commands in the generated command files to use a portable format. That is, all data is copied in and out as ASCII characters. This is useful for transporting databases between VAX and RISC systems, where internal representations of non-ASCII data differ. (Note that the **copy** command automatically converts data stored in this format back to the appropriate ULTRIX/SQL data type for the corresponding table column.)
-d	Stores the **copy.in** and **copy.out** files in the directory specified by *pathname* instead of the default current directory. The specification may be either a full or relative pathname.

After executing the commands in the **copy.out** file, you create the new database using the **createdb** command. Be sure to execute the commands in the **copy.in** file before doing any work (for instance, creating tables) in the new database.

You should also run **sysmod** after creating the new database in order to reinstitute the optimizing effects of storage structures.

Note that system catalogs cannot be copied using **copydb**. Use **unloaddb** to copy a complete database, including System Catalogs.

5.6.4 Examples

Copy "mydb" to tape.

```
cd /usr/mydir/backup      /*  Or whatever directory you wish */
copydb mydb /usr/mydir/backup
sql mydb < copy.out
tar c .
rm *
```

Copy tape to "mydb."

```
cd /usr/mydir/backup      /* Again, your choice */
tar xrpf /dev/rmt0
sql mydb < copy.in
sysmod mydb
```

5.7 createdb

5.7.1 Purpose

Create a database.

5.7.2 Syntax

createdb [**-u***username*] [**-p**] *dbname* [**-c***locationname*] [**-d***locationname*]
[**-j***locationname*]

5.7.3 Description

The **createdb** command creates a new ULTRIX/SQL database. The person who executes this command becomes the Database Administrator (DBA) for the database. The DBA has special powers not granted to ordinary users.

The variable *dbname* is the name of the database to be created and must be unique among all ULTRIX/SQL database names in your installation. It must begin with an alphabetic character, and it can have a maximum of 24 characters.

The optional flags and their purposes are:

Flag	Description
-u	Allows the System Administrator to create a database as the user specified by *username*.
-p	Restricts access to the database to only the DBA and other users specifically named in the **accessdb** command. (By default, the database is created with access permitted to all ULTRIX/SQL users, although access to any tables in the database must be explicitly granted.)
-c	Stores the checkpoint files at the location specified by *locationname*. The default location is ii_checkpoint.
-d	Stores the database files at the location specified by *locationname*. The default location is ii_database.
-j	Stores the journaling files at the location specified by *locationname*. The default location is ii_journal.

If **createdb** fails for any reason, the partially created database should be destroyed using **destroydb**.

Note that before you can specify any of the *locationnames* mentioned above, the locationnames must be created by the ULTRIX/SQL System Administrator using **accessdb**. The procedures for creating *locationnames* are described in the *ULTRIX/SQL Operations Guide*. If you do not specify one of the flags, the files will be placed on the area corresponding to the default *locationname* for the relevant aspect of the database (that is, checkpoint, database and journal). Databases and their associated journal files should not reside on the same device.

There are two ways to use the **-c**, **-d** and **-j** flags to place database components in directories other than the default. This capability is particularly designed to enable you to locate various database (as well as checkpoints and journals) on different file systems in your installation, and thus on different disks.

One alternative is to name a directory after the flag by the end of its pathname. For example, the following command creates the "newdb" database in the $II_DATABASE/ingres/data/altdir location instead of in the $II_DATABASE/ingres/data/default location.

```
createdb -daltdir newdb
```

Because *altdir* could be mounted as a file system, this technique provides the capability of placing different databases on different disks. Please note that you must create such an alternate directory in 777 mode (that is, the ULTRIX/SQL System Administrator must be the owner and must have read, write and execute permission) before using the directory name in a **createdb** command.

The same is true for checkpoints and journals. The following command creates a database and locates its checkpoints in $II_CHECKPOINT/ingres/ckp/altdir instead of the $II_CHECKPOINT/ingres/ckp/default directory.

```
createdb -caltdir newdb
```

The second way to use the **-c**, **-d** and **-j** flags is to supply a prefix of the directory pathname, beginning with a forward slash (/) character. Consider the following command:

```
createdb -d/other newdb
```

In this case a new database is created in a directory named /other/ingres/data/default as opposed to the default location. The directories at all these levels must already exist prior to executing the particular **createdb** command.

The first part of the pathname (in the previous example, /other) can be whatever you choose, including additional directory levels. Thus, /aa/other would also work. (However, note the limit on the number of characters, specified below.) The lower level directories, starting with "ingres," must have the same names as shown in this example.

The ownership and permissions for the sample directories should be as follows:

```
/other/ingres                 -rwxr-xr-x
/other/ingres/data            -rwx------
/other/ingres/data/default    -rwxrwxrwx
```

Note that whichever alternative you use, the part of the directory name supplied after the **-c**, **-d** or **-j** flags may be no more than 12 characters.

5.7.4 Examples

Create a private database on the default device(s).

```
createdb -p mydb
```

Create public databases under different user names.

```
createdb -ueric ericsdb
```

Create a database with files for the database, checkpoints and journal on different devices.

```
createdb bigdb -ddb_sql -cnewdev_sql -jotherdev_sql
```

The files for the preceding example are:

```
$II_SYSTEM/sql/files/dbtmplt/*

$II_DATABASE/ingres/data/db_sql

$II_CHECKPOINT/ingres/ckp/newdev_sql

$II_JOURNAL/ingres/jnl/otherdev_sql
```

5.8 destroydb

5.8.1 Purpose

Destroy an existing database.

5.8.2 Syntax

destroydb [**-s**] [**-p**] [**-u***username*] *dbname*

5.8.3 Description

The **destroydb** command removes all references to an existing database. The directory of the database and all files in that directory are removed.

To execute this command you must either be the Database Administrator for *dbname*, or you must be the ULTRIX/SQL System Administrator and the **-s** flag must be specified.

The optional flags have the following meanings:

Flag	Description
-s	Indicates that you are the ULTRIX/SQL System Administrator.
-p	Requires ULTRIX/SQL to ask if you are sure that you want to destroy the database.
-u	Allows the System Administrator to use **destroydb** as the user specified by *username*.

5.8.4 Examples

Destroy the "empdata" database, if you are the Database Administrator.

```
destroydb empdata
```

Destroy the "empdata" database, if you are the System Administrator.

```
destroydb -s empdata
```

Allow the System Administrator to impersonate user "brad" when using the **destroydb** command.

```
destroydb -ubrad empdata
```

The files affected by the command are:

```
$II_DATABASE/ingres/data/default/empdata/*
```

5.9 finddbs

5.9.1 Purpose

Recover databases when the ULTRIX/SQL system database (master database) is corrupted or when an entry in a database is missing.

5.9.2 Syntax

finddbs [**-a**|**-r**] [**-p**]

5.9.3 Description

The **finddbs** command is used to recover ULTRIX/SQL when the master database (**iidbdb**) has been corrupted. Only the ULTRIX/SQL System Administrator can use **finddbs**. See the *ULTRIX/SQL Operations Guide* for a complete description of this utility. The flags have the following meanings:

Flag	Description
-a	Run **finddbs** in analyze mode (the default), informing you of possible errors in the database table.
-r	Run **finddbs** in replace mode, rebuilding the **iidbdb** database table by scanning a list of directories for databases.
-p	Cause all databases rebuilt in replace mode to be made private, except for the **iidbdb**. By default, replace mode makes all databases globally accessible.

5.10 isql

5.10.1 Purpose

Initiate the forms-based, interative version of the ULTRIX/SQL Terminal Monitor.

5.10.2 Syntax

isql [**+U** | **-U**] [**-u***username*] [**-c***N*] [**-t***N*] [**-i***kN*] [**-f***kxM.N*] [**-v***X*] [**-n***M*] [**+a** | **-a**] [**-l**]
[**+w** | **-w**] [**-x***k*] *dbname*

5.10.3 Description

This command invokes the **isql** utility, a forms-based interface to interactive ULTRIX/SQL. This interface allows you to enter queries on a special form rather than on the Terminal Monitor command line (as the **sql** command does).

The initial **isql** display is a main menu of operations and a form for entering ULTRIX/SQL queries. You enter a query in the form on the input screen, then select an operation from the main menu. To execute the query you select the **Go** menu item. The results of the query are displayed on an output screen that has another set of menu items. The **Quit** operation exits the utility and returns you to the shell. The **End** operation returns you to the query entry screen.

All menus within **isql** contain a **Help** operation that provides a brief description of each currently available menu item.

The optional flags have the following meanings:

Flag	Description
+U | **-U**	Enable or disable user updating of the system catalog tables and secondary indexes. You must have the "update system tables" privilege obtained through **accessdb.** This option is provided for system debugging and is strongly discouraged for normal use. The default is **-U**. Note that this flag causes an exclusive lock of the database during the session for which it is specified.
-u*username*	Allow you to act as the user with login name *username* (found in the users file). This may only be used by the Database Administrator for the specified database or by the ULTRIX/SQL System Administrator.
-c*N*	Set the minimum field width for printing character columns to *N*. The default is 6.
- t *N*	Set the minimum field width for printing text columns to *N*. The default is 6.
-i*kN*	Set integer output column width to *N*. The integer type *k* may be 1, 2 or 4 for **integer1**, **integer2** or **integer4**, respectively. The default for *N* is 6 for **integer1** and **integer2** fields, and 13 for **integer4** fields.

Flag	Description
-f *kxM.N*	Set floating point output column width to *M* characters (total), including *N* decimal places and (if warranted) *e+-xx* and the decimal indicator character itself. The float type *k* may be 4 or 8, to apply to **float4** or **float8** respectively. The format type *x* may be **E**, **F**, **G** or **N** (uppercase or lowercase) to specify an output format. For a number to be displayed in **E** (that is, exponential) format, either **E** must be specified in the flag or the number must be too large for the format indicated in the flag. **E** is exponential format, **F** is floating-point format and **G** and **N** are identical to **F**, unless the number is too large to fit in that field when it is output in **E** format. **G** format guarantees decimal point alignment; **N** does not. The default display format for both **float4** and **float8** is **n10.3** for VAX and **n11.3** for RISC.
-v*X*	Set the column separator for retrievals to the terminal and print commands to be *X*. The default is a vertical bar (|).
-n*M*	Set modify mode on the index command to *M*, which can be any of the storage structures described in the **modify** command in Chapter 2 (**heap**, **cheap**, **heapsort**, **cheapsort**, **isam**, **cisam**, **btree**, **cbtree**, **hash** and **chash**). The default is **isam**.
+a | -a	Set or clear the autoclear option in the Terminal Monitor. The default is **+a**.
-l	Lock the database for your exclusive use. When you specify this flag, no one else can open the database while you are using it. If you attempt to use this flag on a database that is already opened, the system informs you that the database is temporarily unavailable.
+w | -w	Wait or do not wait for the database. If the +w flag is present, ULTRIX/SQL waits, provided that certain processes (**sql -l**, **sql -U**, **verifydb**, **rollforwarddb** and/or **sysmod**) are running on the given database. Upon completion of those processes, ULTRIX/SQL proceeds. When the **-w** flag is specified, a message is returned and execution is stopped if the database is not available. If the +w|-w flag is omitted and the database is unavailable, an error message is returned if ULTRIX/SQL is running in foreground (more precisely, if the standard input is from a terminal). Otherwise the wait option is invoked. Note that this flag can be used only in interactive sessions and not in batch mode. The default is **-w**.
-x*k*	Set arithmetic handling mode for query processing. The variable *k* may be **f** or w. An **f** indicates that all arithmetic exceptions (floating overflow and underflow, integer overflow and division by zero) should be treated as fatal errors. In warning mode, the detection of an arithmetic exception terminates query processing. A w indicates that warning messages should be generated for arithmetic exceptions. In warning mode, the query is run to completion, and a summary of detected exceptions is generated. The default is to ignore exceptions.

This utility uses a forms-based interface and must be run on a supported video terminal or in a window emulating a supported terminal. The terminal type is made known to ULTRIX/SQL by way of the environmental variable TERM or TERM_INGRES, which must be set to one of the terminal types defined in the $II_SYSTEM/sql/files/termcap file.

The user must be a valid ULTRIX/SQL user.

See Chapter 4 for more information on the **isql** command.

5.10.4 Example

Invoke **isql** on the "employee" database.

```
isql employee
```

5.11 optimizedb

5.11.1 Purpose

Generate statistics for use by the optimizer.

5.11.2 Syntax

optimizedb [**-zf***filename*] [**-zv**] [**-zh**] [**-zk**] [**-zx**] [**-zu**#] [**-zr**#] [**-i** *filename*]
[**-zp**] [**-zs**[s]#] [**-zc**] [**-zf**#] [*ULTRIX/SQL flags*]
dbname [{**-r***tablename* {**-a***columnname*}}]

5.11.3 Description

The **optimizedb** command retrieves values from the specified tables and columns. These values are used to generate statistics, which are stored in system catalogs and can be viewed in the Standard Catalogs **iistats** and **iihistograms**. These statistics are used by the query optimizer to select an efficient query processing strategy. Such statistics should be generated for all columns that may appear in the qualification of a query statement. Statistics for columns named in the target list of a query or a query's sort list are not used. After running **optimizedb**, you should run **sysmod**. This is especially true the first time **optimizedb** is run on a database.

More complete and accurate statistics in the system catalogs generally result in more efficient query execution strategies, and hence faster system performance. The process of generating such complete and accurate statistics may require some time, but a compromise between accurate statistics and the time to generate them can be achieved by specifying the **-zx** or **-zs** flag, described below. Another compromise relies on how often you regenerate the statistics. The statistics need only infrequent regeneration, usually when a significant change has occurred in the distribution of a column's values.

There need be no statistics for any columns whatsoever, and any statistics may be incorrect. The only effect is on the speed of query processing, not whether the query will execute or not.

The statistics generated by the **optimizedb** command for any column consist of two basic elements:

- The number of unique values in a column
- A histogram with a variable number of variable-width cells

The accuracy of the histograms can be controlled by the **-zu**# and **-zr**# flags described below. Increasing the number of cells in the histograms increases the amount of space required for the **iihistograms** table and thus increases somewhat the amount of space and time used by the query optimizer. However, the increased accuracy of the statistics will generally result in more efficient query execution strategies.

Note

While **optimizedb** is running, ULTRIX/SQL does not lock either the database or individual tables.

The **optimizedb** command line flags have the following functions:

Flag	Description
-zf*filename*	Directs **optimizedb** to read *filename* for all other command line flags, database names, and any other command line arguments. This file must contain only *one flag per line* (see the examples below). If this flag is specified, no other flags or arguments can appear on the command line; they must, instead, appear in the specified file.
-zv	Displays information about each column as it is being processed.
-zh	Displays the histogram that was generated for each column. This flag also implies the **-zv** flag.
-zk	Generates statistics for columns that are keys on the table or are indexed, in addition to columns specified on the command line.
-zx	Directs **optimizedb** to determine only the minimum and maximum values for each column rather than full statistics. Because minimum and maximum values for columns from the same table can be determined by a single scan through the table, this flag provides a quick way to generate a minimal set of statistics. Minimal statistics cannot be created on columns holding only null values.
-zu#	Specifies the maximum number of cells an exact histogram can contain. In an exact histogram, each cell represents a single, unique value. The default is 100. If the number of unique values exceeds either the default or the number specified with this flag, **optimizedb** creates an inexact histogram. (You can use the **-zr** flag to control the number of cells in an inexact histogram.) Note that **optimizedb** always attempts to create an exact histogram first, but if this is not possible, will then create an inexact histogram.
-zr#	Specifies the maximum number of cells that the histogram can contain if **optimizedb** creates an inexact histogram. In an inexact histogram, each cell represents a range of values. The default number of cells is 15.
ULTRIX/SQL flags	Automatically passes ULTRIX/SQL flags on the **optimizedb** command line to ULTRIX/SQL. Consult the **sql** command summary in this chapter for a complete description of all ULTRIX/SQL flags.
-i*filename*	Directs **optimizedb** to read statistics from *filename* instead of operating directly on the database. The *filename* is the name of a file that has been generated by the **statdump** command using the **-o** flag. This file is in ASCII format and can be edited. However, only two types of changes are acceptable: a) you can modify values, and b) you can add rows describing cells. Do not change the format of the file—that is, do not change the order in which data appears or add an incomplete new row. When the **-r** and **-a** flags are used in conjunction with **-i**, they act as filters. **Optimizedb** will only read in from the file those statistics that belong to the specified table or column.

Flag	Description
	Optimizedb does not use the row and page count values in the file unless the **-zp** flag is also specified. Note that these values are vital for correct operation of the database management system. Be very careful if you use the **-zp** flag to put new values for row and page counts in **iitables.**
-zp	Directs **optimizedb** to read the row and page count values in the file specified with the **-i** flag and to store those values in the appropriate System Catalog. (The values can be viewed in **iitables.**)
-zs[s]#	Creates statistics based on sample data. The percentage of table rows sampled is determined by the value of #. This number must be a floating point number in the range of 0 to 100. The optional s directs ULTRIX/SQL to sort the tuple identifiers (TIDs), which are used to retrieve the sample rows, before the rows are retrieved. This decreases retrieval time but increases the amount of memory used by **optimizedb.**
-zc	Directs **optimizedb** to optimize the system catalogs in addition to the base tables. If you want to optimize selected system catalogs, rather than all of them, use this flag and specify the individual tables with the **-r** flag. The **-zc** flag is operational only if the user issuing the command is the Database Administrator for the specified database.
-zf#	Directs **optimizedb** to read floating point numbers using the precision level specified by #. Use this flag in conjunction with the **-i** *filename* flag.
-r*tablename*	Directs **optimizedb** to generate statistics only for the specified table. All columns for all tables in the database are processed. Otherwise only columns for the specified table are processed. You can include this flag more than once on the command line to generate statistics for two or more specified tables in a single **optimizedb** command.
-a*columnname*	If the **-r***tablename* flag is specified, then you can specify individual columns for the generation of statistics. When tables and columns are specified, statistics processing occurs only for the specified columns unless the **-zk** flag is included. (See the **-zk** flag description for details.)

Some possible diagnostic messages you may receive using **optimizedb** and their causes are:

Message	Description
More than 1000 arguments	There are too many lines in the argument file specified with the **-zf** flag.
Bad unique cells value	The value specified in the **-zu#** flag was not a number, or was less than 1 or greater than 249.
Bad regcells value	The value specified in the **-zr#** flag was not a number, was less than 2 or greater than 499.

5.11.4 Examples

Generate full statistics for all columns in all tables in the "empdata" database.

```
optimizedb empdata
```

Generate statistics for key or indexed columns in the "employee" and "dept" tables, and additionally generate statistics for the "dno" column in the "dept" table.

```
optimizedb -zk empdata -remployee -rdept -adno
```

Do the same as the second example, but from a file.

```
optimizedb -zf flagfile
```

The parameter *flagfile* in the preceding example contains:

```
    -zk
    empdata
    -remployee
    -rdept
    -adno
```

Generate statistics for all key or indexed columns in "employee," "dept" and "salhist." Also process the "eno" column in "employee," whether or not "eno" is a key or indexed column. Generate statistics with only minimum and maximum values from the columns. Print status information as each column is processed.

```
optimizedb -zk -zv -zx empdata -remployee -aeno -rdept
    -rsalhist
```

Allow up to 100 unique values from each column in the "employee" table before merging adjacent values into the same histogram cell.

```
optimizedb -zu100 empdata -remployee
```

5.12 rollforwarddb

5.12.1 Purpose

Recover the database from the last checkpoint and the current journal.

5.12.2 Syntax

rollforwarddb [+**c**|-**c**] [+**j**|-**j**] [-**m***device:*] [-**s**] [-**u***username*]
[-**v**] [+**w**|-**w**] {*dbname*}

5.12.3 Description

The **rollforwarddb** command recovers the named databases from the last checkpoint and the current journal. The recommended procedure is to recover the last checkpoint, then recover from the journal (see "Examples" below).

The command line flags have the following interpretations:

Flag	Description
+c | **-c**	Recover or do not recover the database from the last checkpoint. The default is **+c**.
+j | **-j**	Recover or do not recover the database from the journal. The default is **+j**.
-m*device*	Recover the checkpoint from the specified tape device rather than from disk.
-s	Invoke ULTRIX/SQL superuser (System Administrator) status for system-wide access to any database. You must be the ULTRIX/SQL System Administrator.
-u*username*	Allows you to act as the user with login name *username*. This may only be used by the Database Administrator for the specified database or by the ULTRIX/SQL System Administrator.
-v	Recover the database from the journal in verbose mode, which provides diagnostic information detailing all operations executed during the recovery process.
+w | **-w**	Wait or do not wait for the database to be "free." The default is **-w**.

If you have written to tape the checkpoint from which you want to restore the journal, you can use the **-m** flag to read in the checkpoint from a tape device.

Only the Database Administrator who created the database or the ULTRIX/SQL System Administrator (if the **-u** flag is specified) may run the **rollforwarddb** command on a database.

If no databases are specified, all databases for which you are the DBA are affected. All databases can be purged if the ULTRIX/SQL System Administrator uses the **-s** flag.

The **rollforwarddb** command locks the database because errors can occur if the database is active while the **rollforwarddb** command is running. If a database is busy, the **rollforwarddb** command reports this and proceeds to the next database, if any. By default, or if the **-w** flag is specified, the **rollforwarddb** command does not wait, regardless of standard input. The **+w** flag always causes the **rollforwarddb** command to wait.

Some possible diagnostic messages you may receive and their causes are:

Message	Description
You are not a valid ULTRIX/SQL user	The current login name is not entered in the ULTRIX/SQL users file.
You may not use the -s flag	You have tried to use the -s flag, but you do not have ULTRIX/SQL System Administrator privileges.
You are not the dba for *dbname*	You have tried to recover a database for which you are not the Database Administrator.
Cannot enter *dbname*	The specified database does not exist.

5.12.4 Examples

Recover the "empdata" database in verbose mode from the last checkpoint and journal. This assumes that both the journal and the checkpoint are currently on-line. If not, they should be placed on-line before executing these commands.

```
rollforwarddb -v empdata
```

Recover all databases in verbose mode for which you are the Database Administrator.

```
rollforwarddb -v
```

Recover "empdata" from tape, and then apply the journals.

```
rollforwarddb +c +j -m/dev/rmt0 empdata
```

5.13 sql

5.13.1 Purpose

Invoke the Terminal Monitor command line interface to interactive ULTRIX/SQL.

5.13.2 Syntax

sql [**+U** | **-U**] [**-u***username*] [**-c***N*] [**-t***N*] [**-i***kN*] [**-f***kxM.N*] [**-v***X*] [**-n***M*] [**+a** | **-a**] [**-l**] [**+d** | **-d**] [**+s** | **-s**] [**+w** | **-w**] [**-x***k*] [<*altin*] [>*altout*] *dbname*

5.13.3 Description

This command invokes the ULTRIX/SQL Terminal Monitor command line interface (**sql**). The variable *dbname* is the name of an existing database. The optional flags have the following meanings:

<table>
<tr><th>Flag</th><th>Description</th></tr>
<tr><td>+U | -U</td><td>Enable or disable user updating of the system catalog tables and secondary indexes. You must have the "update system tables" privilege obtained through accessdb. This option is provided for system debugging and is strongly discouraged for normal use. The default is -U. Note that this flag causes an exclusive lock of the database during the session for which it is specified.</td></tr>
<tr><td>-uusername</td><td>Allows you to act as the user with login name username (found in the users file). This flag can only be used by the Database Administrator for the specified database or by the ULTRIX/SQL System Administrator.</td></tr>
<tr><td>-cN</td><td>Set the minimum field width for printing character columns to N. The default is 6.</td></tr>
<tr><td>-tN</td><td>Set the minimum field width for printing text columns to N. The default is 6.</td></tr>
<tr><td>-ikN</td><td>Set integer output column width to N. The integer type k may be 1, 2 or 4 for integer1, integer2 or integer4, respectively. The default for N is 6 for integer1 and integer2 fields, and 13 for integer4 fields.</td></tr>
<tr><td>-fkxM.N</td><td>Set floating point output column width to M characters (total), including N decimal places and (if warranted) e+-xx and the decimal indicator character itself. The float type k may be 4 or 8, to apply to float4 or float8 respectively. The format type x may be E, F, G or N (uppercase or lowercase) to specify an output format. For a number to be displayed in E (that is, exponential) format, either E must be specified in the flag or the number must be too large for the format indicated in the flag. E is exponential format, F is floating-point format and G and N are identical to F, unless the number is too large to fit in that field when it is output in E format. G format guarantees decimal point alignment; N does not. The default display format for both float4 and float8 is n10.3 for VAX or n11.3 for RISC.</td></tr>
<tr><td>-vX</td><td>Set the column separator for retrievals to the terminal and print commands to be X. The default is a vertical bar (|).</td></tr>
</table>

<table>
<tr><th>Flag</th><th>Description</th></tr>
<tr><td>-nM</td><td>Set modify mode on the index command to M, which can be any of the storage structures described in the modify command in Chapter 2 (heap, cheap, heapsort, cheapsort, isam, cisam, btree, cbtree, hash and chash). The default is isam.</td></tr>
<tr><td>+a | -a</td><td>Set or clear the autoclear option in the Terminal Monitor. The default is +a.</td></tr>
<tr><td>-l</td><td>Lock the database for your exclusive use. When you specify this flag, no one else can open the database while you are using it. If you attempt to use this flag on a database that is already opened, the system informs you that the database is temporarily unavailable.</td></tr>
<tr><td>+d | -d</td><td>Print or do not print the dayfile. The default is +d.</td></tr>
<tr><td>+s | -s</td><td>Print or do not print any of the monitor messages, including prompts. This flag is normally set. If cleared, it also clears the -d flag. The default is +s.</td></tr>
<tr><td>+w | -w</td><td>Wait or do not wait for the database. If the +w flag is specified, ULTRIX/SQL waits, provided that certain processes (sql -l, sql -U, verifydb, rollforwarddb and/or sysmod) are running on the given database. Upon completion of those processes, ULTRIX/SQL proceeds. When the -w flag is present, a message is returned and execution is stopped if the database is not available. If the +w|-w flag is omitted and the database is unavailable, an error message is returned if ULTRIX/SQL is running in foreground (more precisely, if the standard input is from a terminal). Otherwise the wait option is invoked. Note that this flag can be used only in interactive sessions and not in batch mode. The default is -w.</td></tr>
<tr><td>-xk</td><td>Set arithmetic handling mode. The variable k may be f or w. An f indicates that all arithmetic exceptions (floating overflow and underflow, integer overflow and division by zero) should be treated as fatal errors. In this mode, the detection of an arithmetic exception terminates query processing. A w indicates that warning messages should be generated for arithmetic exceptions. In warning mode, the query is run to completion, and a summary of detected exceptions is generated. The default is to ignore exceptions.</td></tr>
</table>

Optional Arguments	Description
<*altin*	Use an alternate file to input Terminal Monitor commands to ULTRIX/SQL. The file *altin* should contain all the Terminal Monitor commands needed to run an ULTRIX/SQL session. This can be used to run ULTRIX/SQL interactive procedures, such as processing the output of the **copydb** command.
>*altout*	Use an alternate file for all output from the Terminal Monitor. This option can capture the output of a terminal session for later reference. Note that you do not see any output from ULTRIX/SQL if you use this option.

Some possible diagnostic messages you may receive and their causes are:

Message	Description
Bad flag format	You have specified a flag in an unintelligible format, or an invalid flag.
Database *name* does not exist	The specified database does not exist.

Message	Description
Database temporarily unavailable	Someone else is currently performing some operation on the database; you cannot start ULTRIX/SQL now.
Error starting up ULTRIX/SQL Request for lock failed	The database you tried to access is currently exclusively reserved for another user.
Improper database name	The database name is not legal.
No database name specified	You did not specify the database name.
Too many options to ULTRIX/SQL	You have included too many flags on the **sql** command line.
Too many parameters	You have specified a database name and something else that ULTRIX/SQL cannot decipher.
You are not a valid ULTRIX/SQL user	You are not entered into the user's file; you may not use ULTRIX/SQL at all.
You are not authorized to use the flag	The specified flag requires some special authorization, which you do not have.
You may not access database *name*	You do not have access permission for this database.

5.13.4 Examples

Open the "empdata" database.

```
sql empdata
```

Open "empdata," suppressing the dayfile message.

```
sql -d empdata
```

Open "empdata," suppressing the dayfile message and the Terminal Monitor prompts and messages; read into the workspace the contents of the batchfile file.

```
sql -s empdata < batchfile
```

Open "empdata," display f4 columns in G format with two decimal places and i1 columns with three spaces.

```
sql -f4g12.2 -i13 empdata
```

The files that are affected are:

```
$II_SYSTEM/sql/files/users
$II_DATABASE/ingres/data/default/dbname/*
```

5.14 statdump

5.14.1 Purpose

Print statistics contained in the **iistats** and **iihistograms** catalogs of the Standard Catalog Interface.

5.14.2 Syntax

statdump [**-zq**] [**-zdl**] [**-o***filename*] [**-zc**] [**-zf#**]
    [*ULTRIX/SQL flags*] *dbname* [{**-r***tablename* {**-a***columnname*}}]

5.14.3 Description

The **statdump** command allows you to inspect the **iistats** and **iihistograms** catalogs in the Standard Catalog Interface. These views contain statistical information about columns used by the query optimizer as it selects an efficient query processing strategy. The statistical information is usually generated by issuing the **optimizedb** command.

The command line flags have the following meanings:

Flag	Description
-zq	Prints only the information contained in the **iistats** catalog and not the histogram information contained in **iihistograms**. (Quiet mode.)
-zdl	Deletes statistics from the System Catalogs. When this flag is included, the statistics for the specified tables and columns (if any are specified) are deleted rather than displayed.
-o*filename*	Directs the output of **statdump** to the file specified by *filename*. The resulting file is an ASCII file whose content is identical to the information normally sent to the terminal screen. It is possible to edit the contents of this file; however, only two types of changes are acceptable if this file will be used as input for the **optimizedb** command. (See the **-i** flag description in the **optimizedb** command description for a discussion of using *filename* with the **optimizedb** command.)
-zc	Directs **statdump** to display statistics on the system catalogs as well as the base tables. If you want statistics for selected system catalogs, use this flag and specify the individual tables with the **-r** flag. You must be the Database Administrator of the specified database to use this flag.
-zf#	Directs **statdump** to output floating point values in scientific notation (for example, 9.9999+e9) and sets the precision to the level specified by #. The total width of the displayed number will be equal to the value of the precision level + 7.
[*ULTRIX/SQL flags*]	Automatically passes any ULTRIX/SQL flags on the **statdump** command line to ULTRIX/SQL. For information about the ULTRIX/SQL flags, refer to the **sql** command description in this chapter.
-r*tablename*	Produce statistics for all columns in the specified table. If no table is specified, statistics for all columns in all tables are produced.

Flag	Description
-a*columnname*	Produce statistics for the specified columns only. Note that to specify individual columns you must first specify a table name with the **-r** flag, as the syntax summary indicates.

Note

If a table or column cannot be found, a warning message is printed and processing of other statements in the query continues.

5.14.4 Examples

Print the statistical information for all columns in the "employee" table in the "empdata" database.

```
statdump empdata -remployee
```

For all columns in all tables of the "empdata" database, print only the information in the **iistats** system table.

```
statdump -zq empdata
```

Delete statistics for all columns in the "employee" table.

```
statdump -zdl empdata -remployee
```

5.15 sysmod

5.15.1 Purpose

Modify system tables to predetermined storage structures.

5.15.2 Syntax

sysmod [-s] [+w | -w] *dbname* [*tablename* { , *tablename*}]

5.15.3 Description

The **sysmod** command modifies a database's system tables to the most appropriate storage structure, usually **hash**, for accelerating query processing. You can run **sysmod** on the whole database or on specified tables. The user must be either the Database Administrator for the specified database or the ULTRIX/SQL System Administrator, in which case the **-s** flag must be specified.

The flags have the following meanings:

Flag	Description
-s	Allows the ULTRIX/SQL System Administrator to use **sysmod** on another user's database.
+w | **-w**	Causes ULTRIX/SQL to wait or not wait until the database is free before executing **sysmod**. This can only be used in interactive sessions, not in batch mode. The **sysmod** command locks the database while it modifies the system tables, in order to prevent errors. If the database is in use, **sysmod** reports that the database is not free, and **sysmod** does not execute. If standard input is not a terminal, **sysmod** waits for the database to be free. By default, or if the **-w** flag is specified, **sysmod** does not wait, regardless of standard input. The **+w** flag causes **sysmod** to wait until the database is no longer in use, regardless of standard input.

The **sysmod** command should be run on a database periodically to maintain peak performance. Whenever many tables and secondary indexes are created and/or destroyed, **sysmod** should be run even more often.

5.15.4 Examples

Optimize the system tables in "empdata."

```
sysmod empdata
```

Optimize the **iirelation** and **iiindexes** system tables in "empdata," but only if the database is not currently busy.

```
sysmod -w empdata iirelation iiindexes
```

5.16 unloaddb

5.16.1 Purpose

Create command files for complete unloading and reloading of a database.

5.16.2 Syntax

unloaddb [**-u***username*] [**-c**] [**-d***pathname*] *dbname*

5.16.3 Description

The **unloaddb** command creates a set of command files that can be run by the DBA for an ULTRIX/SQL database to unload completely all tables in the specified database. The **unloaddb** utility works in the same way as the **copydb** command except that the **unloaddb** command files also unload all views, integrity constraints and permissions in the database. Also, unlike the **copydb** command, **unloaddb** command files unload all user-defined tables, views and so forth in the database of which you are the Database Administrator, not merely those items that you own. This utility can be used when a database must be totally rebuilt or for checkpointing the database.

The **unloaddb** utility creates two command files in the current directory that can then be executed by the Database Administrator:

- The **unload.ing file contains commands to read sequentially through the database, copying every user table into its own file in the named directory.**
- The **reload.ing** file contains commands to reload the database with the information contained in the files created by the unload.ing command file.

Note that the **unloaddb** command does not actually do the unloading or reloading of the database. The command files created by **unloaddb** must be executed by the Database Administrator to accomplish these tasks. The directory specified in the **unloaddb** command must not be the actual database directory, $II_DATABASE/ingres/data/default/dbname, because the files created by unloaddb may have the same names as the tables in the database.

The optional flags and their purposes are:

Flag	Description
-u	Allows you to run **unloaddb** as the user specified by *username*. This flag can only be used by the ULTRIX/SQL System Administrator.
-c	Causes the commands in the generated command files to use a portable format. That is, all data is copied in and out as ASCII characters. This is useful for transporting databases between computer systems whose internal representations of non-ASCII data differ.
-d	Stores the **unload.ing** and **reload.ing** files in the location specified by *pathname* instead of the default current directory. The pathname can be either a full or relative directory specification.

The new database is created with the **createdb** command. It is important that the Database Administrator reload the new database using the **reload.ing** file before any work (for instance, creating tables) is done in the database. You should also be sure to run the **sysmod** command in order to optimize performance after recreating and reloading the database.

The **unloaddb** command uses a version of the **copydb** utility to generate the copy commands in the **unload.ing** and **reload.ing** files. Thus all limitations of the **copydb** command apply to the **unloaddb** command.

5.16.4 Example

Unload and reload the "empdata" database.

```
cd /mydir/backup
unloaddb empdata
unload.ing
destroydb empdata
createdb empdata
reload.ing sysmod empdata
```

Key Words A

A.1 ULTRIX/SQL

The following identifiers are key words in ULTRIX/SQL and are therefore reserved:

abort	**count**	**endwhile**	**is**	**privileges**	**then**
all	**create**	**execute**	**like**	**procedure**	**to**
and	**current**	**exists**	**max**	**public**	**union**
any	**cursor**	**for**	**message**	**relocate**	**unique**
as	**declare**	**from**	**min**	**return**	**until**
asc	**delete**	**grant**	**modify**	**revoke**	**update**
alter	**desc**	**group**	**not**	**rollback**	**user**
at	**describe**	**having**	**null**	**save**	**using**
avg	**distinct**	**if**	**of**	**savepoint**	**values**
between	**do**	**immediate**	**on**	**select**	**where**
by	**drop**	**in**	**open**	**set**	**while**
check	**else**	**index**	**or**	**some**	**with**
close	**elseif**	**insert**	**order**	**sql**	**work**
commit	**endif**	**integrity**	**permit**	**sum**	
copy	**endloop**	**into**	**prepare**	**table**	

A.2 Embedded ULTRIX/SQL

The following list contains the key words specific to embedded ULTRIX/SQL. Note that all the ULTRIX/SQL key words listed above are also reserved in embedded ULTRIX/SQL.

activate	**deleterow**	**getform**	**insertrow**	**repeated**
addform	**disconnect**	**getoper**	**loadtable**	**resume**
breakdisplay	**display**	**getrow**	**menuitem**	**screen**
call	**down**	**go**	**message**	**scroll**
clear	**enddata**	**goto**	**next**	**scrolldown**
clearrow	**enddisplay**	**help**	**notrim**	**scrollup**
close	**endforms**	**helpfile**	**open**	**sleep**
column	**endloop**	**identified**	**out**	**stop**
command	**endselect**	**include**	**print**	**submenu**
connect	**fetch**	**indicator**	**prompt**	**tabledata**
continue	**field**	**initialize**	**putform**	**unloadtable**
current	**finalize**	**inittable**	**putrow**	**up**
cursor	**formdata**	**inquire_frs**	**redisplay**	**validate**
declare	**forminit**	**inquire_ingres**	**register**	**validrow**
descriptor	**forms**	**inquire_sql**	**remove**	**whenever**

Double reserved words. The following words are reserved when they appear together on the same line with only spaces separating them.

begin declare	**direct disconnect**
begin transaction	**direct execute**
create link	**drop link**
direct connect	**end transaction**

A.3 ANSI SQL

The list below comprises the proposed ANSI standard key words that are not currently reserved in ULTRIX/SQL or embedded ULTRIX/SQL. You may wish to treat these as reserved words to ensure compatibility with other implementations of SQL.

authorization	**double**	**module**	**public**
char	**float**	**numeric**	**real**
character	**fortran**	**option**	**schema**
cobol	**found**	**pascal**	**smallint**
constraints	**int**	**pli**	**sqlcode**
dec	**integer**	**precision**	**sqlerror**
decimal	**language**	**procedure**	

A.4 Host Language Key Words

You cannot use host language key words, including language-defined data types, as objects in embedded ULTRIX/SQL statements.

Standards Compliance and Compatibility Information B

B.1 Conventions

The following tables compare ULTRIX/SQL to ANSI/ISO, X/Open, and VAX Rdb/VMS SQL. Entries in the columns listed below have the following meanings:

- ULTRIX/SQL column

  Entries are key words, statements or statement options that are included in the ULTRIX/SQL implementation of the SQL language (SQL statements that apply only to utility environments, such as the ULTRIX/SQL **help** statement, have been omitted).

- ANSI/ISO column

  Indicates whether the ULTRIX/SQL language component complies with the ANSI/ISO SQL-89 standard, published as the ANSI X3.135-1989 Database Language SQL standard and as the ISO 9075:1989 Database Language SQL standard.

- X/Open column

  Indicates whether the ULTRIX/SQL language component is included in the X/Open XPG3 standard. An asterisk (*) in this column indicates that the component is not currently included, but planned for XPG4.

- Rdb/VMS SQL column

  Indicates whether the ULTRIX/SQL language component in included in the Rdb/VMS implementation of the SQL language, or if included, whether the ULTRIX/SQL component is compatible with the Rdb/VMS component.

B.2 Data Types

ULTRIX/SQL	ANSI/ISO	X/Open	Rdb/VMS SQL
c1 - c2000	No	No	No
char[(*n*)] *n* <= 2000	Yes, except *n* must be specified	Yes, except *n* <= 240	Yes, except *n* <= 16383
character[(*n*)] *n* <= 2000	Yes, except *n* must be specified	No	Yes, except *n* <= 16383
text(*n*)	No	No	No
varchar(*n*) *n* <= 2000	Yes, except *n* is not specified	Yes, except *n* is not specified	Yes
integer1	No	No	No
integer2	No	No	No
smallint (2-byte)	Yes	Yes	Yes
int (4-byte)	Yes	No	Yes
integer4	No	No	No
integer (4-byte)	Yes	Yes	Yes
real	Yes	No	Yes
float4	Yes	No	Equivalent to **real** and **float**(*x*), where *x* <=7
double precision	Yes	No	Yes
float8	Yes	Yes	Equivalent to **double precision** and **float**(*x*), where *x* > 7
float	Yes	Yes	Equivalent to **double precision** and **float**(*x*), where *x* > 7
date	No	No	Format differs
money	No	No	No

B.3 Statements

ULTRIX/SQL Command	ANSI/ISO	X/Open	Rdb/VMS SQL
commit	No	No	Yes
commit work	Yes	Yes	Yes
copy	No	No	No
create index	No	Yes	Yes
asc/desc clauses	No	Yes	Yes
unique clause	No	Yes	Yes
with			
fillfactor = *n*	No	No	No
key = (*columnlist*)	No	No	Implementation differs
leaffill = *n*	No	No	No
location = (*locationname* ...)	No	No	Syntax differs
maxpages = *n*	No	No	No
minpages = *n*	No	No	No
nonleaffill = *n*	No	No	No
structure = **btree** | **cbtree** | **hash** | **chash** | **isam** | **cisam**	No	No	Syntax and structure types differ
create integrity	Implementation differs	No	Implementation differs
create procedure	No	No	No
create table	Yes	Yes	Yes
as *subselect*	No	No	No
with [no] duplicates	No	No	No
with [no] journaling	No	No	No
with location = (*locationname* ...)	No	No	No
create view	Yes	Yes	Yes
as *subselect*	Yes	Yes	Yes
with check option	Yes	No	Syntax differs
declare *variable*	No	No	No

ULTRIX/SQL Command	ANSI/ISO	X/Open	Rdb/VMS SQL
not null [with default | not default] | with null	No	No	No
delete	Yes	Yes	Yes
where *search_condition*	Yes	Yes	Yes
drop *indexname* | *tablename* | *viewname*	No	No	No
drop index	No	Yes	Yes
drop integrity	No	No	Some forms of; implementation differs
drop permit	No	No	Some forms of; implementation differs
drop procedure	No	No	No
drop table	No	Yes	Yes
drop view	No	Yes	Yes
grant	Yes	Yes	Syntax differs
all	No	Yes	Yes
all privileges	Yes	No	Yes
on procedure	No	No	No
privileges where *privileges* are:			
delete	Yes	Yes	Yes
execute	No	No	No
insert	Yes	Yes	Yes
select	Yes	Yes	Yes
update (*columnname* {, *columnname*})	Yes	Yes	Yes
to public	Yes	Yes	Yes
to *username*	Yes	Yes	Syntax differs
if-then-else	No	No	No
insert	Yes	Yes	Yes
message	No	No	No

ULTRIX/SQL Command	ANSI/ISO	X/Open	Rdb/VMS SQL
modify	No	No	Some forms of; implementation differs
return	No	No	No
rollback	No	No	Yes
rollback work	Yes	Yes	Yes
save	No	No	No
select (interactive)	Yes	Yes	Yes
all	Yes	Yes	Yes
distinct	Yes	Yes	Yes
from clause	Yes	Yes	Yes
group by clause	Yes	Yes	Yes
having clause	Yes	Yes	Yes
order by clause	Yes	Yes	Yes
asc | desc	Yes	Yes	Yes
union	Yes	Yes	Yes
union all	Yes	No	Yes
where clause	Yes	Yes	Yes
set	No	No	Some forms of; implementation differs
update	Yes	Yes	Yes
where clause	Yes	Yes	Yes
while-endwhile	No	No	No

Using Forms-Based Applications C

C.1 Overview

This appendix explains how to use various features of the ULTRIX/SQL forms-based utilities: **accessdb**, **catalogdb**, and **isql**.

C.2 Accessing Databases

Using ULTRIX/SQL you can access:

- ULTRIX/SQL databases on your own computer or local node
- ULTRIX/SQL databases on remote nodes on your network
- Rdb/VMS databases through Remote Access to Rdb/VMS

The general syntax for accessing a database is:

command [*v_node*::]*dbname*[/*server_type*]

Table C-1 Access Syntax

Parameter	Function
command	Any command used to invoke an ULTRIX/SQL software tool, such as **sql** or **isql**.
v_node::	The virtual node name of the remote node on which the database is located (note the two colons). The *v_node* name implies the actual network node address and the network protocol. They are identified when the node name is defined with the **netu** utility.
dbname	The name of the database containing the relevant data.
server_type	The name of the type of server being accessed at the local or remote site, (the default is **ingres**, which is the server_type designated for the database management system server). See the following table for other server types.

Table C-2 ULTRIX/SQL Server Types

Server_Type	ULTRIX/SQL Server Description
ingres	ULTRIX/SQL database management system
rdb	ULTRIX/SQL Remote Access to RDB/VMS

Not all the parameters have to be specified in all cases. For example, when accessing a database on your own computer, your local node, you do not need to specify the *v_node* name.

C.3 Menus

An ULTRIX/SQL menu is displayed at the bottom of each ULTRIX/SQL frame. You can cycle through the display of all of a menu's selections by pressing the **Menu** key. (See the next section.)

On any given frame, ULTRIX/SQL displays the cursor in the window or at the end of the menu. To choose an operation from the menu:

- Press the function key mapped to the operation.
- Move the cursor to the menu with the **Menu** key and type the name of the operation. See the section "Selecting an Operation from the Menu" for details.

C.3.1 Menu Key

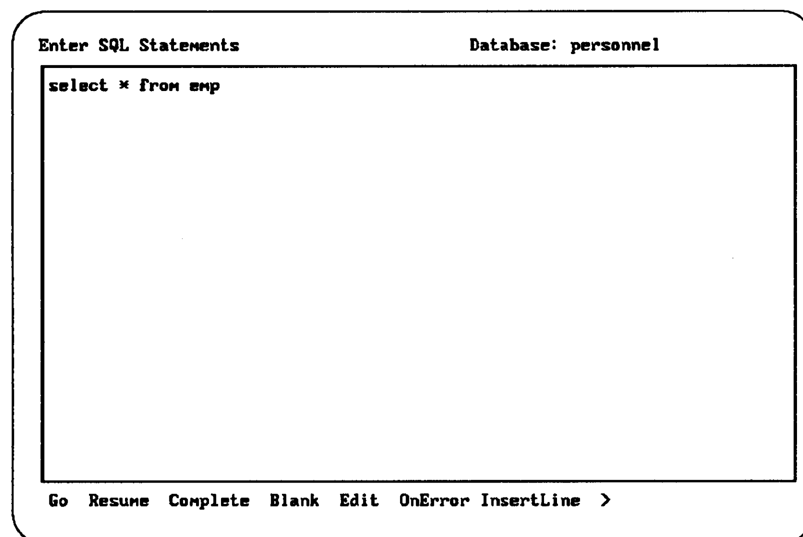
To move the cursor from the window to the menu, press the **Menu** key. To return to the window, press the **Return** key.

The key on your keyboard that acts as the **Menu** key depends on your terminal and the individual key mappings you have chosen. For example, on a VT100 keyboard, the **Menu** key is normally mapped to the **PF1** key. In some instances the **Menu** key may be mapped to the **Escape** key or the **F1** key. See Appendixes D through F for further information on defining and customizing your terminal keyboard.

C.3.2 Long Menus

Some menus contain more items than will fit across the bottom of the frame. The presence of additional menu items is indicated by either a "greater than" character (>) at the right end of the menu, a "less than" character (<) at the left end, or both.

Figure C-1 Long Menu



In this case, you can cycle through the entire set of menu options by pressing the **Menu** key repeatedly.

The example frames in this manual normally show the full set of operations available on the menu. The key mappings specific to different terminals indicated in parentheses after each operation are normally omitted in the manual, although they are displayed on your terminal.

C.4 Selecting an Operation from the Menu

In ULTRIX/SQL, you can select an operation from the menu by either function key or operation name.

C.4.1 Selection by Function Key

If the operation has a function key (or key combination) mapped to it, you can simply press that key. This invokes the operation regardless of where the cursor is when you press the key. The function key that is mapped to an operation is shown in parentheses after the operation. For example, **Quit(PF4)** means that pressing the **PF4** key will invoke the **Quit** operation. If **^F** is shown in parentheses, you hold down the **Control** key and type **f**.

The operation you select begins immediately. You do not have to press **Return**. See Appendix E for information about defining and customizing your terminal keyboard.

If you want to see all the available operations or if you need to use the menu, press the **Menu** key to put the cursor on the menu line. You can then select an operation by using the name of the operation or by using the number that appears in parentheses after the operation.

C.4.2 Selection by Name

If your terminal does not have the correct mapping of function keys or if for some other reason the function keys do not work, to select an operation from the menu:

1. Move the cursor to the menu by pressing the **Menu** key (see above).
2. Type as many letters as necessary to make the operation name unique. For example, if both the **end** and **edit** operations appear on a menu, you need to type **ed** to use the edit operation or **en** to use the end operation.
3. Press **Return**.

C.4.3 Moving Between Menus

When you choose an operation from a menu, you are often presented with another ULTRIX/SQL frame containing a submenu of operations. An operation chosen from that submenu may in turn present another frame with a further set of operations.

When you leave a submenu with the **End** operation, you always return to the next higher level menu.

When you leave a submenu with the **Quit** operation, you will return to the main menu. When you quit from the **isql** main menu, you quit interactive ULTRIX/SQL and return to the operating system prompt.

C.5 Standard ULTRIX/SQL Operations

Certain operations appear frequently on ULTRIX/SQL frames. Table C-3 lists the standard ULTRIX/SQL menu operations.

Table C-3 Standard ULTRIX/SQL Operations

Operation	Function
Blank	Clears the screen of all entries.
Bottom	Moves the cursor to the bottom of a table field.
End	Ends operations on the current screen and returns to the previous screen.
Find	Searches for a specified string of characters within text or a list (table field) on the screen.
Forget	Returns to a previous screen without saving changes entered on the current screen.
Go	Follows the specified request.
Help	Displays Help screens relevant to the current action.
Insert	Puts a blank line at the cursor location.

Operation	Function
Quit	Exits the module and returns to the operating system.
Shell	Escapes to the operating system without ending the ULTRIX/SQL session. Enter "exit" to resume the ULTRIX/SQL session.
Top	Moves the cursor to the top of a table field.
Undo	"Undoes" or cancels the previous operation.

C.6 ULTRIX/SQL Keys

Through the use of key mapping, you can attain a high degree of flexibility in your keyboard interaction with ULTRIX/SQL.

C.6.1 Function Keys

The way the different keys on your keyboard function in ULTRIX/SQL depends almost entirely on your terminal and the specific choices you make for individual keys on your terminal keyboard. Different terminals support function keys in different ways. See Appendixes D and E.

To review the way the keys on your keyboard are assigned, use ULTRIX/SQL online help. Place the cursor on the operations menu line and type **h** for **Help**. If you already know which key is assigned this function, press the **Help** key. For information on how to use **Help**, see "On-Screen Help" below.

C.6.2 Cursor Movement and Editing Keys

Some keys facilitate moving the cursor within forms on the screen. Although this is terminal-dependent, you can often use keys in the numeric keypad on the right of your keyboard to control scrolling and other movements of the cursor on the screen.

The way the different control keys on your keyboard function in ULTRIX/SQL again depends entirely on your terminal and the specific choices you make. See Appendix E. For example, some terminals do not support arrow key cursor movement. You can see the current mappings for your terminal by choosing **Help** from an operations menu and then choosing **Keys** from the Help submenu.

Certain keys operate consistently throughout ULTRIX/SQL. These are the **Tab** key, the combination of **Control** and **P**, and the **Return** key. They all move the cursor from field to field but with the following differences:

- The **Tab** key moves the cursor to the next field or next column in a table field. If the cursor is in the last field, **Tab** moves the cursor to the first field of the same form.
- The **Control-P** combination moves the cursor back to the previous field or previous column in a table field. Use **Tab** or **Return** to move forward again.

- The **Return** key moves the cursor to the next field *and clears data to the end of the field at the same time* (unless the form or the table field is read only). In a table field, **Return** moves the cursor to the next column. If the cursor is in the last column, **Return** moves the cursor to the first column of the next row.

If you do not want the **Return** key to clear the data to the end of the field, you can map the **Return** key so that it works like the **Tab** key. See Appendix E for information on how to do this. Instead of clearing to the end of the field or modifying any field item, the cursor simply moves to the next item. If the cursor is in a simple field or any table field column but the last, **Return** moves the cursor to the next field or column. If the cursor is in the last column of a table field, **Return** moves the cursor to the first column of the next table field row.

C.6.3 Insert and Overstrike

Insert and *overstrike* modes allow you to work in either overstrike or insert mode. In overstrike mode, each character typed replaces the existing character beneath it. In insert mode, characters are moved to the right as you enter new characters.

The default mode is overstrike. The **Control-E** key combination is a toggle that enables you to change between modes.

C.7 On-Screen Help

ULTRIX/SQL provides context-sensitive help. This means that the assistance is based on the current task you are attempting to accomplish and the current field you are attempting to complete. You can obtain help by placing the cursor on the operations menu line and typing **h** for **Help**. You can also press the **Help** key to get help at any time. For the VT100, the **Help** key is normally **PF2**; for the VT220 it is the **Help** key itself. Sometimes, help is provided on several screens. Table C-4 describes the Help screen menu option.

Table C-4 Help Screen Menu Options

Operation	Function
What to do	Describes the current screen and the operations menu.
Keys	Describes the function and control keys and their current definition.
Field	Displays a list of valid values for a field or the display format, data type, and validation check, if any, for a field.
Help	Displays the type of Help available.
End	Exits from any Help screen to the previous screen.

To make a selection on a Help screen, type the first few unique characters of the operation and press **Return**, or press one of the keys in parentheses after the operation. To move through the Help screens, use the cursor movement keys specific to your terminal. You can see a list of the keys available by selecting the **Keys** operation from the Help menu.

C.8 Error Messages

ULTRIX/SQL provides context-sensitive error messages in a pop-up window that appears along the bottom of your screen. The error message you receive indicates both the error type and the error code.

ULTRIX/SQL also provides explanations for many of the errors. For messages without an explanation, ULTRIX/SQL displays a single-line message with a prompt that tells you to press **Return** after reading the message. When you press **Return,** ULTRIX/SQL removes the message and returns you to your work in progress.

For messages with explanations, ULTRIX/SQL displays the first line in the pop-up window with a prompt that tells you to press either the designated **End** key or the designated **More** key. To exit the message without reading the explanation, press the **End** key. To read the explanation, press the **More** key.

When you press the **More** key, ULTRIX/SQL displays the error message explanation. After reading the explanation, press **Return** to return to your work in progress.

Defining Your Terminal D

D.1 Overview

Before you use the ULTRIX/SQL forms-based commands, **accessdb**, **catalogdb**, or **isql**, you must define your terminal to ULTRIX/SQL to make the features of the forms system available to you. Your computer system can probably support a wide variety of terminals, each with its own particular characteristics. If you do not define your terminal to ULTRIX/SQL, ULTRIX/SQL will not permit you to use any of its forms-based utilities.

This appendix describes defining your terminal to ULTRIX/SQL and lists the names for various commercially available terminals on which you can run ULTRIX/SQL. It also includes information on additional features of terminal use.

D.2 Defining Your Terminal

To define your terminal, select the appropriate command for your shell. Note that once the ULTRIX/SQL forms system has started, the environment variable TERM_INGRES cannot be reset until the session has ended. It is a good idea to include the command that defines TERM_INGRES in the automatic login procedures on your individual account on the computer.

For the C shell:

setenv TERM_INGRES *termname*

For the Bourne shell:

TERM_INGRES=*termname* **export TERM_INGRES**

where *termname* is the designation for your terminal type (see "Terminal Names for ULTRIX/SQL" below).

For instance, if you have a VT100 terminal and you want to be able to use the arrow keys as cursor movement keys and the keypad keys as definable function keys, consulting "Terminal Names for ULTRIX/SQL" tells you that "vt100i" is the proper designation. To define your terminal accordingly to ULTRIX/SQL, enter one of the following commands:

C shell:

```
setenv TERM_INGRES vt100i
```

Bourne shell:

```
TERM_INGRES=vt100i
export TERM_INGRES
```

Thereafter, you may use the default assignment of key strokes to cursor motion and forms commands with ULTRIX/SQL.

The "vt100nk" is another terminal designation available for VT100 terminals. This terminal designation is particularly suited to applications that require use of the keypad for numeric input. The designation gives the user access to the arrow keys and the top four function keys on the numeric keypad. The other keys on the keypad are available for numeric input.

Note

> VT220 and VT320 series terminals are fully supported by ULTRIX, and you may use the VT220 and VT320 keystrokes. However, you should use the generic definitions, "vt200" for the VT220 terminal and "vt300" for the VT320 terminal.

D.3 Additional Features of Terminal Use

In addition to the capabilities of the ULTRIX/SQL Forms Run-Time System described in Appendix E, ULTRIX/SQL provides the ability to:

- Print the screen
- Redraw the screen

D.3.1 Printing the Screen

If you have defined your terminal to use function keys (by setting TERM_INGRES to vt100i, for example), you can print the contents of the currently displayed screen.

This may be useful for preparing documents in which the status of the terminal screen must be depicted. To print the current screen, press the key assigned to that function. On a VT100, press **Control-G**, and on a VT200 series or VT300 series, press **F8**. This function assignment may vary from terminal to terminal.

A prompt appears:

```
Enter file name:
```

ULTRIX/SQL appends an image of the current form, including all displayed data values, to the specified file. The entire form is included, even if it is longer and wider than the terminal screen.

If you enter the special name **printer** as the filename, the image is sent to the printer.

You may also set the environment variable II_PRINTSCREEN_FILE to specify a filename to which the results of the printscreen function are automatically written. See the *ULTRIX/SQL Operations Guide* for more information on this variable.

D.3.2 Redrawing the Screen

You can also *redraw* the current screen, including any data you have entered into its field. This is useful if you receive messages on the screen or if disruptions in communication with the computer occur. This redrawing function is assigned by default to **Control-W**, regardless of the terminal.

D.4 Terminal Names for ULTRIX/SQL

Digital Equipment Corporation has every reason to believe that ULTRIX/SQL forms-based operations should function correctly on the terminals listed here. However, not all these terminals have been tested to determine the functionality of the ULTRIX/SQL forms run-time system. If you have any problems with the terminal designations, please submit an SPR (Software Performance Report).

Table D-1 Terminal Names for ULTRIX/SQL

Terminal Type	Menu Key	Name
ADDRINFO	ESC	addrinfo
ADDS CONSUL 980	ESC	a980
ADDS REGENT 100	ESC	regent100
ADDS REGENT 20	ESC	regent20
ADDS REGENT 25	ESC	regent25
ADDS REGENT 40	ESC	regent40
ADDS REGENT 60	ESC	regent60
REGENT 60 w/no arrow keys	ESC	regent60na
ADDS REGENT SERIES	ESC	regent
AMPEX DIALOGUE 80	ESC	ampex
ANN ARBOR	ESC	aa
ANN ARBOR AMBASSADOR 48 with destructive backspace	ESC	aaadb
ANN ARBOR AMBASSADOR/48 lines	ESC	aaa
BEEHIVE SUPER BEE	ESC	sb1
FIXED SUPERBEE	ESC	sb2
BEEHIVE IIIm	ESC	bh3m
CONCEPT 100	ESC	c100
CONCEPT 100 slow	ESC	c100s

Terminal Type	Menu Key	Name
CONCEPT 100 slow reverse video	ESC	c100rvs
C100 reverse video	ESC	c100rv
C100 with 4 pages	ESC	c1004p
C100 reverse video with 4 pages	ESC	c100rv4p
C100 with no arrows, reverse video, 4 pages	ESC	c100rv4pna
C100 with printer port, reverse video, 4 pages	ESC	c100rv4ppp
CDC	ESC	cdc456
CDC456tst	ESC	cdc456tst
CDI1203	ESC	cdi
COMPUCOLORII	ESC	compucolor
CYBERNEX mdl-110	ESC	mdl110
DATAMEDIA 1520	ESC	dm1520
DATAMEDIA 1521	ESC	dm1521
DATAMEDIA 2500	ESC	dm2500
DATAMEDIA 3025a	ESC	dm3025
DATAMEDIA 3045a	ESC	dm3045
DATAMEDIA dt80/1	ESC	dt80
DATAMEDIA dt80/1 in 132 character mode	ESC	dt80132
DATAPOINT 3360	ESC	datapoint
DELTA DATA 5000	ESC	delta
DIGILOG 333	ESC	digilog
ENVISION	PF1	envision
ENVISION with color	PF1	envisionc
GENERAL TERMINAL 100A (formerly INFOTON 100)	ESC	i100
HAZELTINE 1500	ESC	h1500
HAZELTINE 1510	ESC	h1510
HAZELTINE 1520	ESC	h1520
HAZELTINE 1552	ESC	h1552

Terminal Type	Menu Key	Name
HAZELTINE 1552 reverse video	ESC	h1552rv
HAZELTINE 2000	ESC	h2000
HEATHKIT h19	ESC	h19
HEATHKIT h19 ansi mode	ESC	h19A
HEATHKIT with keypad shifted	ESC	h19bs
HEATHKIT with keypad shifted, underscore cursor	ESC	h19us
HEATHKIT with underscore cursor	ESC	h19u
HEWLETT PACKARD 2621	ESC	2621
HEWLETT PACKARD 2621 with 45 keyboard	ESC	2621k45
HEWLETT PACKARD 2621 with labels	ESC	2621wl
HEWLETT PACKARD 2621 with no labels	ESC	2621nl
HEWLETT PACKARD 2621 48 lines	ESC	big2621
HEWLETT PACKARD 2626	ESC	hp2626
HEWLETT PACKARD 2640a	ESC	2640
HEWLETT PACKARD 2648a graphics terminal	ESC	hp2648
HEWLETT PACKARD 264x series	ESC	2640b
HEWLETT PACKARD 264x series	ESC	hp
IBM 3101-10	ESC	ibm
INFOTON 400	ESC	i400
INFOTON KAS	ESC	infotonKAS
ISC modified owl 1200	ESC	intext
ISC8001	ESC	8001
LSI adm2	ESC	adm2
LSI adm3	ESC	adm3
LSI adm3a+	ESC	adm3a+
LSI adm31	ESC	adm31
LSI adm3a	ESC	adm3a

Terminal Type	Menu Key	Name
LSI adm42	ESC	adm42
MICRO BEE SERIES	ESC	microb
MICROTERM ACT IV	ESC	microterm
MICROTERM ACT V	ESC	microterm5
MICROTERM MIME1	ESC	mime
FULL BRIGHT MIME1	ESC	mimefb
HALF BRIGHT MIME1	ESC	mimehb
MICROTERM MIME2A (emulating an enhanced SOROC iq120)	ESC	mime2as
MICROTERM MIME2A (emulating an enhanced VT52)	ESC	mime2a
MIME1 emulating 3A	ESC	mime3a
MIME1 emulating enhanced 3A	ESC	mime3ax
NETRONICS	ESC	netx
PERKIN ELMER 1100	ESC	fox
PERKIN ELMER 1200	ESC	owl
SOL	ESC	sol
SOROC 120	ESC	soroc
SOUTHWEST TECHNICAL PRODUCTS CT82	ESC	swtp
SUPER BEE with insert character	ESC	superbeeic
TEKTRONIX 4105	PF1	tk4105
TEKTRONIX 4105 with color	PF1	tk4105c
TELERAY 1061	ESC	t1061
TELERAY 1061 with fast PROMs	ESC	t1061f
DUMB TELERAY 3700	ESC	t3700
TELERAY 3800 series	ESC	t3800
TELETEC DATASCREEN	ESC	teletec
NEW TELEVIDEO 912	ESC	912b
NEW TELEVIDEO 920	ESC	920b

Terminal Type	Menu Key	Name
OLD TELEVIDEO 912	ESC	tvi912
OLD TELEVIDEO 920	ESC	tvi920
VISUAL 200 no function keys	ESC	vi200f
VISUAL 200 reverse video	ESC	vi200rv
VISUAL 200 reverse video using insert character	ESC	vi200rvic
VISUAL 200 using insert character	ESC	vi200ic
VISUAL 200 with function keys	ESC	vi200
VT100 with function keys activated	PF1	vt100f
VT100 with function keys activated (3.0 version)	PF1	vt100k
VT100 with numeric keypad	PF1	vt100nk
VT100 without function keys activated	ESC	vt100
VT100 in 132-column mode with function keys activated	PF1	vt100fw
VT100 in 132-column mode with function keys activated (3.0 version)	PF1	vt100kw
VT100 in 132-column mode with numeric keypad	PF1	vt100nkw
VT100 with function keys activated and **Return** key mapped to **Nextitem** instead of **Clearrest** (works like vt100f, but Return does not clear to end of field)	PF1	vt100i
VT100 in 132-column mode with function keys activated and **Return** key mapped to **Nextitem** instead of **Clearrest** (works like vt100fw, but Return does not clear to end of field)	PF1	vt100iw
VT100 in 132-column mode without function keys activated	PF1	vt100w
VT100 with no initialization	ESC	vt100n
VT125	ESC	vt125
VT220	PF1	vt220
VT220 with **Return** key mapped to **Nextitem** instead of **Clearrest**	PF1	vt220i
VT220 in 132-column mode	PF1	vt220w

Terminal Type	Menu Key	Name
VT220 in 132-column mode with **Return** mapped to **Nextitem** instead of **Clearrest**	PF1	vt220iw
VT241	PF1	vt241
VT300 series	PF1	vt300
VT50	ESC	vt50
VT50h	ESC	vt50h
VT52	ESC	vt52
VT132	ESC	vt132
XEROX 1720	ESC	x1720
XITEX sct-100	ESC	xitex
ZENTEC 30	ESC	zen30

Defining Function and Control Keys E

E.1 Overview

The ULTRIX/SQL forms run-time system (FRS) is a built-in screen management system that is a common interface to the ULTRIX/SQL forms-based utilities **accessdb, catalogdb** and **isql**. The forms system enables you or the ULTRIX/SQL System Administrator to redefine the function and control keysKeyboard keys;Mapping!B to be used in the preceding forms-based utilities. The ULTRIX/SQL System Administrator is usually responsible for implementing such customization of the terminal user's environment.

All operations–menu item operations, cursor movement, and so forth–can be mapped to function or control keys on a terminal. Once the mapping has been specified, you can execute the operation by simply pressing the specified key. If the terminals at your installation do not support function keys, you can still map operations to control keys, so that entering a control character will execute the operation.

The ULTRIX/SQL forms system allows you to define control and function key mappings on three levels:

- Installation
- Terminal type
- User

This allows you to tailor key definitions to the specific requirements of the environment and the user.

Note

> The text and examples in this appendix refer mostly to the VT100 and VT200 series terminals. Most of what applies to the VT200 series also applies to the VT300 series terminals.

E.2 The Purpose of the ULTRIX/SQL Termcap File

Before you can take advantage of the function key feature, the terminal must be defined to ULTRIX/SQL so that *physical* keys on the terminal become associated with *logical* ULTRIX/SQL functions. Through this mapping, the physical keys can execute various operations that are built into the ULTRIX/SQL forms system.

The locations and availability of function (PF) and control keys are unique to the type of terminal you are using. Control keys are available on all ASCII terminals, but only certain types of terminals also support function keys. The specific function keys and the escape sequences they generate are documented by the terminal vendor.

The ULTRIX/SQL termcap file contains a description of all terminals supported by ULTRIX/SQL, including their available function and control keys. Each supported terminal has a termcap entry that is based on the vendor's specifications for that device. Appendix D lists supported terminals. For unsupported terminals, you must write your own termcap entries (see Appendix F).

When you start up one of the ULTRIX/SQL forms-based utilities, the forms run-time system uses the TERM_INGRES logical to determine the user's terminal type and verifies basic terminal attributes. The terminal type tells the forms system which entry to read from the ULTRIX/SQL termcap file. The type is checked only once for each session, so the user must exit the current session in order to change terminal types. However, logical key definitions can be changed dynamically by the application.

Note that fundamental changes to the forms system's interpretation of keystrokes for a particular terminal would require modification of that terminal's termcap entry. For example, if you want the forms system to permanently interpret a Down Arrow key as **Nextfield** for the VT100i, you must change the ULTRIX/SQL termcap entry for VT100i. Key map files cannot accomplish such changes directly.

As noted in the termcap entry comments, some termcap entries allow you to define the terminal to recognize function keys (see Appendix D). For example, to use function keys with a VT100 or VT100-like terminal, the terminal should be defined as vt100i. The vt100i definition turns the terminal's keypad into a set of 18 function keys, as shown in Figure E-1.

Figure E-1 VT100i Function Keys on Keypad

PF1 (PF1)	PF2 (PF2)	PF3 (PF3)	PF4 (PF4)
7 (PF5)	8 (PF6)	9 (PF7)	– (PF8)
4 (PF9)	5 (PF10)	6 (PF11)	, (PF12)
1 (PF13)	2 (PF14)	3 (PF15)	ENTER (PF18)
0 (PF16)		. (PF17)	

In the figure, the numbers correspond to the numbers that appear on the physical keys, while those in parentheses correspond to the function key *represented by* the physical key (for example, key #7 represents the fifth function key (**PF5**)).

By default, a VT220 terminal uses function keys; therefore this terminal should be defined as vt220i. A VT220 can also emulate a VT100 terminal. If the terminal itself is set this way, it is appropriately defined to ULTRIX/SQL as vt100i. Other types of terminals with function keys can be defined to accept function key mappings by editing the termcap file entries for the terminal types, as described in Appendix F.

Users of terminals defined as vt100i or vt220i can, in addition, use arrow keys to move the cursor. The use of arrow keys on any other terminal type may require the editing of its termcap file entry.

Note that no special terminal definition is required for control keys. Certain control keys, however, are reserved by the operating system for its own use and should not be mapped to any operations. These may include, but are not limited to:

- **Control-C**
- **Control-O**
- **Control-Q**
- **Control-S**
- **Control-T**
- **Control-X**
- **Control-Y**

Also, if your terminal uses escape sequences to define function keys (the case for terminals defined as vt100i and vt220i and most other terminals), you must also consider **Escape** as reserved.

E.3 Defining Function and Control Key Mappings

Typically, entries within *mapping* files define a connection between *physical* function and control key mappings and *logical* objects that can then be accessed by the forms system.

The mapping file uses a simple yet powerful syntax. The following example of a mapping file illustrates the full range of statements available to specify any sort of mapping:

```
/* This is an example of a mapping file */
menuitem2 = pf3 (Key 3)
menuitem3 = controlE (^E)
frskey7 = pf8
previousfield = controlP
rubout = controlDEL
controlA = off
/* this turns control-A off */
pf7 = off
```

The first line of this sample file is a *comment.* Comments can appear anywhere in a mapping file. Their purpose is to provide information to someone looking at the file; they are ignored by the forms system.

The next five lines are examples of *mapping statements.* All mapping statements follow the same basic syntax. To the left of the equal sign is a *mapping object,* which specifies the operation or function to which the key is being mapped. To the right of the equal sign is the physical control or function key that maps to and will activate the mapping object. For instance, the following statement specifies that function key 3, or **PF3**, maps to the second item on each menu line ("menuitem2"):

```
menuitem2 = pf3
```

A user may perform the operation specified by the second item on any menu simply by pressing **PF3**.

The next statement enables the user to move the cursor to the previous field on a form by pressing **Control-P**:

```
previousfield = controlP
```

A few of the mapping statements in the long example above contain *labels* within parentheses to the right of the function or control key. These labels serve the purpose of providing information to the user, as described later, and do not affect the actual mapping between object and key.

The last two statements in the example are known as *disabling statements.* A disabling statement is used to disable a control or function key. For example, the following statement turns **PF7** off:

```
pf7 = off
```

This means that the key has no effect in a forms-based application governed by this map file and merely produces a beep when pressed.

E.4 Types of Mapping Objects

You can map function and control keys to several types of objects. These mappings allow the keys to be used to perform menu item operations, cursor movement, and virtually any other functions available in the ULTRIX/SQL forms-based utilities.

The three types of mapping objects are:

- FRS commands
- Menu items

- FRS keys

Each of these types of mapping objects is described in its own section below.

E.4.1 FRS Commands

FRS commands are built-in functions of the forms system, which enable the user to view and edit the data on a form. Through key mapping, you can link such capabilities as deleting a character, moving to the menu line, or scrolling within a table field, to specific function or control keys. For example, the following statement maps a physical key, **Control-P**, to an FRS command, **Previousfield**:

```
previousfield = controlP
```

Any function or control key may be mapped to any FRS command. The following table lists the FRS commands and their definitions.

Table E-1 FRS Commands

FRS Command	Meaning
menu	Goes to the menu line (the **Menu** key).
nextfield	Goes to the next field on the form.
previousfield	Goes to the previous field on the form.
nextword	Moves forward one word within a field.
previousword	Moves backward one word within a field.
mode	Switches between insert and overstrike editing mode.
redraw	Redraws the frame.
deletechar	Deletes the character at the current cursor position.
rubout	Deletes the character immediately to the left of the cursor.
editor	Starts the text editor on the current field.
leftchar	Moves left one character within the current field.
rightchar	Moves right one character within the current field.
downline	Moves down one line in the current field or next row in the table field.
upline	Moves up one line in the current field or previous row in the table field.
newrow	Moves to the first column of the next row in the table field.
clear	Clears the current field or menu input.

FRS Command	Meaning
clearrest	Clears the rest of the field, beginning from the current cursor position. Then moves the cursor to the next field if the cursor is not in a table field, to the next column if the cursor is not in the last column of a table field, or to the first column of the next row if the cursor is in the last column of a table field.
scrollup	Scrolls up in the current table field or form, leaving the cursor on the same field.
scrolldown	Scrolls down in the current table field or form, leaving the cursor on the same field.
scrollleft	Scrolls the form to the left, leaving the cursor on the same field.
scrollright	Scrolls the form to the right, leaving the cursor on the same field.
duplicate	Auto-duplicates a simple field value.
printscreen	Sends a copy of the form currently displayed to a file or the printer.
Nextitem	Moves to the next field if the cursor is not in a table field, to the next column if the cursor is not in the last column of the table field, or to the first column of the next row if the cursor is in the last column of the table field. Unlike **Clearrest**, **Nextitem** does not clear the rest of the current field.

ULTRIX/SQL provides a default installation-wide mapping file, along with default mapping files for terminals. These mapping files assign keys to some or all of the FRS commands in $II_SYSTEM/sql/files. Check online for the listing of mapping files to see whether your terminal has one, or see your ULTRIX/SQL System Administrator.

The section "Levels of Mapping" below provides the text of the default mapping files. The section "Default Settings for FRS Commands" lists the results of those mappings, the default keys for the various FRS commands.

When used as mapping objects, FRS commands should be typed as shown in the table above.

Note

FRS commands cannot be executed directly in program code; they are only available to the user through mapping to terminal keys.

E.4.2 Menu Items

You can also map a function or control key to any of the items appearing on a menu line. This mapping occurs by position within the menu line.

The syntax for a menu item is:

menuitem*N*

where *N* is in the range 1 to 25, indicating the position of the menu item in the line. Alternatively, you can also designate **menuitem*N*** as **menu*N*** in a mapping statement.

For example, the following statement maps **PF3** to the second item on the menu line:

```
menuitem2 = pf3
```

This statement causes **PF3** to perform the operation indicated by the second menu item. As the user moves to a new frame and the menu changes, **PF3** continues to correspond to the item in the second position on the new menu line.

By default, the menu line automatically displays the current mappings between menu items and function/control keys. It uses either the label provided in the map file or a default label, if none has been specified. For example, assuming such mappings have been specified, a menu may appear like this:

```
Help(PF2) Add(PF3) Editor(control-E) End(PF4)
```

In this example, pressing **PF2** is equivalent to moving to the menu line and typing **Help**. Similarly, **PF3** selects the **Add** option, **Control-E** selects the **Editor** option and **PF4** selects the **End** option.

The text that shows the corresponding function or control key, such as **PF2** is known as a *label*. The label appears on the menu as specified in the mapping file (see "Mapping File Syntax").

E.4.3 FRS Keys

FRS keys enable you to invoke specific operations in ULTRIX/SQL forms-based utilities using standardized function or control keys, regardless of which function/control key is mapped to its current menu item position. These utilities consistently equate standard operations with certain FRS keys. For example, the **Help** operation is always associated with **frskey1**. If **frskey1** is mapped to the ASCII character **controlH**, you can always get help by pressing **Control-H**, regardless of its current menu item key designation. These standard operations are often located at the end of the menu line, allowing those operations that are unique to a particular frame to appear first.

Because terminals vary with respect to available function keys, ULTRIX/SQL forms-based operations are not mapped directly to *physical* function or control keys. To achieve the needed flexibility, these operations have been equated with *logical* FRS keys. You can map these logical FRS keys to actual function or control keys on your terminal at any of the three mapping levels.

If an operation is associated with both a menu item and an FRS key, you can activate the operation either with the key that is mapped to the menu item's position or with the key that is mapped to the FRS key. Whenever an operation has been associated with both a menu item and an FRS key, the label for the menu item indicates the function/control key that maps to the corresponding FRS key, *not* the function/control key that maps to the item's position on the line.

Table E-2 lists the FRS keys and their meanings.

Table E-2 Predefined FRS Keys

FRS Key	Menu Item	Meaning
frskey1	**Help**	Accesses the ULTRIX/SQL help facility.
frskey2	**Quit**	Exits from ULTRIX/SQL.
frskey3	**End**	Exits the frame, returning to the previous frame.
frskey4	**Go/Next**	Executes the current function.
frskey5	**Top**	Moves to the top of the table field.
frskey6	**Bottom**	Moves to the bottom of the table field.
frskey7	**Find**	Searches the table field for the specified string.
frskey8	**Save**	Saves the object in the database.
frskey9	**Undo/Forget**	Undoes the last action or rolls back the changes made in the frame.

Mapping the FRS key to a function/control key is handled identically to mapping a menu item or FRS command. An FRS key is designated by the key word **frskey**, followed by an integer in the range 1 to 40. For example, the following statement maps FRS key 7 to **PF2**:

```
frskey7 = pf2
```

By pressing **PF2**, the user invokes whatever operation FRS key 7 has been defined as within the current frame.

E.4.4 Mapping File Syntax

Mapping files are the main method for mapping function or control keys to menu items, FRS commands, and FRS keys. These files can be created for each of three levels of mapping: installation, terminal type and user. The syntax for the mapping files is the same across all levels of mapping; the only way that the FRS distinguishes one level from another is by the way the mapping file is defined to it. See the section "Levels of Mapping" below for information on how to do this.

As mentioned earlier, a mapping file may consist of three components:

- *Mapping statements* to designate the actual mappings. These are the most commonly used. The mapping statements may also designate nonstandard labels for menu items.
- *Disabling statements* to disable the use of function/control keys.
- *Comments* to provide explanatory text.

The statements in a mapping file can appear in any order. However, each statement must fit entirely on one line. Alphabetic characters within a mapping file may appear in either upper- or lowercase with no difference in meaning. Blank lines are ignored.

E.4.4.1 Mapping Statements

A mapping statement has the following syntax:

mapping_object = **pf***N*|**control***X*[(*label*)]

Parameter	Description
mapping_object	Specifies a menu item, FRS command, or FRS key, designated through mapping.
pf*N*	Designates a function key. *N* must be in the range 1 to 40. (Note the maximum number of definable keys set in the termcap file for your terminal may be less than 40. You may need to raise this limit in order to set additional keys.)
control*X*	Designates a control key sequence. *X* may be any single letter, or the designations **del** to indicate the **Delete** key or **esc** to indicate the **Escape** key (for instance, **controldel** or **controlesc**). **Control** may be abbreviated **ctrl**. (Note that on most terminals **Control-M** is equivalent to the **Return** key and **Control-I** is equivalent to the **Tab** key.)
label	Specifies an alphanumeric string identifying the key to press. It appears in place of the default label for a menu item. It also appears in the **Keys** operation of the ULTRIX/SQL help facility.

Each mapping object can map to only a single control or function key at a time. While you can map more than one mapping object to the same physical key, one mapping will override the others based on the precedence described in the "Levels of Mapping" section.

The exception to this rule might appear to occur when a menu item can be called either by a function/control key mapped to it by the item's position or by a function/control key mapped to an FRS key which is equivalent to the menu item. However, this is not really an exception at all, because two mapping objects, the menu item and the FRS key, are mapped to two different function/control keys.

Conversely, within a file, each control or function key can map to only a single mapping object at a time. If any conflicting mappings occur in the file, the first mapping takes precedence.

The following example illustrates mapping statements:

```
frskey8 = pf16 (0)
menuitem1 = pf13 (1)
menuitem2 = pf3 (PF3)
menuitem3 = controlE (^E)
previousfield = controlP
menuitem4 = pf9 (4)
rubout = controlDEL
```

In this example, **PF16** ("0" on a VT100 keypad) maps to the operation associated with FRS key 8, **PF13** ("1" on a VT100 keypad) activates the first item on the menu line, **PF3** activates the second, **Control-E** activates the third menu item, **Control-P** moves the cursor to the previous field on the form, **PF9** activates the fourth item on the menu line, and **Control-Delete** deletes the character immediately to the left of the cursor. Any previous mappings that do not conflict with these statements remain in effect.

Notice the effect that including an explicit label has on the appearance of a menu line. Assume a frame's menu includes the following operations:

```
Help    Add    Editor    End
```

Assume, also, that the frame containing these menu items also specifies that the **Help** operation be invoked either by selecting the **Help** menu item or by pressing the key mapped to FRS key 8. The mapping file above, with its labels, would cause the menu to appear as follows:

```
Help(0)    Add(PF3)    Editor(^E)    End(4)
```

Two different labels from the map file could be used for the **Help** menu item: "1" (the label for the first menu item) or "0" (the label for FRS key 8). In a case like this, the label for the FRS key takes precedence. All other labels on the menu are those associated with the menu item's position in the menu line.

E.4.4.2 Disabling Statements

Any of the function/control keys can be disabled. Once disabled, a key remains so until used within a mapping statement of higher priority.

A disabling statement has the following syntax:

pf*N*|**control***X* = **off**

The following two statements disable **Control-A** and **PF7**:

```
controlA = off
pf7 = off
```

While these statements are in effect, typing **Control-A** or **PF7** will only produce a beep from the terminal.

To disable an FRS command, map the FRS command to a control or function key and then disable that key.

E.4.4.3 Comments

Comments are delimited by **/*** and ***/**. They may appear anywhere, including on the same line as a statement. The whole comment must appear within a single line. Below are two examples of comments:

```
/* This is a comment */
controlA = off /*this turns Control-A off*/
```

E.4.4.4 Mapping File Errors

As described in detail in the following sections, when the FRS starts up, the mapping files for the various levels of mapping are merged, and conflicts between files are resolved based on a specific precedence. When the FRS detects errors in a mapping file, you may find detailed error messages in a file called **ingkey.err** in the user's current directory. After exiting the utility, the user can look at the error file to determine the nature of the errors.

E.5 Levels of Mapping

Function and control key mappings may be defined on three separate levels:

- installation
- terminal type
- user (environment)

The user-level mapping has the highest precedence. This mapping allows each individual user a good degree of latitude in the use of function/control keys. Through terminal type-level mapping, a default can exist for all terminals of a given type, such as VT100s or VT200s. This default is overridden by any conflicting user mappings. The installation-level mapping is overridden by all other mappings.

When the FRS starts up, the three levels of mapping are merged, and conflicts are resolved based on the precedence outlined above. While the three levels of mappings may coexist, any of the levels can be omitted. Since it is possible that a function/control key will be defined at more than one of the three levels, the FRS always honors the most *recent* reference to any mappable key from a higher-level precedence file.

E.5.1 Installation-Level Mapping

As mentioned in the preceding section, installation-level mapping has the lowest precedence. It provides an underlying default, common to all terminal types, which can be overlaid with mappings for specific terminal types, as well as mappings for individual users and applications.

A default installation-level mapping file is shipped with ULTRIX/SQL. Since this default file references only control keys and not function keys, it should be usable for all ULTRIX/SQL terminal types.

The complete file specification for this mapping file is:

```
$II_SYSTEM/sql/files/frs.map
```

It contains the following statements:

```
/* Move cursor to next field */
    nextfield = controlI (Tab)

/* Move cursor to previous field */
    previousfield = controlP (^P)
```

```
/* Move up one word within field */
    nextword = controlB (^B)

/* Move back one word within field */
    previousword = controlR (^R)

/* Switch between insert and */
/* overstrike mode*/
    mode = controlE (^E)

/* Redraw the screen */
    redraw = controlW (^W)

/* Delete character under the cursor */
    deletechar = controlD (^D)

/* Delete character to left of cursor */
    rubout = controlDEL (Delete)

/* Start default text editor on field */
    editor = controlV (^V)

/* Move left one space within a field */
    leftchar = controlH (^H)

/* Move right one space within a field */
    rightchar = controlL (^L)

/* Move down one line */
    downline = controlJ (^J)

/* Move up one line */
    upline = controlK (^K)

/* Move to first column of next row */
/* in table field */
    newrow = controlN (^N)

/* Clear the field */
    clear = controlX (^X)

/* Clear out rest of field */
/* and move to next field */
    clearrest = controlM (Return)

/* Scroll up on the form */
    scrollup = controlF (^F)

/* Scroll down on the form */
    scrolldown = controlG (^G)

/* Scroll left on a form */
    scrollleft = controlO (^O)

/* Scroll right on a form */
    scrollright = controlU (^U)

/* Auto-duplicate value while in fill mode*/
    duplicate = controlA (^A)
```

While you can edit this file to customize your installation's default mappings, you should *not* change the name of this file. The forms run-time system automatically looks for the file when starting up. If you do modify this file, be sure to map only those keys that are available for all terminal types at your installation.

Notice that the file does not specify a **Menu** key. This is because the FRS command **menu** automatically defaults to **Escape**.

E.5.2 Terminal-Type Level Mapping

The next higher level of mapping is terminal type. Each terminal type used at your installation may need its own mapping file because function key support varies from terminal to terminal. Combined with the installation mapping, the terminal mapping files provide common terminal defaults, which can be altered by mappings at higher levels to fit the needs of individual users.

The terminal-type mapping file must be placed in the following directory:

```
$II_SYSTEM/sql/files
```

You can specify the filename with the **mf** capability in the termcap file entry for each terminal type in use at an installation. (The termcap file is discussed in Appendix F.) You can also point to a termcap file by using the II_TERMCAP_FILE environmental variable.

Default mapping files for VT100 and VT220 terminals are shipped with ULTRIX/SQL. You may edit these files if desired. (The termcap file entries for the vt100i and vt220i terminal definitions discussed earlier already have the names of their mapping files specified; therefore, there is no need to edit those termcap entries.)

E.5.2.1 VT100 Terminals

For VT100 terminals defined as the vt100i terminal type, the mapping file location is:

```
$II_SYSTEM/sql/files/vt1imap.unx
```

It contains the following statements:

```
/* Menu Key */
    menu = pf1 (PF1)

/* Help facility */
    frskey1 = pf2 (PF2)

/* Quit from program */
    frskey2 = pf4 (PF4)

/* End current screen and return */
/* to previous screen */
    frskey3 = pf3 (PF3)

/* Go or execute function */
    frskey4 = pf18 (Enter)

/* Put cursor on top of form or */
/* table field*/
    frskey5 = controlK (^K)
```

```
/* Put cursor on bottom of form or */
/* table field*/
    frskey6 = controlJ (^J)

/* Find next occurrence of string*/
/* in this column of table field  */
    frskey7 = controlF (^F)

/* Save object in database */
    frskey8 = pf16 (0)

/* Forget and undo */
    frskey9 = pf17 (.)

/* Scroll page or form left */
    scrollleft = controlL (^L)

/* Scroll page or form right */
    scrollright = controlH (^H)

/* Previous screen or set of rows */
/* in table field */
    scrolldown = pf8 (-)

/* Next screen or set of rows in */
/* table field */
    scrollup = pf12 (,)

/* Print contents of current screen */
/* to file or printer */
    printscreen = controlG (^G)

/* Select first menu item */
    menuitem1 = pf13 (1)

/* Select second menu item */
    menuitem2 = pf14 (2)

/* Select third menu item */
    menuitem3 = pf15 (3)

/* Select fourth menu item */
    menuitem4 = pf9 (4)

/* Select fifth menu item */
    menuitem5 = pf10 (5)

/* Select sixth menu item */
    menuitem6 = pf11 (6)

/* Select seventh menu item */
    menuitem7 = pf5 (7)

/* Select eighth menu item */
    menuitem8 = pf6 (8)

/* Select ninth menu item */
    menuitem9 = pf7 (9)

/* Move cursor to next field */
/* defined to controlI in frs.map */

/* Move cursor to previous field */
/* defined to controlP in frs.map */
```

```
/* Move up one word within field */
    nextword = controlU (^U)

/* Move back one word within field */
/* defined to controlR in frs.map */

/* Switch between insert and overstrike */
/* mode defined to controlE in frs.map */

/* Redraw the screen defined to controlW */ /* in frs.map */

/* Delete the character under the cursor */
/* defined to controlD in frs.map */

/* Delete character immediately to left */
/* of cursor--defined to controlDEL */
/* in frs.map */

/* Start default text editor on field */
/* defined to controlV in frs.map */

/* Move to first column of next row */
/* in table field defined to controlN */
/* in frs.map */
/* Clear out the field */
      clear = controlX (^X)

/* Move to nextitem in form. If on */
/* regular field, move to next field. */
/* If in table field, move to next column */
/* if NOT in last accessible column.*/
/* Move to next row if in last accessible */
/* column of table field. */
    nextitem = controlM (Return)

/* Auto-duplicate value while in */
/* fill mode defined to controlA */
/* in frs.map */
```

E.5.2.2 VT220 Terminals

The default files for VT220 terminals are located in:

```
$II_SYSTEM/sql/vt2imap.unx

$II_SYSTEM/sql/vt220map.unx
```

The VT220 mapping file vt2imap.unx, which is the same as vt220map.unx with the addition of the **Nextitem** command, contains these statements:

```
/* Menu Key */
    menu = pf1 (PF1)

/* Help facility */
    frskey1 = pf15 (Help)

/* Quit from program */
    frskey2 = pf4 (PF4)

/* End current screen and */
/* return to previous screen */
    frskey3 = pf3 (PF3)
```

```
/* Go or execute function */
    frskey4 = pf16 (Do)

/* Put cursor on top of form or */
/* table field */
    frskey5 = controlK (^K)

/* Put cursor on bottom of form or */
/* table field */
    frskey6 = controlJ (^J)

/* Find next occurrence of string */
/* in this column of table field */
    frskey7 = pf21 (Find)

/* Save function */
    frskey8 = pf10 (PF10)

/* Undo and forget */
    frskey9 = pf2 (PF2)

/* Scroll page or form left */
    scrollleft = controlL (^L)

/* Scroll page or form right */
    scrollright = controlH (^H)

/* Previous screen or set of rows */
/* in table field */
    scrolldown = pf25 (Prev Screen)

/* Next screen or set of rows */
/* in table field */
    scrollup = pf26 (Next Screen)

/* Print contents of current screen */
/* to file or printer */
    printscreen = pf8 (PF8)

/* Select first menu item */
    menuitem1 = pf11 (PF11)

/* Select second menu item */
    menuitem2 = pf12 (PF12)

/* Select third menu item */
    menuitem3 = pf13 (PF13)

/* Select fourth menu item */
    menuitem4 = pf14 (PF14)

/* Select fifth menu item */
    menuitem5 = pf17 (PF17)

/* Select sixth menu item */
    menuitem6 = pf18 (PF18)

/* Select seventh menu item */
    menuitem7 = pf19 (PF19)

/* Select eighth menu item */
    menuitem8 = pf20 (PF20)
```

```
/* Remove character under cursor */
    deletechar = pf23 (Remove)

/* Switch between insert and overstrike */
    mode = pf22 (Insert Here)

/* Move cursor to next field defined */
/* to controlI in frs.map */

/* Move cursor to previous field */
/* defined to controlP in frs.map */

/* Move up one word within field */
    nextword = controlU (^U)

/* Move back one word within field */
/* defined to controlR in frs.map */

/* Redraw the screen defined to */
/* controlW in frs.map */

/* Delete character immediately to left */
/* of cursor defined to controlDEL */

/* in frs.map */

/* Start default text editor on field */
/* defined to controlV in frs.map */

/* Move to first column of next row in */
/* table field defined to controlN */
/* in frs.map */
/* Clear the field */
      clear = controlX (^X)

/* Move to nextitem in form. If on */
/* regular field, move to next field. */
/* If in table field, move to next column */
/* if NOT in last accessible column.*/
/* Move to next row if in last accessible */
/* column of table field. */
    nextitem = controlM (Return)

/* Auto-duplicate value while in fill */
/* mode defined to controlA in frs.map */
```

E.5.3 User-Level Mapping

Highest in precedence are the individual user's mappings.

To make the user-level mapping file known to the FRS, the user must execute a command at the operating system level. The syntax for the command is:

For the C shell:

setenv INGRES_KEYS *full-pathname/file_name*

For the Bourne shell:

INGRES_KEYS=*full-pathname/file_name*
export INGRES_KEYS

where *full-pathname/file_name* is the full pathname and filename for the mapping file. To eliminate the need to invoke this command for each terminal session, you can include this command in the file .login (C shell) or .profile (Bourne shell).

Of the three levels, user-level mapping is probably the least frequently used; the combination of mappings at the other two levels suffices for most users.

E.6 Obtaining Information on Mappings

Within the ULTRIX/SQL forms-based system, you can invoke the **Help** menu item to find out the current settings for function and control keys. See Appendix C for more information on this.

E.7 FRS Command Defaults

ULTRIX/SQL provides several default mapping files. The first is an installation-level mapping file, valid for all terminal types. The other two are default mapping files for the VT100 and VT220 terminals. These mapping files assign default control or function keys to the FRS commands. The installation file assigns control keys to most of the FRS commands. The terminal-type files expand and, in certain instances, override the installation mappings.

The following table lists the FRS commands and their default assignments. If you define your terminal as vt100i, the defaults for the VT100 terminal pertain to you. If your terminal is defined as vt220i, the defaults for the VT220 terminal pertain to you. Check the files directory for other mapping files.

Note

> The ULTRIX/SQL System Administrator has the ability to modify the default mapping files provided with ULTRIX/SQL. If the files have been modified, the mappings for the FRS commands may have been altered. In addition, the System Administrator can create mapping files for other terminal types besides vt100i and vt220i. These terminal-type files would then override the installation-level file.

Consult your ULTRIX/SQL System Administrator to determine whether the defaults listed in the table are valid for your installation and terminal type. In addition, mapping files for VT100 and VT220 terminals have been optimized so functions designated in the frs.map file are not remapped. Be careful when you modify the frs.map file to ensure the reliability of the mapping files for VT100 and VT220 terminals.

Table E-4 Default Settings for FRS Commands

FRS Command	Installation	VT100i	VT220i
Menu	Escape	PF1	PF1
Nextfield	Tab	Tab	Tab

FRS Command	Installation	VT100i	VT220i
Previousfield	Control-P	Control-P	Control-P
Nextword	Control-B	Control-U	Control-U
Previousword	Control-R	Control-R	Control-R
Mode	Control-E	Control-E	Insert Here
Redraw	Control-W	Control-W	Control-W
Deletechar	Control-D	Control-D	Remove
Rubout	Delete	Delete	Delete
Editor	Control-V	Control-V	Control-V
Leftchar	Control-H	left_arrow	left_arrow
Rightchar	Control-L	right_arrow	right_arrow
Downline	Control-J	down_arrow	down_arrow
Upline	Control-K	up_arrow	up_arrow
Newrow	Control-N	Control-N	Control-N
Clear	Control-X	Control-B	Control-B
Clearrest	Return	N/A	N/A
Scrollup	Control-F	PF12	Next Scr
Scrolldown	Control-G	PF8	Prev Scr
Scrollleft	Control-O	Control-L	Control-L
Scrollright	Control-U	Control-H	Control-H
Duplicate	Control-A	Control-A	Control-A
Printscreen	(no default)	Control-G	PF8
Nextitem	(no default)	Return	Return

E.8 Mapping Restrictions and Troubleshooting

This section describes some of the problems you may encounter when defining function and control keys, and provides general hints about how to solve them. Please note, however, these are only general guidelines; this section is by no means comprehensive.

E.8.1 Restrictions and Limitations

When defining function and control keys, note the following restrictions and limitations of the forms run-time system (FRS) and of your hardware.

- The FRS has the following internal limitations: 40 function keys (**PF***n* or **F***n*), 40 FRS keys, and 25 menu items.
- The FRS imposes no restrictions on which function and control keys can be mapped or remapped; however, certain control keys may be captured by the operating system before they reach the FRS (for example, **Control-Q** and **Control-S**).
- FRS commands cannot be mapped to an FRS key. This is syntactically illegal because FRS commands and FRS keys both appear on the left side of the equals sign in the mapping statement.
- TERM_INGRES cannot be reset dynamically by the application once the forms run-time system has been initialized.
- Positional menu item mapping cannot be turned off. On the vt100i termcap description, the application key pad is assigned by default to positional menu items. "1" on the key pad is assigned to the first menu item. You can see this if you look at the mapping file, for example:

```
menuitem1 = pf13
```

- Currently, map files are the only way to turn off a function or control key (for example, controlV = off). Keys cannot be turned off by the embedded SQL set command.
- Review of keyboard definitions for the VT* series:

TERM_INGRES	Top Row Keys	Alternate Keypad	Menu key
vt100	disabled	disabled	**Escape**
vt100k	disabled	disabled	**PF1**
vt100nk	disabled	numerics	**PF1**
vt100i	disabled	Functions (PFn)	**PF1**
vt200i	Functions (Fn)	numerics	**PF1**
vt220	Functions (Fn)	numerics	**PF1**

- **Escape** is considered reserved for vt100, vt220, and all other terminals that use **Escape** sequences to define function keys.

E.8.2 Troubleshooting Checklist

1. Have you checked the contents of the map error files in the working directory (**ingkey.err** or **app_ingkey.err**)?

   The error messages written to these files indicate map file problems. For example, if the same key is referenced twice, a warning error message will be written to this file. If this file is empty, an error occurred before the map file was parsed, and an error message was sent to the terminal screen.

2. Are the terminal's physical setup or emulation characteristics compatible with the current TERM_INGRES and key map definitions?

   For example, you will have a problem if a VT220 terminal is setup as a VT220 but TERM_INGRES is set to vt100i, whose map file references PF keys).

3. Is the user's TERM_INGRES terminal type compatible with the active key definitions?

   If not, mapping may seem to be "broken" when an ULTRIX/SQL subsystem starts up. For example, if TERM_INGRES is set to vt100nk but INGRES_KEYS points to vt220ak.map, an error occurs.

4. Is the environment variable INGRES_KEYS set unintentionally?

   This often happens when applications are moved to a new machine where INGRES_KEYS is defined, unlike the previous environment.

5. Have any of the key map file path names become invalid?

   For example, when the file system was moved to a new device, the file pathnames referred to by INGRES_KEYS or **set mapfile** are now invalid or perhaps file permissions were changed and the FRS cannot open the map files.

6. Are lower-level key definitions showing through on the user's menu line?

   This is the result of the key map merging by the FRS. This most often occurs when you forget to remap or turn off an intended key in the application map file or do not realize that INGRES_KEYS is also pointing to a map fileKeyboard keys;Mapping!E.

How to Write ULTRIX/SQL Termcap Descriptions F

F.1 Overview

To use the ULTRIX/SQL forms system on a specific terminal, ULTRIX/SQL needs information about the terminal's characteristics. The following read-only file supplies this information:

```
$II_SYSTEM/sql/files/termcap
```

If you wish to write termcap descriptions for terminals not described in the standard ULTRIX/SQL termcap file or modify an existing termcap entry, you may do so by following the guidelines described in this appendix. See Appendix D for a list of terminals currently in the supported termcap file. You may use terminals that are on this list without having to do any of the special programming described here.

The ULTRIX/SQL termcap file is based closely on the standard ULTRIX termcap file. The ULTRIX/SQL termcap descriptions, however, contain extra commands for use with the ULTRIX/SQL forms-based utilities. These extra commands fall into three categories:

- Video attributes such as inverse video, blinking cursors and color
- Boxing characters
- Commands to set up ULTRIX/SQL function keys

Note

A correctly written ULTRIX termcap entry should allow the ULTRIX/SQL forms-based utilities to work with that terminal. It may not support all of the advanced features that the terminal provides, such as function keys and video attributes. See the section "Optional Termcap Entries for Advanced Features" for information about using those advanced features.

To use this chapter, you should have already read through the terminal manufacturer's user guide. This chapter describes only the format of the termcap description. You must learn what strings and numbers to put after each command from the programmer's guide for the terminal. You should also have the terminal in front of you so that you can check the operation of the terminal as you are working on the termcap description.

Note

ULTRIX/SQL users not familiar with programming or lacking general knowledge about terminals may find creating new terminal descriptions difficult. Terminal programmer's guides are difficult to decipher, so you may want to get help from an experienced programmer to create new termcap entries.

F.2 Writing the Description

You can set the II_TERMCAP_FILE environmental variable to point to a working copy of the termcap file. This allows you to edit a new termcap file in any directory that you wish, without interfering with other users or the distribution copy of the termcap file. Use the command appropriate to your system to use an alternate termcap file. The syntax is as follows:

For the C shell:

setenv II_TERMCAP_FILE = *full-pathname/file_name*

For the Bourne shell:

II_TERMCAP_FILE = *full-pathname/file_name*
export II_TERMCAP_FILE

If this variable is defined, any ULTRIX/SQL forms-based utility will start up with that file instead of the standard distribution file.

F.2.1 Preparing the Description

Consider the following sample description:

```
Q1|qxz100|dec fictitious terminal:\
     :co#132:li#25:\
     :am:bs:\
     :is=\E[0m:cm=\E|%2;%2:
```

This fictitious terminal is designed to illustrate some of the features of termcap. The name of this terminal is "qxz100." It has an abbreviated name "Q1" and a long name "qxz fictitious terminal." (In order to use this with the ULTRIX/SQL forms system you would set the TERM_INGRES environment variable to "qxz100.") The qxz100 screen is 132 columns wide and 25 lines high. It has automatic margins (**am**) and uses **Control-H** for the backspace character (**bs**). The description contains an initialization string (**is**) and a cursor positioning string (**cm**).

F.2.2 General Format

As shown in the previous example, the first line is the list of names. All names must be separated by vertical bars (|). There must be a colon between the last name and the first capability.

If the termcap description is more than one line (as it should be for clarity), then each line except the last must have a backslash (\ ) at the end of it to signify continuation. Capabilities, which may be presented in any order, must be separated by a colon (:).

The last line must have a colon (:) at the end of it to signify that it is the end of the description. You may place a tab at the beginning of each line after the first one for readability.

F.2.3 Special Characters

The following table describes special symbols used in the termcap description:

Table F-1 Special Symbols in the Termcap Description

<table>
<tr><th>Symbol</th><th>Function</th></tr>
<tr><td>:</td><td>Separates capabilities.</td></tr>
<tr><td>|</td><td>Separates names.</td></tr>
<tr><td>\</td><td>Indicates that the definition continues on the next line (when specified at the end of a line).</td></tr>
<tr><td>\E</td><td>Specifies the escape character.</td></tr>
<tr><td>100</td><td>Causes the terminal to pause for 100 milliseconds (when specified before a command).</td></tr>
<tr><td>#</td><td>Indicates that capability is a number which immediately follows.</td></tr>
<tr><td>=</td><td>Sets the capability to a string that follows.</td></tr>
<tr><td>^X</td><td>Stands for Control-X, where X is any appropriate letter. Thus, ^g is Control-G, ^h is Control-H, etc.</td></tr>
<tr><td>\n</td><td>Indicates the Newline character.</td></tr>
<tr><td>\r</td><td>Indicates the Return character.</td></tr>
<tr><td>\t</td><td>Indicates the Tab character.</td></tr>
<tr><td>\b</td><td>Indicates the Backspace character.</td></tr>
<tr><td>\f</td><td>Indicates the Formfeed character.</td></tr>
<tr><td>\072</td><td>Indicates the colon character. (The octal value for ":" is 072.) In general, any character can be specified as the three-digit octal value of the ASCII character by preceding it with a backslash.</td></tr>
<tr><td>@</td><td>When placed after a command, means "do not apply this command." It is used in descriptions that have the tc command. (Refer to the section "The Eleven Basic Commands" later in this appendix for information on the tc command.)</td></tr>
</table>

F.2.4 Names

Names have a special format that you must follow. The first name must be two letters long. The second name is the common name to which you will set TERM_INGRES. The last name may be a concise description of the terminal's brand and model number. The last name may contain blanks, though the other names cannot have blanks.

All names must be separated by a vertical bar (|). Additional names may be placed between the second name and the last name. The additional names may be used as alternative names for TERM_INGRES.

Note

Names must always be checked for uniqueness. You should check through the termcap file before writing a new description to make sure that the names you wish to use have not already been selected. A duplicated name will not be recognized; only the first one will be used.

F.2.5 Capabilities

Capabilities are designated by commands, which must be separated by colons. All commands are two letters long. String and numeric commands are followed by additional information that is read by the ULTRIX/SQL forms system. The three types of commands are:

- String
- Numeric
- Boolean

Strings contain sequences of characters. The command must be followed by an equal sign (=). For example, "up=\EA" says that the command **up** (which stands for the sequence to move the cursor up) is set to the sequence **Escape-A**.

Some string commands have to be preceded by a time delay, which is referred to as *padding*. Padding is required in situations where the terminal may be reading characters at a slower rate than they are being transmitted. Padding ensures that the terminal has time to execute commands, such as moving the cursor, without losing characters.

The two types of padding are *nonproportional* (or *straight*) padding, and *proportional* padding (to the number of lines affected). To specify straight padding, put the time (in milliseconds) of delay needed before the command. To specify proportional padding, place an asterisk (*) after the amount of time. For example, on the "concept 100" terminal, the **ta** command (tab character) takes a straight time delay of 8 milliseconds and the **cd** command (clear display) takes a proportional delay of 16 milliseconds. The termcap entries look like this:

Straight padding:

```
ta=8\t
```

Proportional padding:

```
cd=16*\EarC
```

Numeric commands are followed by a number sign (#). For example, "co#80" says that there are 80 columns on the screen. (**co** is the command which specifies the number of columns on the screen.)

Boolean commands signify the existence of a capability by their presence. They are not followed by any sequence or other symbols.

F.2.6 Suggested Approach to Getting Started

To create a termcap definition, start by reading through the technical manual on the terminal to find the information for the 11 basic capabilities described under "The Eleven Basic Commands." When you have included these 11 capabilities in the description, try the terminal to see if it works. Once you get it working, you can try adding additional features listed under "Optional Termcap Entries for Advanced Features."

If you have problems, first check to make sure that you entered the sequences from the manual correctly and that the format of the termcap entry is correct. If it still does not work, check to see if there are some additional capabilities that need to be added to make it work. Also, certain terminals require special initialization commands. Check the technical manual to see which additional sequences you should add to the initialization string.

One excellent way of preparing termcap descriptions is to examine the termcap entries for similar terminals. If you are trying to write a description for a terminal that is similar to one in termcap, you can use the **tc** command to indicate that all attributes for the new terminal are to be taken from the description of a terminal already in termcap. Then, you only need to specify the few differences.

Alternatively, you can manually copy the capabilities from a similar terminal and see if it works. Most terminals conform to a system of specifying escape sequences that is ANSI standard. Thus, if you have an ANSI standard terminal, you should be able to get about 90 percent of the capabilities by copying them from another ANSI standard terminal. The VT100 is an example of an ANSI terminal that has capabilities similar to many different terminals. For this reason, this document contains numerous references to the VT100 sequences in its examples.

Finally, if your terminal has a VT100 emulation mode, you can save time by using VT100 emulation mode and using the VT100 termcap description. In most cases it will not be necessary to make a termcap description. In other cases, the terminal will work with a termcap description that is identical to the VT100 except that it contains the VT100 emulation sequence in its initialization string. If your terminal has VT100 emulation mode, try it, as a VT100 supports the most advanced features of the ULTRIX/SQL forms system.

F.3 The Eleven Basic Commands

There are 11 commands that all terminals must have in order to work properly with ULTRIX/SQL. Termcap descriptions that have only these 11 basic descriptions usually work, although they lack extra features such as function keys and video attributes. These 11 descriptions form the core of the termcap description.

Table F-2 The Eleven Basic Commands

Command	Description
co	Specifies the number of columns on the screen. Without this command ULTRIX/SQL does not know how wide to make a form. This command is numeric and should always be followed by a number. VT100 Example: **co#80**
li	Specifies the number of rows down the screen. This command is numeric and should always be followed by a number. VT100 Example: **li#24**
bs	Indicates that this terminal can backspace using **Control-H.** This is a boolean command. Include it if your terminal can backspace using **Control-H**. VT100 terminals use **Control-H** for backspacing.
bc	Indicates that the terminal does not use **Control-H** for backspacing. Note that you cannot use **bc** and **bs** together.
cd	Clears everything from the cursor to the end of the display. VT100 Example: **cd=\E[J**
ce	Clears everything from the cursor to the end of the line. VT100 Example: **ce=\E[K**
cl	Clears the entire screen. VT100 Example: **cl=\E[;H\E[2J**
cm	Sends the cursor motion string (called the "cursor position string" in some manuals) to the terminal when ULTRIX/SQL needs to move the cursor from one location to another. As such, the string must accept two parameters: an x-coordinate and a y-coordinate, whose values are obtained by counting the number of rows/spaces from the top-left corner of the screen. Because these values must be sent along with the string at run time, special place markers must be left in the string to tell the forms system where to place the x and y coordinates. To implement this, find the cursor-addressing scheme described in the manual for your terminal. Then substitute the special place marker characters in the spot where numbers are expected. Also, be sure to include any special modifiers in the description if they are needed. For detailed information on cursor place markers and special modifiers, refer to the discussion of the cursor motion command which follows this table.
nd	Indicates a nondestructive space. This string specifies the command for moving the cursor right one space without overwriting the contents of the screen at that point. VT100 Example: **nd=\E[C**

Command	Description
is	Specifies a terminal initialization string. This string includes any sequences needed to set up the terminal prior to running an ULTRIX/SQL forms program. VT100 Example: **is=\E>\E[?3l\E[?4l\E[?7l\E[?8h** The preceding terminal initialization string does five things: Puts the terminal in keypad numeric mode **\E>** Puts terminal into 80-column mode **\E[?3l** Puts terminal into jump mode **\E[?4l** Turns wraparound off **\E[?7l** Turns on autorepeat **\E[?8h** The **is** command is not always needed, but for most terminals it is necessary for tailoring the setup to your needs.
tc	Allows you to use all the capabilities listed for another terminal without rewriting them. This command is actually optional, but is so useful that it has been listed as one of the basic commands. This capability must always be the *last* capability in the description. This rule, which exists so that duplicated commands can be unambiguously defined, is the only exception to the general rule that commands can be presented in any order. **(tek4115):** **dk|tk4115|tek-4115|tektronix4115:** **:ld@:tc=vt100f:** In this example, the "tek4115" is given all the commands from the "vt100f" description, except the **ld** command to initialize the boxing characters.

The following table lists cursor place markers and cursor-addressing options for the **cm** command:

Table F-3 Place Markers

Place Marker	Description
%d	Place marker for a decimal integer that prints out as many digits as necessary without leading zeros.
%2	Place marker for a decimal integer of two places.
%3	Place marker for a decimal integer of three places.
%.	Place marker for a binary value character.
%+*N*	Place marker for a binary value character, with the value of the character *N* added to it.
%%	Place marker that gives a single %.

Table F-4 Common Modifiers

Modifier	Description
%i	Increments line/column. Include this in your **cm** string if the terminal uses "1" as the coordinate origin.
%r	Reverses the usual order of the *x* and *y* cursor position coordinates. Usually the terminal cursor positioning string expects the column to be substituted for the first coordinate place marker. If your terminal cursor positioning string expects the row first, include the characters %r before the first place marker in the cursor motion sequence.

Table F-5 Modifiers Needed Only for Special Terminals

Modifier	Description
%*xy*	If the value for the place marker is greater than *x*, add *y* to the value before generating the output string.
%B	BCD (16*(x/10)) + (x mod 10) : the value of the parameters are transformed according to this formula.
%n	Do an exclusive OR on the row and column values with the octal value 0140 before generating the string for cursor motion. (This is used only for the Datamedia 2500 terminal.)
%D	Reverse coding (x-2*(x mod 16)): the values of the parameters are transformed according to this formula. (This is used only for Delta Data terminals.)

The following table and examples show the cursor motion string, usage, and termcap entry for four different terminals.

Table F-6 Terminals and Termcap Descriptions

<table>
<tr><th>Terminal</th><th>Entry Listed in Manual</th><th>Example Usage on Terminal Mode</th><th>Example of the Termcap Description</th></tr>
<tr><td>qxz100</td><td>ESC|x;y</td><td>ESC|07;16</td><td>cm=\E|%2;%2</td></tr>
<tr><td>vt100</td><td>ESC[x;yH</td><td>ESC[08;17H</td><td>cm=5\E[%i%2;%2H</td></tr>
<tr><td>dm3045</td><td>ESCYyx</td><td>ESCY2*</td><td>cm=\EY%r%+ %+</td></tr>
<tr><td>delta</td><td>Ctrl-Oxy</td><td>Ctrl-ORS</td><td>cm=^O%D%+9%
D%+9</td></tr>
</table>

Example 1: (qxz100–fictitious terminal)

The cursor motion string listed in the user's manual for the qxz100 (a fictitious terminal) is **ESC|*x*;*y*** where *x* and *y* are two-digit integers specifying the column and the row, respectively. The qxz100 is an example of a terminal that uses a (0,0) origin, so *x* and *y* must be one less than the whole number that represents the position on the screen. Thus, to move the cursor to column 8 row 17 on the qxz100, you would enter ESC|07;16. The termcap entry is:

```
cm=\E|%2;%2
```

where **\E** maps to **ESC** and **%2** is the place marker that maps to a decimal integer of two places.

Example 2: (vt100)

The cursor motion string for the VT100 is **ESC[*x*;*y*H** where *x* and *y* are two-digit integers specifying the column and row, respectively. The VT100, as opposed to Example 1 above, positions the cursor relative to a origin of (1,1). Thus, to move the cursor to column 8 row 17 on the VT100, you would enter ESC[08;17H. The termcap entry is:

```
cm=\E[%i%2;%2H
```

where **\E** maps to **ESC**, **%2** maps to a decimal integer of two places, and the **%i** signifies that the VT100 uses a (1,1) origin. The default setting for the **cm** string is for a (0,0) origin. If your terminal uses a (1,1), origin you must explicitly state that by placing a **%i** somewhere inside the **cm** string.

Example 3: (Datamedia 3045)

The cursor motion string for Datamedia 3045 is **ESCY*yx*;** where *y* and *x* are characters whose binary values are offset by 20 hex. (Note that for this terminal the row must be given before the column.)

To move the cursor to the position (19,11) on this terminal, you must include the sequence **ESCY2***. The **ESCY** is the first part of the sequence. The "2" is the ASCII character with hexadecimal value 32, which is the same as 12 hex plus the 20 hex offset. Note that 12 hex corresponds to column 19 on the screen. The "*" is the ASCII character with a hexadecimal value of 2A, which equals 0A hex plus the 20 hex offset. Again, note that 0A hex corresponds to row 11 on the screen.

The **cm** string for this terminal is **cm=\EY%r%+ %+** , where **\E** maps to **ESC**, **%r** is a modifier that tells the forms system that the row and column parameters are reversed, and **%+** is the place marker for a character offset by a blank (which has the ASCII value of 20 hex).

Example 4: (Delta Data 5000)

The cursor motion string for the Delta Data 5000 is **Ctrl-O*xy***, where *x* and *y* are characters whose binary values must be offset by 3A hex, and converted according to the reverse coding formula:

```
(x-2*(x mod 16))
```

The **cm** string for this terminal is **cm=^O%D%+9%D%+9**, where **^O** stands for **Ctrl-O**, **%D** indicates that the parameters must be transformed according to the reverse coding formula, and **%+9** is the place marker for a character offset by 3A hex (ASCII character "9").

F.4 Optional Termcap Entries for Advanced Features

These are features that will improve the appearance of screen displays and make the terminal easier to use, but are not essential for the basic functions of ULTRIX/SQL.

F.4.1 Commands Used to Program Video Attributes

The four basic modes are: underscore, blinking, reverse video, and high intensity. All the commands below are combinations of the four basic modes.

Table F-7 Video Attribute Commands

Command	Description	VT100 Example
rv	Turns on reverse video.	rv=1\E[7m
bl	Turns on blinking mode.	bl=1\E[5m
bo	Turns on high intensity mode.	bo=1\E[1m
us	Turns on underscore mode.	us=2\E[4m
ea	Turns off all special display characteristics.	ea=1\E[m
za	Turns on reverse video, blinking, high intensity, and underscore modes.	za=1\E[1;4;5;7m
zb	Turns on high intensity and underscore modes.	zb=1\E[1;4m
zc	Turns on high intensity and blinking modes.	zc=1\E[1;5m
zd	Turns on high intensity and reverse video modes.	zd=1\E[1;7m
ze	Turns on underscore and blinking modes.	ze=1\E[4;5m
zf	Turns on underscore and reverse video modes.	zf=1\E[4;7m
zg	Turns on blinking and reverse video modes.	zg=1\E[5;7m
zh	Turns on high intensity, blinking, and underscore modes.	zh=1\E[1;4;5m

Command	Description	VT100 Example
zi	Turns on blinking, underscore, and reverse video modes.	zi=1\E[4;5;7m
zj	Turns on high intensity, blinking, and reverse video modes.	zj=1\E[1;5;7m
zk	Turns on high intensity, underscore, and reverse video modes.	zk=1\E[1;4;7m

F.4.2 Commands Needed for Boxing Characters

Not all terminals have special boxing characters. If your terminal does not have them, ULTRIX/SQL uses dashes (—) and vertical bars (|) instead. Using boxing characters, however, greatly improves the appearance of forms that display table column values and rows.

Table F-8 Commands for Boxing Characters

Command	Description	VT100 Example
ld	Initializes terminal to draw solid lines.	**ld=\E)0**
ls	Interprets subsequent characters for drawing solid lines.	**ls=\016**
le	Interprets subsequent characters as regular characters.	**le=\017**
qa through qk	Indicates the boxing characters. See the table below.	

The next table describes the boxing characters **qa** through **qk** mentioned above.

Table F-9 Boxing Characters

Command	Description	VT100 Example
qa	Lower right corner of a box	qa=j
qb	Upper right corner of a box	qb=k
qc	Upper left corner of a box	qc=l
qd	Lower left corner of a box	qd=m
qe	Crossing lines	qe=n
qf	Horizontal line	qf=q
qg	Left T (stem points right)	qg=t

Command	Description	VT100 Example
qh	Right T (stem points left)	qh=u
qi	Bottom T (upside down T)	qi=v
qj	Top T (right side up T)	qj=w
qk	Vertical line	qk=x

F.4.3 Commands Needed for Function Keys

To activate function keys, the termcap file uses the following commands:

Table F-10 Commands for Function Keys

Command	Description
ke	Takes the terminal out of "keypad transmit" mode.
ks	Puts the terminal in "keypad transmit" mode.
kn	Specifies the number of function keys available. For example, "kn#18" indicates that you can use 18 functions keys on the VT100. You can have a maximum value of 40 for the number of function keys.
ky	Indicates that the terminal has cursor and function keys. This is a boolean command. It must be present if you wish to use function keys. Using it disables the **Escape** key and sets the first function key **PF1** to the menu function.
k0 through **KD**	Specifies strings sent by function keys.

The following table describes the commands that you can set to send strings by using function keys.

Table F-11 Function Key Commands for Sending Strings

Command	Function Key Number	VT100 Example
k0	function key 1	k0=\EOP
k1	function key 2	k1=\EOQ
k2	function key 3	k2=\EOR
k3	function key 4	k3=\EOS
k4	function key 5	k4=\EOw
k5	function key 6	k5=\EOx

Command	Function Key Number	VT100 Example
k6	function key 7	k6=\EOy
k7	function key 8	k7=\EOm
k8	function key 9	k8=\EOt
k9	function key 10	k9=\EOu
kA	function key 11	kA=\EOv
kB	function key 12	kB=\EOl
kC	function key 13	kC=\EOq
kD	function key 14	kD=\EOr
kE	function key 15	kE=\EOs
kF	function key 16	kF=\EOp
kG	function key 17	kG=\EOn
kH	function key 18	kH=\EOM
kI	function key 19	none
kJ	function key 20	none
kK	function key 21	none
kL	function key 22	none
kM	function key 23	none
kN	function key 24	none
kO	function key 25	none
kP	function key 26	none
kQ	function key 27	none
kR	function key 28	none
kS	function key 29	none
kT	function key 30	none
kU	function key 31	none
kV	function key 32	none
kW	function key 33	none
kX	function key 34	none
kY	function key 35	none

Command	Function Key Number	VT100 Example
kZ	function key 36	none
KA	function key 37	none
KB	function key 38	none
KC	function key 39	none
KD	function key 40	none

F.5 Commands Needed for Arrow Keys

The arrow key commands are all strings.

Table F-12 Commands for Arrow Keys

Command	Description	VT100 Example
ku	Sent by the terminal up arrow key	ku=\EOA
kd	Sent by the terminal down arrow key	kd=\EOB
kr	Sent by the terminal right arrow key	kr=\EOC
kl	Sent by the terminal left arrow key	kl=\EOD

F.5.1 Commands Used for Color

You can use these commands to turn on color in terminals that support color. These are mapped to the color codes, from 0 to 7, as defined for your terminal. All the commands are strings and may have optional padding. The commands are described in the following table.

Table F-13 Commands for Color

Command	Description	Envision Example
ya	Default foreground color (0)	2\Ea7
yb	Alternate foreground color #1	2\Ea1
yc	Alternate foreground color #2	2\Ea2
yd	Alternate foreground color #3	2\Ea3
ye	Alternate foreground color #4	2\Ea4
yf	Alternate foreground color #5	2\Ea5

Command	Description	Envision Example
yg	Alternate foreground color #6	2\Ea6
yh	Alternate foreground color #7	2\Ea7

Envision Example:

```
E3|envisionc|envision230|this has the color definitions:\
     :ya=2\Ea7:yb=2\Ea1:yc=2\Ea2:yd=2\Ea3:\
     :ye=2\Ea4:\:yf=2\Ea5:yg=2\Ea6:\
     :yh=2\Ea7:tc=vt100k:
```

F.5.2 Command to Specify FRS Mapping File for Terminal

You can use the **mf** command to specify a default forms run-time system (FRS) key mapping file for a terminal. You must include the name of a file in the ULTRIX/SQL files directory without an extension (*not* the full directory specification or path name of the file). This file should contain the default FRS key mapping for the terminal.

vt100i Example:

```
:mf=vt1imap:
```

F.5.3 Commands to Optimize Cursor Movement

These commands generally improve the way the ULTRIX/SQL forms system moves the cursor around the form. They are usually optional.

Table F-14 Commands to Optimize Cursor Movement

Command	Description
am	Automatic margins. This boolean command is important on forms that run to the edge of the screen.
cs	Change scrolling region. This command improves the appearance of the cursor movements when scrolling on a long form. The forms system will still work if this is not defined; it just may not look as nice. This command is very similar in form to the **cm** command; however, the **cs** command's parameters are the upper and lower limits of scrolling instead of the position on the screen. Otherwise, all the place markers and modifiers are the same. If you use the **cs** command, you *must* also include the **sr** command.
do	Down one line. Inclusion of this command helps the forms system move the cursor faster.
sr	Scroll reverse. This command makes the form scroll backwards instead of jumping if you are moving up on a long form.

vt100 Example:

```
cs=5\E[%2;%2r
```

F.6 Commands for Special Situations

As mentioned in the introduction, the ULTRIX/SQL termcap is based upon the ULTRIX termcap file. Below is a list of ULTRIX termcap entries that are included in the ULTRIX/SQL termcap but are usually not needed. For additional information, refer to the ULTRIX termcap(5) reference page.

F.6.1 Commands from the ULTRIX Termcap File

Table F-15 ULTRIX Termcap File Commands

Name	Type	Description
bt	str	Back **Tab**. Padding may be required on this command.
ho	str	Sequence to move the cursor to the home position. This command should be used if and only if the terminal does not possess a cursor positioning string (**cm**).
ll	str	Last line, first column (if no **cm**).
ms	bool	Safe to move while in standout and underline mode.
pc	str	Pad character (rather than null).
sf	str	Scroll forward. Padding may be required on this command.
ta	str	**Tab**, other than **Control-I** or with padding. Padding may be required on this command.
te	str	String to end programs that use **cm**.
ti	str	String to begin programs that use **cm**.
ve	str	Sequence to end open/visual mode.
vs	str	Sequence to start open/visual mode.

F.6.2 Commands for Specific Terminals

Table F-16 Commands for Specific Terminals

Name	Type	Description
hz	str	Hazeltine; cannot print apostrophes.
nc	bool	No correctly working carriage return (DM2500

Name	Type	Description
xb	bool	Beehive (f1=ESC, f2=Ctrl–C)
xn	bool	A newline is ignored after a wrap (Concept).
xr	bool	Return acts like ce \r \n (Delta Data).
xs	bool	Standout not erased by writing over it (HP 2640 series).
xt	bool	Tabs are destructive, magic so character (Teleray 1061).

F.7 Examples of Termcap Descriptions

This section includes several examples of termcap entries. They illustrate the format of ULTRIX/SQL termcap entries. The example includes commentary on each of the descriptions. If you want to learn more about the termcap process, compare these descriptions with manuals for the particular terminals.

F.7.1 VT100 (All-Inclusive)

This description contains all the features described above. The example is longer than most termcap descriptions.

```
d7|vt100k|vt-100k|pt100k|vt100 with everything:\
     :co#80:li#24:cl=20\E[;H\E[2J:bs:cm=5\E[%i%2;%2H:\
     :nd=2\E[C:\
     :up=2\E[A:ce=3\E[K;cd=50\E[J:us=2\E[4m:ue=2\E[m:\
     :is=\E>\E[?3l\E[?4l\E[?7l\E[?8h:ks=\E[?1h\E=:\
     :ke=\E[?1l\E>:ku=\EOA:kd=\EOB:kr=\EOC:kl=\EOD:\
     :ld=\E)0:\
     :qa=j:qb=k:qc=l:qd=m:qe=n:qf=q:qg=t:qh=u:qi=v:\
     :qj=w:qk=x:\
     :ls=\016:le=\017:\
     :cs=5\E[%2;%2r:bl=1\E[5m:be=1\E[m:\
     :bo=1\E[1m:eb=1\E[m:rv=1\E[7m:re=1\E[m:ea=1\E[m:\
     :za=1\E[1;4;5;7m:zb=1\E[1;4m:zc=1\E[1;5m:\
     :zd=1\E[1;7m:\
     :ze=1\E[4;5m:zf=1\E[4;7m:zg=1\E[5;7m:\
     :zh=1\E[1;4;5m:\
     :zi=1\E[4;5;7m:zj=1\E[1;5;7m:zk=1\E[1;4;7m:\
     :kh=\E[H:ky:k0=\EOP:k1=\EOQ:k2=\EOR:k3=\EOS:pt:\
     :sr=5\EM:\
     :k4=\EOw:k5=\EOx:k6=\EOy:k7=\EOm:\
     :k8=\EOt:k9=\EOu:kA=\EOv:\
     :kB=\EOl:kC=\EOq:kD=\EOr:kE=\EOs:kF=\EOp:\
     :kG=\EOn:kH=\EOM:\
     :kn#18:mf=vt1imap:
```

F.7.2 VT100 (Simple)

Here is the VT100 using only basic features. This description lacks many of the niceties found in the longer description above, but it illustrates that minimal descriptions that provide basic functioning with ULTRIX/SQL are not too hard to write.

```
d8|vt100s|simple vt100 entry:\
     :co#80:li#24:cl=20\E[;H\E[2J:bs:\
     :cm=5\E[%i%2;%2H:nd=2\E[C:\
     :is=\E>\E[?3l\E[?4l\E[?7l\E[?8h:\
     :up=2\E[A:ce=3\E[K:cd=50\E[J:
```

F.7.3 Envision 230

This termcap description illustrates the use of the **tc** command. This description contains all the features of the VT100 except that it does not employ the VT100 initialization string. Also note the large number of names; this example covers three different varieties of Envision terminal. If you need to write descriptions for terminals similar to known terminals, you may find this example particularly pertinent.

```
E1|envision|envision230|envision220:\
     :is@:tc=vt100k:
```

The ULTRIX/SQL Standard Catalog Interface G

G.1 Introduction

This appendix describes the Standard Catalog Interface views and lists the System Catalogs for the database management system (DBMS System Catalogs).

The Standard Catalog Interface is implemented as a group of views defined on the System Catalogs. These views are the supported catalogs. Users who need to query the System Catalogs should use these views. System catalogs are tables, just like user tables in a database. Each system catalog has a distinct set of columns (attributes), each of which has a distinct database management function.

Note

> The information in this appendix about the DBMS System Catalogs is provided for the convenience of ULTRIX/SQL users. However, the base table catalogs are subject to change at any time. Therefore, user-defined programs, tools, or other user-defined interfaces to ULTRIX/SQL should access the System Catalogs only through the supported views (Standard Catalog Interface). Digital Equipment Corporation does not support any program, tool, or interface that uses the System Catalogs directly rather than through the Standard Catalog Interface.

The following conventions apply with respect to the columns in the system catalogs:

- All values are left justified in a column unless otherwise noted.
- All columns are uppercase unless otherwise noted.
- Columns are assumed to be non-nullable except where explicitly noted.

The column definitions in the following tables list all possible column values.

Many columns that are **char(32)** names are valid ULTRIX/SQL names. ULTRIX/SQL names are described in Chapter 1.

Allowable values for those columns described as ULTRIX/SQL usernames are determined by ULTRIX in general, but should be drawn from the list of values in the **iidbconstants** catalog, which contains the current *username* and current *dbname.*

All **char(25)** fields described as ULTRIX/SQL standard dates have the following format:

yyyy_mm_dd hh:mm:ss GMT

In the preceding syntax:

yyyy	is the year (for instance, 1987)
mm	is the month (for instance, 11)
dd	is the day of month (for instance, 21)
hh	is the military hour (for instance, 14)
mm	is the minute (for instance, 43)
ss	is the second (for instance, 32)
GMT	indicates Greenwich Mean Time

The underscores and colons are required between the parts of the date, and a space is required between *ss* and GMT.

G.2 Standard Catalog Interface

All database users can read the Standard Catalog Interface views, but the views may be updated only by a privileged ULTRIX/SQL user who specifies the **+u** flag when the database is accessed.

G.2.1 The iidbcapabilities Catalog

The **iidbcapabilities** table contains information about the capabilities the database management system (DBMS) provides. This is the only real table in the Standard Catalog Interface.

The following table describes the columns in the **iidbcapabilities** catalog:

Column Name	Data Type	Description
cap_capability	**char(32)**	Contains one of the values listed in the following table. If the cap_capability has a value, it will be activated by the value in the "cap_value" column.
cap_value	**char(32)**	Most capabilities are binary, and will be set to the string "Y" or "N," depending on whether or not the DBMS supports them. Some, however, have values. For these, this field contains the value of the capability.

The "cap_capability" column in the **iidbcapabilities** catalog contains one or more of the following values:

Capability	Value
DB_NAME_CASE	The type of case sensitivity the database has with respect to database objects. It takes on the value of "LOWER," "UPPER," or "MIXED." If not present, this capability defaults to "LOWER." Database objects may be specified in programs and queries in either mixed, lower, or upper case if the value is "LOWER" or "UPPER." If the value is "MIXED," be careful to preserve the case specified by the user for database objects. Database objects are stored in the system catalogs, as specified by DB_NAME_CASE. Database and user names are stored in upper case if the value of DB_NAME_CASE is "UPPER" or "MIXED." If the value is "LOWER," they are stored in lower case in the system catalogs.
INGRES	Set to "Y" if the DBMS supports, in all respects, 100% of the current ULTRIX/SQL release. Otherwise "N." Defaults to "Y."
INGRES/SQL_LEVEL	Version of ULTRIX/SQL support provided by the DBMS. Examples: 00602 DBMS supports ULTRIX/SQL version 1.0 (based on INGRES 6.2) 00000 DBMS does not support ULTRIX/SQL. Default is 00602.
SAVEPOINTS	For internal use by ULTRIX/SQL
DBMS_TYPE	The type of DBMS the application is communicating with. Valid values are "INGRES" and "Rdb." The default value is "INGRES."
PHYSICAL_SOURCE	Indicates whether the physical table description in **iitables** is correct or if **iiphysical_tables** must be checked for the correct physical table description. Due to base catalog normalization, it is possible for the physical description information in **iitables** to be defaulted, while the actual information is present in **iiphysical_tables**. Values for this column are either "T," which indicates that both **iitables** and **iiphysical_tables** contain the physical information, or "P," which indicates that the physical information is only in **iiphysical_tables**.

G.2.2 The iidbconstants Catalog

The **iidbconstants** view contains a list of values that must be known by the ULTRIX/SQL application.

The following table describes the columns in the **iidbconstants** catalog:

Column Name	Data Type	Description
user_name	**char(32)**	The name of the current user.
dbaname	**char(32)**	The name of the database's owner.

G.2.3 The iitables Catalog

The **iitables** view contains an entry for each queryable object in the database. In ULTRIX/SQL these objects are tables, views, and indexes. The **iitables** catalog contains basic system-independent logical information. User programs can query this catalog to find out what tables, views, and indexes exist in a database.

In ULTRIX/SQL, this view is keyed on **table_name** and **table_owner**, so the best way to query this view is with a query such as:

```
select   *
from     iitables
where    (table_name = (anyname))
and      (table_owner = (myname) or table_owner = (dbaname))
```

The following table describes the columns in the **iitables** catalog:

Column Name	Data Type	Description
table_name	**char(32)**	The object's name. This is an ULTRIX/SQL name.
table_owner	**char(32)**	The object's owner, expressed as an ULTRIX/SQL username. Generally the creator of the object.
create_date	**char(25)**	The object's creation date, expressed as an ULTRIX/SQL standard date. This will be blank if unknown.
alter_date	**char(25)**	The last time this table was altered, expressed as an ULTRIX/SQL standard date. The alter_date is the same as the create_date until the logical structure of the table is changed. The alter_date is updated whenever the logical structure of the table changes, either through changes to the columns in the table or changes in the primary key itself. Physical changes to the table, such as changes to data, indexes, or physical keys, do not change this date. This is blank if unknown.
table_type	**char(8)**	The type of the query object. The possible values are: "T" Object is a table "V" Object is a view "I" Object is an index Further information about tables can be found in **iiphysical_tables** and about views in **iiviews**.
table_subtype	**char(8)**	This describes the type of table or view that this is. Possible values are: "N" (native) for standard ULTRIX/SQL databases "I" (imported tables) for Remote Access to Rdb/VMS Blank (" ") if unknown.

Column Name	Data Type	Description
table_version	**char(8)**	This is the version of the object, which allows the application to determine where additional information about this particular object is stored. This reflects the database type, as well as the version of an object within a given database. For ULTRIX/SQL tables, the value for this field is "ING6.0."
system_use	**char(8)**	Specifies whether the object is a system object or a user object. The "system_use" field is used by the application in order to screen lists of tables in catalog displays. Values are "S" (for system) and "U" (if unknown). The distinction between "S" and "U" is used in utilities to know which tables need reloading. If the value is unknown, the utilities will use the naming convention of "ii" for tables in order to distinguish between system and user catalogs. Also, ULTRIX/SQL assumes any table beginning with **ii_** is a front end object, rather than a DBMS system object. The standard system catalogs themselves must be included in the **iitables** catalog and are considered system tables.

The following columns in **iitables** have values only if the table_type is "T" or "I." The columns are set to the default values, "-1" for numeric data types and a blank for character data types, if this information is not available through the Remote Access to Rdb/VMS interface and you are accessing an Rdb/VMS database.

This information may also be present in the **iiphysical_tables** catalog, whether or not it is present in the **iitables** catalog.

The columns are described as follows:

Column Name	Data Type	Description
table_stats	**char(8)**	"Y" if this object has entries in the **iistats** table, or "N" if this object does not have entries. Whether this is blank or not is *not* a determinant of "Y" or "N." If the field is blank, then a probe of the **iistats** table should be done in order to determine if entries exist there. This column is used only for optimization of ULTRIX/SQL databases.
table_indexes	**char(8)**	"Y" if this object has entries in the **iiindexes** table that refer to this as a base table, or "N" if this object does not have entries. Whether this is blank or not is *not* a determinant of "Y" or "N." If the field is blank, a probe of the **iiindexes** table on the **base_table** column should be done in order to determine if entries exist there. This field is used only for optimization of ULTRIX/SQL databases.

Column Name	Data Type	Description
is_readonly	**char(8)**	"N" if updates are physically allowed, or "Y" if no updates are allowed. This will be blank if it is unknown. This is used for tables which are defined to the Remote Access to Rdb/VMS only for retrieval. If this field is set to "Y," no updates will work, independent of what permissions might be set. If it is set to "N," updates may be allowed, depending on whether or not the permissions allow it.
num_rows	**integer**	The estimated number of rows in the table. The value is set to -1 if the number is unknown.
storage_structure	**char(16)**	The storage structure for the table. It is one of the following: "HEAP" If table is a heap structure "HASH" If table is a hash structure "ISAM" If table is an isam structure "BTREE" If table is a btree structure Blank (" ") If table structure is unknown
is_compressed	**char(8)**	Set to "Y" if the table is stored in compressed format, or "N" if the table is uncompressed. The field is blank if this information is unknown.
duplicate_rows	**char(8)**	"D" if the table, as created, allows duplicate rows or "U" if it does not. The table storage structure (as defined by the "unique_rule" column, which specifies unique or non-unique keys) can override this setting. This column is blank if this information is unknown.
unique_rule	**char(8)**	"U," "D," or a blank. If the value is "U" and the object is an ULTRIX/SQL object, it indicates that the object has a unique storage structure key(s). Refer to the "key_sequence" column of the **iicolumns** catalog for the key(s). If the value is "U" and the object is not an ULTRIX/SQL object, it indicates that the object has a unique key, described in either the **iicolumns** or **iialt_columns** catalogs. If the value is "D," it indicates that duplicate physical storage structure keys are allowed. (A unique alternate key may exist in the **iialt_columns** catalog, and any storage structure keys may be listed in the **iicolumns** catalog.)
number_pages	**integer**	The estimated number of physical pages in the table. This value is set to -1 if unknown.
overflow_pages	**integer**	The estimated number of overflow pages in the table. This value is set to -1 if unknown.
row_width	**integer**	The size, in bytes, of the uncompressed binary value for a row of this query object.

The following columns are used by the ULTRIX/SQL DBMS. If you are accessing an Rdb/VMS database and this information is not available through the Remote Access to Rdb/VMS interface, the columns are set to the default values, "-1" for numeric data types and a blank for character data types.

The information in this section is *not* contained in **iiphysical_tables**.

Column Name	Data Type	Description
expire_date	**integer**	Expiration date of table. This is an ULTRIX/SQL _bintime date.
modify_date	**char(25)**	The date on which the last physical modification to the storage structure of the table occurred. This is an ULTRIX/SQL standard date. This column will be blank if unknown or inapplicable.
location_name	**char(24)**	The first location of the table. If there are additional locations for a table, they will be shown in the **iimulti_locations** table, and multi_locations will be set to "Y."
table_integrities	**char(8)**	"Y" if this object has ULTRIX/SQL style integrities. If the value is blank, a probe of the **iiintegrities** table will determine if integrities exist or not.
table_permits	**char(8)**	"Y" if this object has ULTRIX/SQL style permissions. A value of blank is not determinant on entries in the **iipermits** table.
all_to_all	**char(8)**	"Y" if this object has the ULTRIX/SQL permission "all to all," or "N" if not.
ret_to_all	**char(8)**	"Y" if this object has the ULTRIX/SQL permission "retrieve to all," or "N" if not.
is_journalled	**char(8)**	"Y" if ULTRIX/SQL journaling is enabled on this object, or "N" if it is not. This value is set to "C" if journaling will be enabled at the next checkpoint. This information will be omitted if ULTRIX/SQL journaling does not apply.
view_base	**char(8)**	"Y" if this is a base for a view definition, "N" if it is not, or blank if the information is unknown.
multi_locations	**char(8)**	"Y" if the table is located in multiple areas, "N" if not.
table_ifillpct	**smallint**	The fill factor for the index pages used on the last **modify** command in the **nonleaffill** clause, expressed as a percentage from 0 to 100. This is used for ULTRIX/SQL **btree** structures in order to rerun the last **modify** command.

Column Name	Data Type	Description
table_dfillpct	**smallint**	The fill factor for the data pages used on the last **modify** command in the **fillfactor** clause, expressed as a percentage from 0 to 100. This is used for ULTRIX/SQL **btree**, **hash**, and **isam** structures in order to rerun the last **modify** command.
table_lfillpct	**smallint**	The fill factor for the leaf pages used on the last **modify** command in the **leaffill** clause, expressed as a percentage from 0 to 100. This is used for ULTRIX/SQL **btree** structures in order to rerun the last **modify** command.
table_minpages	**integer**	The *minpages* parameter from the last execution of the **modify** command. This is used for ULTRIX/SQL **hash** structures only.
table_maxpages	**integer**	The *maxpages* parameter from the last execution of the **modify** command. This is used for ULTRIX/SQL **hash** structures only.
table_relstamp1	**integer**	The high part of the last **create** or **modify timestamp** for the table.
table_relstamp2	**integer**	The low part of the last **create** or **modify timestamp** for the table.
table_reltid	**integer**	The first part of the internal relation ID. This is used to derive the file name for the table.
table_reltidx	**integer**	The second part of the internal relation ID. This is used to derive the file name for the table.

G.2.4 The iicolumns Catalog

For each object in **iitables**, there are one or more entries in **iicolumns**. Each row in the **iicolumns** view contains the information on a column of the queryable object. This view is used by the user programs to perform dictionary operations and dynamic queries.

Column Name	Data Type	Description
table_name	**char(32)**	The name of the table. This is an ULTRIX/SQL name.
table_owner	**char(32)**	The owner of the table. This is an ULTRIX/SQL username.
column_name	**char(32)**	The column's name. This is an ULTRIX/SQL name.
column_datatype	**char(32)**	The column's data type name. Valid type names are: **integer, smallint, int, float, real, double precision, char, character, varchar, c, text, date** and **money.**

Column Name	Data Type	Description
column_length	**integer**	The length of the column as specified by the user. If a data type contains two length specifiers, this column uses the first length. For the data types which are specified without length (**money** and **date**), this will be set to zero. Note that this length is not the actual length of the column's internal storage.
column_scale	**integer**	The second number in a two-part user length specification. For example, for *typename*(len1, len2) it will be len2.
column_nulls	**char(8)**	Tells whether the column can be null. It will be "N" if the column cannot be null. It will be "Y" if the column can be null.
column_defaults	**char(8)**	Tells whether the column is given a default value. It will be "N" if the column is *not* given a default value on insert. It will be "Y" if the column *is* given a default value on insert.
column_sequence	**integer**	The number of this column in the corresponding table's **create** statement, numbered from 1.
key_sequence	**integer**	The order of this column in the primary key, numbered from 1. For an ULTRIX/SQL table, this indicates the column's order in the primary storage structure key. If the value is 0, this column is not part of the primary key. This is unique if the "unique_rule" column for the table's corresponding entry in **iitables** is set to "U."
sort_direction	**char(8)**	Set to "A" for ascending when the key_sequence is greater than (>) 0. Otherwise, this value is a blank.
column_ingdatatype	**smallint**	The ULTRIX/SQL data type of the column. If the value is positive, the column is not nullable; if the value is negative, the column is nullable. The data types and their corresponding values are: **integer** -30/30 **float** -31/31 **c** -32/32 **text** -37/37 **date** -3/3 **money** -5/5 **char** -20/20 **varchar** -21/21

G.2.5 The iiphysical_tables Catalog

The information in the **iiphysical_tables** view is the same as that in a portion of **iitables**. The capability, PHYSICAL_SOURCE, in **iidbcapabilities** can be used to determine whether **iiphysical_tables** must be used. If you do not want to check the **iidbcapabilities** PHYSICAL_SOURCE capability, you should always use **iiphysical_tables** to be sure of getting the correct information.

If a queryable object is type "T," signifying a table, it is a physical table and may have an entry in **iiphysical_tables** as well as **iitables**.

Column Name	Data Type	Description
table_name	**char(32)**	The table name. This is an ULTRIX/SQL name.
table_owner	**char(32)**	The table's owner. This is an ULTRIX/SQL username.
table_stats	**char(8)**	"Y" if the object has entries in **iistats** or "N" if it does not. If the field is blank, it is undetermined if the object has entries in **iistats**, and you should check **iistats** directly. This column is used only for optimization of ULTRIX/SQL databases.
table_indexes	**char(8)**	"Y" if this object has entries in the **iiindexes** table that refer to this as a base table, and "N" if not. If this is blank, it is undetermined if the object has entries in the **iiindexes** table that refer to it as a base table; you must check the **iiindexes** table directly. This field is used only for optimization for ULTRIX/SQL databases.
is_readonly	**char(8)**	"N" if updates are physically allowed on this object and "Y" if not. The field is blank if this is unknown. This field is always set to "N" for ULTRIX/SQL and Rdb/VMS tables.
num_rows	**integer**	The estimated number of rows in the table. This value is set it -1 if this is unknown.
storage_structure	**char(8)**	The storage structure of the table. Possible values are: "HEAP" If the table is a **heap** structure "HASH" If the table is a **hash** structure "ISAM" If the table is a **isam** structure "BTREE" If the table is a **btree** structure Blank (" ") If the structure is unknown
is_compressed	**char(8)**	"Y" if the table is stored in compressed format, "N" if it is not compressed, or blank if this is unknown.
duplicate_rows	**char(8)**	"D" if duplicate rows are allowed in the table, "U" if the rows are unique, or blank if this is unknown.
unique_rule	**char(8)**	"U" if the storage structure is unique, "D" if duplicates are allowed in the physical storage structure key, or blank if this is unknown or does not apply.

Column Name	Data Type	Description
number_pages	**integer**	The estimated number of physical pages in the table. This value is set to -1 if this is unknown.
overflow_pages	**integer**	The estimated number of overflow pages in the table. This value is set to -1 if unknown.
row_width	**integer**	The size, in bytes, of the uncompressed binary value for a row in the object for ULTRIX/SQL. This value is set to -1 if this is unknown.

G.2.6 The iiviews Catalog

The **iiviews** view contains one or more entries for each view in the database. (Views are represented in **iitables** by *table_type* = "V.") Because the "text_segment" column is limited to 240 characters per row, a single view may require more than one entry to represent all its text. There will be as many entries in this table as needed to represent all the text of a view.

The text may be broken in mid-word across the sequenced rows. The text column is pure text. Also, the text may or may not contain newline characters.

Column Name	Data Type	Description
table_name	**char(32)**	The view name. This is an ULTRIX/SQL name.
table_owner	**char(32)**	The view's owner. This is an ULTRIX/SQL username.
view_dml	**char(8)**	The language the view was created in. "S" (for ULTRIX/SQL).
check_option	**char(8)**	"Y" if the check option was specified in the **create view** statement, "N" if not. This will be blank if unknown.
text_sequence	**integer**	The sequence number for the text field, numbered from 1.
text_segment	**varchar(256)**	The text of the view definition.

G.2.7 The iiindexes Catalog

Each queryable object with a *table_type* of "I" in **iitables** has an entry in the **iiindexes** view. In ULTRIX/SQL, all indexes also have an entry in **iiphysical_tables**.

Column Name	Data Type	Description
index_name	**char(32)**	The index name. This is an ULTRIX/SQL name.
index_owner	**char(32)**	The index owner. This is an ULTRIX/SQL username.

Column Name	Data Type	Description
create_date	**char(25)**	Creation date of the index. This is an ULTRIX/SQL standard date.
base_name	**char(32)**	The base table name. This is an ULTRIX/SQL name.
base_owner	**char(32)**	The base table owner. This is an ULTRIX/SQL name.
storage_structure	**char(16)**	The storage structure for the index. It is one of the following: "HEAP" If the table is a heap "HASH" If the table is a hash structure "ISAM" If the table is an isam structure "BTREE" If the table is a btree Blank (" ") If the table structure is unknown
is_compressed	**char(8)**	Set to "Y" if the table is stored in compressed format, or "N" if the table is uncompressed. This will be blank if this is unknown.
unique_rule	**char(8)**	"U" if the index is unique, "D" if duplicate key values are allowed, or blank if unknown.

G.2.8 The iiindex_columns Catalog

For indexes, any ULTRIX/SQL columns that are defined as part of the primary index key will have an entry in the **iiindex_columns** view. For a full list of all columns in the index, use the **iicolumns** view.

Column Name	Data Type	Description
index_name	**char(32)**	The index containing *column_name*. This is an ULTRIX/SQL name.
index_owner	**char(32)**	The index owner. This is an ULTRIX/SQL username.
column_name	**char(32)**	The name of the column. This is an ULTRIX/SQL name.
key_sequence	**integer**	The sequence of the column within the key, numbered from 1.
sort_direction	**char(8)**	Set to "A" for ascending.

G.2.9 The iialt_columns Catalog

For each alternate key, any columns which are defined as part of the key will have an entry in the **iialt_columns** view.

Column Name	Data Type	Description
table_name	**char(32)**	The table that *column_name* belongs to.
table_owner	**char(32)**	The table owner.
key_id	**integer**	The number of the alternate key for this table.
column_name	**char(32)**	The name of the column.
key_sequence	**smallint**	The sequence of the column within the key, numbered from 1.

G.2.10 The iistats Catalog

If a column has statistics, it has a row in the **iistats** view.

Column Name	Data Type	Description
table_name	**char(32)**	The name of the table. This is an ULTRIX/SQL name.
table_owner	**char(32)**	The owner of the table. This is an ULTRIX/SQL username.
column_name	**char(32)**	The column name to which the statistics apply. This is an ULTRIX/SQL name.
create_date	**char(25)**	The date when statistics were gathered. This is an ULTRIX/SQL standard date.
num_unique	**float8**	The number of unique values in the column.
rept_factor	**float8**	The repetition factor, or the inverse of the number of unique values (number of rows/number of unique values).
has_unique	**char(8)**	"Y" if the column has unique values, "N" otherwise.
pct_nulls	**float8**	The percentage (fraction of 1.0) of the table which contains NULL for the column.
num_cells	**integer**	The number of cells in the histogram.

G.2.11 The iihistograms Catalog

The **iihistograms** view contains histogram information used by the optimizer.

Column Name	Data Type	Description
table_name	**char(32)**	The table for the histogram. This is an ULTRIX/SQL name.
table_owner	**char(32)**	The table owner. This is an ULTRIX/SQL username.
column_name	**char(32)**	The name of the column. This is an ULTRIX/SQL name.
text_sequence	**integer**	The sequence number for the histogram, numbered from 1. There may be several rows in this table, used to order the "optdata" data when the histogram is read into contiguous memory.
text_segment	**char(228)**	The histogram data, created by **optimizedb**. This is encoded.

G.2.12 The iipermits Catalog

The **iipermits** view contains one or more entries for each permission defined. Because the text of the permission definition may contain more than 240 characters, **iipermits** may contain more than one entry for a single permission. The text may or may not contain newlines and may be broken mid-word across rows.

This view is keyed on **object_name** and **object_owner**.

Column Name	Data Type	Description
object_name	**char(32)**	The table, view, or procedure name. This is an ULTRIX/SQL name.
object_owner	**char(32)**	The owner of the table, view, or procedure. This is an ULTRIX/SQL username.
object_type	**char(8)**	The type of the object: "T" for a table or view, "P" for a database procedure.
create_date	**char(25)**	The permission's creation date. This is an ULTRIX/SQL standard date.
permit_user	**char(32)**	The username to which this permission applies.
permit_number	**smallint**	The number of this permission.
text_sequence	**smallint**	The sequence number for the text, numbered from 1.
text-segment	**varchar(240)**	The text of the permission definition.

G.2.13 The iiintegrities Catalog

The **iiintegrities** view contains one or more entries for each integrity defined on a table. Because the text of the integrity definition may contain more than 240 characters, **iiintegrities** may contain more than one entry for a single integrity. The text may or may not contain newlines and may be broken mid-word across rows.

This view is keyed on **table_name** and **table_owner**.

Column Name	Data Type	Description
table_name	**char(32)**	The table name. This is an ULTRIX/SQL name.
table_owner	**char(32)**	The table's owner. This is an ULTRIX/SQL username.
create_date	**char(25)**	The integrity's creation date. This is an ULTRIX/SQL standard date.
integrity_number	**smallint**	The number of this integrity.
text_sequence	**smallint**	The sequence number for the text, numbered from 1.
text_segment	**varchar(240)**	The text of the integrity definition.

G.2.14 The iimulti_locations Catalog

Because a table, due to size or space constraints, may be located on multiple volumes, the **iimulti_locations** view contains an entry for each additional location on which a table resides. The first location for a table can be found in the **iitables** catalog.

This view is keyed on **table_name** and **table_owner**.

Column Name	Data Type	Description
table_name	**char(32)**	The table name. This is an ULTRIX/SQL name.
table_owner	**char(32)**	The table's owner. This is an ULTRIX/SQL username.
sequence	**integer**	The sequence of this location in the list of locations, as specified in the **modify** command. This is numbered from 1.
location_name	**char(32)**	The name of the location.

G.2.15 The iiprocedures Catalog

The **iiprocedures** view contains one or more entries for each database procedure defined on a database. Because the text of the procedure definition may contain more than 240 characters, **iiprocedures** may contain more than one entry for a single procedure. The text may or may not contain newlines and may be broken mid-word across rows.

This view is keyed on **procedure_name** and **procedure_owner**.

Column Name	Data Type	Description
procedure_name	**char(32)**	The database procedure name, as specified in the **create procedure** statement.
procedure_owner	**char(32)**	The procedure's owner. This is an ULTRIX/SQL username.
create_date	**char(25)**	The procedure's creation date. This is an ULTRIX/SQL standard date.
proc_subtype	**char(8)**	The subtype of this procedure. For standard ULTRIX/SQL procedures, this will be "N" (native).
procedure_type	**char(8)**	"L" if the procedure is a link, "N" if it is a native procedure.
text_sequence	**smallint**	The sequence number for the test_segment.
text_segment	**varchar(240)**	The text of the procedure definition.

G.2.16 The iiregistrations Catalog

The **iiregistrations** view contains the text of **register** statements.

Column Name	Data Type	Description
table_name	**char(32)**	The name of the registered table, view, or index.
table_owner	**char(32)**	The name of the owner of the table, view, or index.
Table_dml	**char(8)**	The language used in the registration statement. Will be "S" for ULTRIX/SQL.
table_type	**char(8)**	"T" if the object type is a table, "V" if it is a view, or "I" if it is an index.
table_subtype	**char(8)**	The type of table or view created by the **register** statement. For Remote Access to Rdb/VMS, this will be "I" for an imported object.
text_sequence	**integer**	The sequence number of the text field, numbered from 1.
text_segment	**varchar(240)**	The text of the **register** statement .

G.3 The DBMS System Catalogs

The section provides a list of the System Catalogs for the database management system (DBMS System Catalogs) with a short description of each. The table names of the DBMS System Catalogs may be used as arguments to the **sysmod** command (see Chapter 4). These catalogs are not supported for any other use.

DBMS System Catalog	Description
iirelation	Describes each table in the database.
iirel_idx	Indexes the **iirelation** table by table name and owner. This catalog is an index table.
iiattribute	Describes the properties of each column of a table.
iiindex	Describes all the indexes for a table.
iidevices	Describes additional locations when a user table spans more than one ULTRIX/SQL location.
iiintegrities	Contains information about the integrities applied to tables.
iiprotect	Contains information about the protections applied to tables.
iitree	Contains the DBMS internal representation of the query text for views, protections, and integrities.
iiqrytext	Contains the actual query text for views, protections, and integrities.
iidbdepends	Describes the dependencies between views or protections and their base tables.
iixdbdepends	Locates the rows that reference a dependent object in the **iidbdepends** catalog. This catalog is an index table.
iiprocedure	Contains information about database procedures.
iihistogram	Contains database histograms that are collected by the **optimizedb** program.
iistatistics	Contains database statistics that are collected by the **optimizedb** program.
iidatabase	Describes various attributes of each database in an installation.
iidbid_idx	Provides a secondary index built on a column in the **iidatabase** catalog.
iidbaccess	Describes which users have access to private databases.
iiextend	Defines the extended data locations of a database.
iilocations	Maps locations to physical areas and indicates what that location can be used for.

DBMS System Catalog	Description
iirelation	Describes each table in the database.
iiuser	Defines valid users and their privileges in an ULTRIX/SQL installation.

Index

Characters

! (exclamation point)
as comparison operator, 1-31, 1-33
; (semicolon)
as statement separator, 1-2
(number sign)
in object names, 1-2
in termcap descriptions, F-3 to F-5
$ (dollar sign)
in currency displays, 1-8 to 1-9
in object names, 1-2
% (percent sign)
as pattern match character, 1-31
in termcap descriptions, F-7
( ) (parentheses)
and precedence of arithmetic operations, 1-15
for expressions, 1-15
for logical operator grouping, 1-29
for subqueries, 1-30
in predicates, 1-33
* (asterisk)
as Terminal Monitor prompt character, 3-2
count function and, 1-27
exponentiation and, 1-15
in termcap descriptions, F-4
multiplication and, 1-15
+ (plus sign)
addition and, 1-15
- (minus sign)
subtraction and, 1-15
. (period)
as decimal indicator, 1-5, 1-9
/ (slash)
as comment indicator (with asterisk), 1-2, E-10
division and, 1-15
: (colon)
in termcap descriptions, F-2 to F-4
in where clause, 1-46
preceding a variable, 1-46
<> (greater/less than symbol)
as comparison operator, 1-31, 1-33
= (equal sign)
as comparison operator, 1-31, 1-33 to 1-34
in termcap descriptions, F-4
@ (at sign)
in object names, 1-2
[ ] (square brackets)
in pattern matching, 1-32
\ (backslash)
as dereference character, 2-6, 3-4
in termcap descriptions, F-3
Terminal Monitor commands and, 3-1
^ (caret)
as comparison operator, 1-31
_ (underscore)
in object names, 1-2, 1-47
in pattern matching, 1-31
| (vertical bar)
in termcap descriptions, F-2, F-4, F-11

A

\a (Terminal Monitor command), 3-3
Aborting
see also Rollback
transactions, 1-42 to 1-47
Absolute
value, 1-20
Accessdb (command), 5-2
Accessing databases, C-1

Aggregate functions
see Set functions
Aggregates
nulls in, 1-40
All clause, 1-33
And (Boolean operator), 1-29
ANSI format
standard key words in, A-2
Any-or-All (predicate), 1-33
\append (Terminal Monitor command), 3-3
Arctangent function, 1-20
Arithmetic
dates and, 1-16
expressions, 1-15
operators, 1-15
Arrow keys
in termcap descriptions, F-14
As clause, 1-35, 2-20
ASCII characters
allowable, 1-4
conversion to blanks, 3-2
Asterisk (*)
see character list at front of index
At sign (@)
see character list at front of index
Audit trails
for tables, 2-21
Auditdb (command), 5-3
Avg function, 1-27

B

Base tables, 2-22
\bell (Terminal Monitor command), 3-3
Between (predicate), 1-32
Binary format
see Bulk copying
Binary operators, 1-15
Blank operation, C-4
Interactive SQL frame, 4-2
Blanks
in character data type, 1-3
padding with, 1-21, F-4
trailing, 1-21 to 1-22
boolean expressions
If-Then-Else (statement), 2-34 to 2-36
While (statement), 2-62
Boolean operators
SQL, 1-29
Bottom operation, 4-6, C-4, E-8
Boxes (around objects), F-11
Btree (storage structure), 2-42, 2-55
Bulk copying, 2-9, 5-3

C

C data type, 1-3
Caret (^)
see character list at front of index
Cartesian product operator, 1-39
Case
lowercase function, 1-21
uppercase function, 1-22
Catalogdb (command), 5-6
Catalogs (DBMS system), G-1, G-17 to G-18
Catalogs (system)
dates in, G-2
described, G-1 to G-16
iialt_columns, G-13
iicolumns, G-8
iidbcapabilities, G-2
iidbconstants, G-3
iihistograms, G-14
iiindex_columns, G-12
iiindexes, G-11
iiintegrities, G-15
iimulti_locations, G-15
iipermits, G-14
iiphysical_tables, G-10
iiprocedures, G-16
iiregistrations, G-16
iistats, G-13
iitables, G-4
iiviews, G-11
printing statistics from, 5-29
updating, G-2
Cbtree (storage structure), 2-42, 2-55
\cd (Terminal Monitor command), 3-3
Character data
comparing, 1-3

converting, 1-18
in SQL, 1-3, 1-18, 1-20 to 1-23
Chash (storage structure), 2-42, 2-55
\chdir (Terminal Monitor command), 3-3
Cheap (storage structure), 2-42, 2-55
Cheapsort (storage structure), 2-42, 2-55
Checkpoints
establishing, 5-8
Cisam (storage structure), 2-41, 2-55
Ckpdb (command), 5-8
Clauses, 1-29
escape, 1-31
Colon (:)
see character list at front of index
Colors
termcap description for, F-14
Columns (in tables)
as expressions, 1-15
defaults for, 1-40
formats of, 2-19
handling by sets of, 1-26 to 1-28, 1-38 to 1-40
in subselects, 1-36
maximum number of, 1-11, 2-20
naming, 2-20
nullability of, 1-40
selecting, 2-50
sorting, 2-43
updating, 2-60
Columns (Terminal Monitor screen)
in termcap descriptions, F-6
Command
defined, C-1
Comments
in mapping files, E-4, E-10
in SQL, 1-2
Commit (statement), 1-41, 2-2
Comparison operators
predicates in SQL, 1-29
Comparison predicate, 1-31
Complete (command), 4-4
Compression, 2-41 to 2-45
Computation
logarithms and, 1-20
mantissa and, 1-4
Concat function, 1-21
Concurrency, 1-41
Constants
hex, 1-10
null, 1-11
numeric, 1-11
string, 1-10
Constraints
integrity, 2-15
Continue (Terminal Monitor message), 3-2
Control key
for transaction interrupt (Control-C), 1-42
Conversion
of numeric data, 1-17, 1-20
of string/character data, 1-18
Copying
bulk copy for, 2-9
Copy (statement) for, 2-3 to 2-11
Copy from (statement) for, 2-5
Copydb (command) for, 5-10
databases, 5-10
error detection in, 2-4
files to/from tables, 2-3 to 2-11
performance hints for, 2-8
Correlation names, 1-12 to 1-13
Cosine function, 1-20
Count function, 1-27
Create index (statement), 2-12 to 2-14
Create integrity (statement), 2-15
Create procedure (statement), 2-16
Create table (statement), 2-19
Create view (statement), 2-22
Createdb (command), 5-12
CTRL key
see Control key
Cursor
activating on terminals, F-15
in termcap descriptions, F-6, F-8 to F-10
moving within forms, C-2, C-4 to C-5, C-5 to C-6

D

Data
copying, 2-3 to 2-11
deleting, 1-37
inserting, 1-37
manipulating, 1-35 to 1-38

Data types
see also Conversion, Numeric data type
c, 1-3
char, 1-3, 1-18
character, 1-3
date, 1-5 to 1-8
described, 1-2 to 1-10
floating-point, 1-4, 1-17
formats for storage of, 1-9
integer, 1-4
money, 1-8, 1-17
text, 1-4, 1-18
varchar, 1-4, 1-18
Database Administrator (DBA)
establishing, 5-12
Database procedures, 1-43 to 1-47
Databases
accessing/terminating access to, 5-2, 5-12, C-1
audit trail creation for, 5-3
checkpointing of, 5-8
copying, 5-10
creating, 5-12
default locations for, 5-12
destroying, 5-15
example of, 1-12
listing names of, 5-6
moving, 1-48
naming, 5-12
private, 5-12
relocating, 1-48
syntax for access, C-1
transactions in, 1-41 to 1-47
unloading, 5-32
Dates
see also Time
\date (Terminal Monitor command), 3-3
arithmetic operations upon, 1-16
Date_part function, 1-24
Date_trunc function, 1-24
formats of, 1-5 to 1-8
functions for, 1-23 to 1-25
German format, 1-8
in catalogs (system), G-2
interval function, 1-25
ISO (Multinational) format, 1-8
Multinational format, 1-8
selecting current/system, 1-26
Sweden/Finland format, 1-8
unit expressions of, 1-23
US format, 1-8
Dayfile, 3-1
DBA
see Database Administrator (DBA)
Dbmsinfo (command), 1-25
Dbname, C-1
Deadlock
causes of, 1-42
definition of, 1-41
Decimal point, 1-5
Declarations
see also Variable declarations
Declare (statement), 2-24
Defaults
for directory subpaths, 1-48
for field nullability, 1-40
for FRS commands, E-18
for mapping files, E-11
for menu item mapping, E-7
for storage structures, 2-43, 2-55
for terminal-type mapping, E-13, E-18
Delete (statement), 1-37, 2-25
Deleting
data, 1-37
Delete (statement), 1-37, 2-25
rows, 2-25
table space recovery and, 2-25
Destroying
Destroydb (command) for, 5-15
Drop (statement) for, 2-26 to 2-28
Directories
locationnames for, 1-47
Distinct clause, 2-50
Dollar sign ($)
see character list at front of index
Drop (statement), 2-26 to 2-28
Drop integrity (statement), 2-27
Drop permit (statement), 2-28
Drop procedure (statement), 2-29
Duplicates
of table rows, 2-21

E

\e (Terminal Monitor command), 3-3
\ed (Terminal Monitor command), 3-3
Edit (command), 4-3
\edit (Terminal Monitor command), 3-3
\editor (Terminal Monitor command), 3-3
Elseif (statement), 2-35
Embedded SQL
  see also SQL
  key words and, A-1 to A-2
End operation, C-4, E-8
Equijoin, 1-39
Error messages, C-7
  in SQL, 4-7
Errors
  Database procedures and, 1-44
  finding during copy operations, 2-4
  iierrornumber, 1-44
  iirowcount, 1-44
Escape clauses
  in like (predicate), 1-31
Exists (predicate), 1-34
Expiration date (tables), 2-20, 2-49
Exponential functions, 1-20
Exponential notation, 1-4, 1-20
Expressions
  classes of, 1-14 to 1-28

F

Field key
  in help, C-6
Files
  copying to/from, 2-3 to 2-11
Fillfactor, 2-12 to 2-13, 2-43
Find key
  in help, C-6
Find operation, C-4, E-8
Finddbs (command), 5-16
  see also Recovery
Floating-point
  conversion to, 1-17
  data type, 1-4
Forget operation, C-4, E-8
Forms
  see also FRS (Forms Run-Time System)
  cursor movement optimization in, F-15
  Interactive SQL user interface and, 4-1 to 4-9
  Isql (command) and, 5-17 to 5-19
  key mapping for, E-1 to E-21
  using forms-based applications, C-1 to C-7
Forms-based applications, C-1 to C-7
From clause, 1-35, 2-51
FRS (Forms Run-Time System)
  commands for, E-5, F-15
  defined, E-1
  FRS keys and, E-7 to E-8
  key definition in, E-1 to E-21
Function keys, C-5, E-1
  see also Mapping
  activating on terminals, F-12
  and menu operations, C-3
Functions
  avg, 1-27
  date, 1-23, 1-23 to 1-25
  max, 1-27
  min, 1-27
  numeric, 1-20
  scalar, 1-19
  set, 1-26 to 1-28
  string, 1-20 to 1-23
  sum, 1-27

G

\g (Terminal Monitor command), 3-3
Go (command), 4-4
\go (Terminal Monitor command), 3-2 to 3-3
Go operation, C-4, E-8
Grant (statement), 2-30
Granularity, 1-24, 2-56
Graphics
  boxes and, F-11
Greater/less than symbol (<>)
  see character list at front of index
Group by clause, 1-14, 1-28, 1-35 to 1-36, 2-50

H

Hash (storage structure), 2-41, 2-55
Having clause, 1-29, 1-35, 1-37, 2-50
Heap (storage structure), 2-42, 2-55
Heapsort (storage structure), 2-42, 2-55
Help (statement), 2-31
Help operation, C-4, C-6, E-8
  Interactive SQL frame, 4-3
Help Screens, C-6

I

\i (Terminal Monitor command), 3-3
If-Then-Else (statement), 2-34 to 2-36
II_CHECKPOINT, 1-48
II_DATABASE, 1-48
II_DECIMAL, 1-5
II_JOURNAL, 1-48
II_MONEY_FORMAT, 1-9
II_MONEY_PREC, 1-9
II_PRINTSCREEN_FILE, D-2
II_TERMCAP_FILE, F-2
iialt_columns catalog, G-13
iicolumns catalog, G-8
iidbcapabilities catalog, G-2
iidbconstants catalog, G-3
iierrornumber, 1-44
iihistograms catalog, 5-20, 5-29, G-14
iiindex_columns catalog, G-12
iiindexes catalog, G-11
iiintegrities catalog, G-15
iimulti_locations catalog, G-15
iipermits catalog, G-14
iiphysical_tables catalog, G-10
iiprocedures catalog, G-16
iiregistrations catalog, G-16
iirowcount, 1-44
iistats catalog, 5-20, 5-29, G-13
iitables catalog, G-4
iiviews catalog, G-11
In (predicate), 1-33
\include (Terminal Monitor command), 3-3
Indexes
  Create index (statement) and, 2-12 to 2-14
  destroying, 2-14, 2-26
  sorting, 2-12
  storage structure of, 2-41 to 2-45
Input screen, 4-3
Insert (statement), 1-37, 2-37
Insert editing mode, C-6, E-5
Insert operation, C-4
Integers
  as constants, 1-4
  range of, 1-4
Integrity
  constraints, 2-15
  Create integrity (statement), 2-15
  destroying, 2-27
  printing, 2-32
  unloading, 5-32
Interactive SQL
  command line interface, 3-1 to 3-4, 5-26 to 5-28
  forms-based interface, 4-1 to 4-9, 5-17 to 5-19
  Isql (command), 5-17 to 5-19
  Sql (command), 5-26 to 5-28, 5-28
Interactive SQL frame, 4-1 to 4-2
Interrupts, 1-42
Interval function, 1-25
Isam (storage structure), 2-41, 2-55
Isql (command), 4-1, 5-17 to 5-19
ISQL (Interactive SQL)
  see Interactive SQL

J

Join operator, 1-39
Journal entries, 5-3
Journaling
  Auditdb (command) and, 5-3
  Ckpdb (command) and, 5-8
  described, 2-54
  invoking of, 2-21, 2-54
  recovery and, 2-21
  table creation with, 2-21

K

Key words
ANSI, A-2
asc, 1-35
desc, 1-35
distinct, 1-35
embedded SQL and, A-1 to A-2
SQL and, A-1 to A-2
Keyboard keys, C-5, D-2
arrow, F-14
Ctrl key, 1-42
help, C-6
terminal considerations for, D-2
Keys operation, C-6

L

Labels
in mapping files, E-4, E-7
Leaffill, 2-13, 2-44
Left function, 1-21
Length function, 1-21
Level Mapping
user, E-17
Levels
of table access, 2-56
Like (predicate), 1-31
escape clauses in, 1-31
Literals
see Constants
Locate function, 1-21
Locationnames, 1-47 to 1-48, 2-20
Locking
level of, 2-55
Set lockmode (statement) and, 2-55
timeout, 2-57
Lowercase function, 1-21

M

Mapping
described, E-1
disabling of, E-4, E-10
file errors, E-11
files for, E-3, E-11
getting information about, E-18
levels of, E-11
of FRS commands, E-5
of FRS keys, E-3, E-7 to E-8
of menu items, E-6
querying settings of, E-18
statements, E-9
syntax of statements for, E-8, E-10
Max function, 1-27
Maxlocks, 2-57
Maxpages, 2-13, 2-43
Menus, C-2
keys for, C-2, E-6
menu item and, E-7
menuitem and, E-6
standard operations in, E-8
Message (statement)
in Database procedures, 1-45, 2-39
Min function, 1-27
Minpages, 2-13, 2-43
Mode
insert, C-6
overstrike, C-6
Mode (FRS command), E-5
Modify (statement), 1-48, 2-41 to 2-45
Modulo arithmetic, 1-20
Money data type, 1-8

N

Naming
columns, 2-20
conventions for, 1-2
correlation names and, 1-12 to 1-13
in termcap descriptions, F-4
Nesting
of function calls, 1-23
of if statements, 2-35

of queries, 1-30
Next operation, E-8
\nobell (Terminal Monitor command), 3-3
Not (Boolean operator), 1-29
Not null column format, 2-20
Null values
in set functions, 1-27
in SQL, 1-34, 1-40
Nullability
aggregates and, 1-40
for data types, 1-11
for table columns, 1-40
Ifnull function and, 1-25
Isnull (predicate) and, 1-29, 1-34
Number sign (#)
see character list at front of index
Numeric data type
functions of, 1-20
ranges/precision of, 1-4

O

OnError (command), 4-8
Operations
aborting, E-8
menus, C-2
standard, C-4
undoing, E-8
Operators
arithmetic, 1-15
logical, 1-29
relational, 1-38 to 1-40
Optimizedb (command), 5-20 to 5-23
Or (Boolean operator), 1-29
Order by clause, 1-35, 2-50
Output frame, 4-5
Output screen, 4-4
Overstrike editing mode, C-6, E-5
Ownership
see also Permissions
of tables, 2-19, 2-30
of views, 2-30

P

\p (Terminal Monitor command), 3-3
Pad function, 1-21
Padding
see Blanks
Parentheses ( )
see character list at front of index
Patterns
matching, 1-31
Percent sign (%)
see character list at front of index
Permissions
creating, 5-12
dropping, 2-28
on Database procedures, 1-44, 2-28, 2-30
printing, 2-32
unloading, 5-32
Predicates
any-or-all, 1-33
between, 1-32
comparison, 1-31
exists, 1-34
in, 1-33
isnull, 1-34
like, 1-31
search conditions and, 1-29 to 1-34
Printing
\print (Terminal Monitor command), 3-3
Printscreen (statement) and, D-2
screen contents, D-2
Privileges, 2-30
Projection operator, 1-38

Q

\q (Terminal Monitor command), 3-3
Qualifications
see Search conditions
Queries
nested, 1-30

optimizing, 5-20 to 5-23, 5-29
subqueries and, 1-30
\quit (Terminal Monitor command), 3-3
Quit operation, C-5, E-8

R

\r (Terminal Monitor command), 3-3
Range variables, 1-13
\read (Terminal Monitor command), 3-3
Readlock, 2-56
Recovery
checkpoints and, 5-8, 5-24 to 5-25
Finddbs (command) and, 5-16
journaling and, 2-21
Rollforwarddb (command) and, 5-24 to 5-25
Redrawing the screen, D-3
Relational algebra, 1-38
Relational operators, 1-38 to 1-40
Reserved words
see key words
\reset (Terminal Monitor command), 3-3
Restriction operator, 1-39
Result
column, 1-36
structure, 2-54
Retrieving
Select (statement) and, 2-50 to 2-53
values, 2-50 to 2-53
Return (statement)
in Database procedures, 2-47
Right function, 1-21
Rollback, 1-41, 1-42 to 1-47, 2-4, 2-48
Rollforwarddb (command), 5-24 to 5-25
Rows (in tables)
counting, 1-27
deleting, 2-25
duplicates of, 1-37, 2-21, 2-41 to 2-45
grouping, 1-14, 1-39 to 1-40
inserting, 2-37
maximum length of, 1-12, 2-20
selecting, 2-50
sorting, 2-43
Rows (Terminal Monitor screen)
in termcap descriptions, F-6

S

\s (Terminal Monitor command), 3-3
Save operation, E-8
Saving
Save (statement) and, 2-49
table updates, 2-49
Scalar functions, 1-19
Screen
boxing characters on, F-11
clearing, F-6
input, 4-3
output, 4-4
printing contents of, D-2
printing messages on, 2-39
redrawing, D-3
video attributes of, F-10
\script (Terminal Monitor command), 3-4
Scrolling, C-5
direction of, F-15
Search conditions
in SQL, 1-29 to 1-34
Select (statement)
described, 1-35 to 1-37, 2-50 to 2-53
in Database procedures, 2-17
Semicolon (;)
see character list at front of index
Server type, C-1 to C-2
Set (statement)
described, 2-54 to 2-59
set lockmode, 1-42
Set clause, 1-37
Set functions, 1-26 to 1-28
\sh (Terminal Monitor command), 3-3
\shell (Terminal Monitor command), 3-3
Shell operation, C-5
Shift function, 1-21
Sine function, 1-20
Size function, 1-21

Slash (/)
see character list at front of index
Sorting
columns, 2-43
indexes, 2-12
rows, 2-43
SQL
comments in, 1-2
data types in, 1-2 to 1-10
error messages, 4-7
invoking command line interface to, 5-26 to 5-28
invoking forms-based interface to, 5-17 to 5-19
Isql (operating system command), 5-17 to 5-19
key words and, A-1 to A-2
names in, 1-2
Sql (operating system command), 5-26 to 5-28
statement placement in, 1-41 to 1-47
statements/commands in, 2-1 to 2-61
syntax overview, 1-1 to 1-50
Square brackets ([ ])
see character list at front of index
Square root function, 1-20
Squeeze function, 1-22
Standard Catalog Interface, G-1 to G-16
Statdump (command), 5-29
Statistics
for optimizer, 5-20
Storage structures
default keys for, 2-43
modifying, 2-41 to 2-45, 5-31
sort order for, 2-43
Strings
c function, 1-3, 1-20 to 1-23
char function, 1-3, 1-20 to 1-23
concat function, 1-21
find operation for, E-8
functions, 1-20 to 1-23
in SQL, 1-3
left function, 1-21
length function, 1-21
locate function, 1-21
lowercase function, 1-21
padding, 1-21
right function, 1-21
shift function, 1-21
size function, 1-21
squeeze function, 1-22
text function, 1-4, 1-20 to 1-23
trim function, 1-22
uppercase function, 1-22
varchar function, 1-20 to 1-23
varying length, 1-3
Subselects, 1-35
Sum function, 1-27
Superuser (System Administrator) status, 5-8, 5-24
Syntax
for database access, C-1
syntactic level in, 1-30
Sysmod (command), 5-31
System catalogs
see Catalogs (system)
Systems
administrator, 1-47, 5-8, 5-24, E-18
operating system commands, 5-1 to 5-33
returning information about, 5-29
tables for, 1-11, 5-31
ULTRIX/SQL settings for, 2-54 to 2-59

T

Tables
see also Columns, Rows, Views
base, 2-22
combining subsets of, 1-38 to 1-40
copying data from/to, 2-3 to 2-11
creating, 2-19
defined, 1-11
destroying, 2-26
examples of, 1-11
expiration of, 2-20
granting privileges on, 2-30
naming, 1-11
obtaining information about, 2-31
ownership of, 2-19
retrieving into/from, 2-50 to 2-53
saving, 2-49
size of, 2-20
storage structure of, 2-41 to 2-45, 5-31
system, 1-11
virtual, 2-22

Tape devices
checkpoint writing to, 5-8
Termcap descriptions
see also Terminals
examples of, F-17
for specified terminals, F-8, F-16
getting advanced features in, F-10
getting basic features in, F-6 to F-7
how to write, F-1, F-5
list of commands for, F-6
need for, F-1
special characters in, F-3
Termcap file
purpose of, E-1
Terminal Monitor
commands for, 3-2 to 3-4
flags for, 3-4
messages on, 3-1
stacking of commands for, 3-3
ULTRIX/SQL command line interface, 3-1 to 3-4
ULTRIX/SQL forms-based interface, 4-1 to 4-9
use of, 3-1, 4-1
Terminals
see also Termcap descriptions
activating cursor on, F-15
defining, D-1
Digital VT100, F-17
Envision 230, F-18
initializing, F-7
mapping files for, E-13, E-17 to E-18
that support color, F-14
types functional with ULTRIX/SQL, D-3 to D-8
video attributes on, F-10
VT, E-2, E-13, E-17, F-17
Text data type, 1-4
Time
\time (Terminal Monitor command), 3-3
formats for, 1-5
functions for, 1-23 to 1-25
interval function, 1-25
selecting current/system, 1-26
Timeouts, 1-42, 2-57
Top operation, C-5, E-8
Transactions
aborting, 1-42 to 1-47
Commit (statement) for, 1-41, 2-2
control statements for, 1-41
management of, 1-41 to 1-47
rolling back, 1-42, 2-48
Trim function, 1-22
Truncation
of dates, 1-24
Truth functions, 1-29
Tuple
defined, 1-11

U

ULTRIX
see also Systems
Termcap file for, F-1, F-16
ULTRIX/SQL
error messages, C-7
help, C-6
keys, C-5
standard menu item operations in, C-4
Unary operators, 1-15
Underlining
termcap descriptions for, F-10
Underscore (_)
see character list at front of index
Undo operation, C-5, E-8
Union
in select statements, 1-35, 2-50
Unique clause, 2-12, 2-42
Unit expression, 1-23
Unloaddb (command), 5-32
Updating
Update (statement) for, 1-37, 2-60
Upline (FRS command), E-5
Uppercase function, 1-22
User
listing databases accessible to, 5-7
mapping customized for, E-17

V

V_node, C-1
Values
  retrieving, 2-50 to 2-53
  transferring from procedures, 2-47
Values clause, 1-38
Varchar data type, 1-4
Variable declarations
  in Database procedures, 1-46, 2-24
Variables
  dbname, C-1
  range, 1-13
  server_type, C-1
  v_node, C-1
Vertical bar (|)
  see character list at front of index
Video attributes
  in Termcap descriptions, F-10
Views
  creating, 2-22
  destroying, 2-26
  granting privileges on, 2-30
  ownership of, 2-23
  printing of, 2-32
  unloading, 5-32
  updating, 2-22
VT terminals, E-2, E-13, E-17, F-17

W

\w (Terminal Monitor command), 3-4
Where clause, 1-29, 1-35, 2-50
While (statement), 2-62
With clause
  Copy (statement), 2-3
  Create index (statement), 2-12
  Create table (statement), 2-19
  Create view (statement), 2-22
  Modify (statement), 2-41
\write (Terminal Monitor command), 3-4

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-baud modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital Subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road . Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal*	————	SSB Order Processing - WMO/E15 *or* Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

* For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

ULTRIX
ULTRIX/SQL Reference Manual
AA-PBZ6A-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? ______________________________

____

____

What do you like best about this manual? ______________________________

____

____

What do you like least about this manual? ______________________________

____

____

Please list errors you have found in this manual:

Page	Description
______	______________________________
______	______________________________
______	______________________________
______	______________________________
______	______________________________

Additional comments or suggestions to improve this manual:

____

____

____

____

What version of the software described by this manual are you using? ____________

Name/Title ______________________________ Dept. ____________

Company ______________________________ Date ____________

Mailing Address ______________________________

______________________________ Email ____________________ Phone ____________

------ Do Not Tear – Fold Here and Tape --

digital™

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST-CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3-2/Z04
110 SPIT BROOK ROAD
NASHUA NH 03062-9987

------ Do Not Tear – Fold Here --

Cut
Along
Dotted
Line

Reader's Comments

ULTRIX
ULTRIX/SQL Reference Manual
AA-PBZ6A-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? ______________________________

____

____

What do you like best about this manual? ______________________________

____

____

What do you like least about this manual? ______________________________

____

____

Please list errors you have found in this manual:

Page	Description
______	______________________________
______	______________________________
______	______________________________
______	______________________________
______	______________________________

Additional comments or suggestions to improve this manual:

____

____

____

____

What version of the software described by this manual are you using? ______________

Name/Title ______________________________ Dept. ______________

Company ______________________________ Date ______________

Mailing Address ______________________________

______________ Email ______________ Phone ______________

Do Not Tear – Fold Here and Tape

digital™

BUSINESS REPLY MAIL

FIRST–CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3–2/Z04
110 SPIT BROOK ROAD
NASHUA NH 03062–9987

Do Not Tear – Fold Here

Cut
Along
Dotted
Line