ULTRIX/SQL

digital

ULTRIX/SQL Database Adminstrator's Guide

Order Number: AA–PBZ8A–TE

ULTRIX

ULTRIX/SQL Database Administrator's Guide

Order Number: AA-PBZ8A-TE
June 1990

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

This manual describes database administration using ULTRIX/SQL software. Version 1.0 of ULTRIX/SQL is based on INGRES Release 6.2.

digital equipment corporation
maynard, massachusetts

Table of Contents

5 Creating Tables and Views

6 Loading and Unloading a Database

7 Populating Tables

8 ULTRIX/SQL Locking

9 Backup and Recovery

Preface

Purpose of this Document

The *ULTRIX/SQL Database Administrator's Guide* provides the ULTRIX/SQL Database Administrator with information on creating, maintaining, backing up, and recovering ULTRIX/SQL databases. In addition, it describes different types of ULTRIX/SQL users and provides instructions for authorizing users to access ULTRIX/SQL software and databases.

Intended Audience

The *ULTRIX/SQL Database Administrator's Guide* is primarily intended for ULTRIX/SQL Database Administrators. However, in some cases, the responsibilities of the ULTRIX/SQL Database Administrator and the ULTRIX/SQL System Administrator may overlap. Therefore, some of the tasks and responsibilities described in this manual may require permissions typically given to the ULTRIX/SQL System Administrator, but not necessarily given to a Database Administrator. In these cases, you may need to work with your ULTRIX/SQL System Administrator to carry out these responsibilities.

Before reading this manual, the reader should be familiar with ULTRIX/SQL and the ULTRIX operating system.

Organization of this Document

The *ULTRIX/SQL Database Administrator's Guide* is divided into the following parts:

- Chapter 1 describes ULTRIX/SQL users and their tasks, permissions and responsibilities.
- Chapter 2 explains how to allow users access to ULTRIX/SQL, provides instructions on using the **accessdb** utility, and tells how to authorize more than one user at a time by building a users file.
- Chapter 3 tells how to create a database with the **createdb** command, how to list information about databases with the **accessdb** and **catalogdb** utilities, and how to destroy a database with the **destroydb** command.

- Chapter 4 provides instructions on how to create alternate locations for databases, how to create a database using alternate locations, and how to extend an existing database to an alternate location.

- Chapter 5 explains how to create tables and views; it also provides information on table limits, handling duplicate rows in tables, and manipulating columns.

- Chapter 6 tells how to unload and reload a database or selected tables with the **unloaddb** and **copydb** commands.

- Chapter 7 presents methods for loading data into tables using the **copy** statement and discusses considerations for ensuring the data's integrity and validity as well as ways to avoid data errors.

- Chapter 8 discusses the automatic placement of locks on tables or pages of tables to ensure that the many ULTRIX/SQL users trying to access the same database at the same time do not interfere with each other.

- Chapter 9 describes various methods for backing up ULTRIX/SQL databases, such as checkpoints and journals, and the use of the **rollforwarddb** command to recover a database from checkpoints and journals.

- Appendix A describes the directories and files for ULTRIX/SQL system code, data files, checkpoints, journals, the transaction log file, and other ULTRIX/SQL files and provides the environment variable set for each during the initialization of ULTRIX/SQL.

- Appendix B tells how to use the **lockstat** utility to examine the state of the ULTRIX/SQL Lock Database and describes its output.

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 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. In sample terminal output, boldface is used to emphasize sections that require further explanation.

- 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.

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

- Clauses 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.

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 Users and Responsibilities 1

1.1 Overview

This chapter describes ULTRIX/SQL users and their tasks and responsibilities. It discusses the ULTRIX/SQL *superuser* permission, who holds it, and why it is needed. Lastly, it provides a table listing:

- Tasks involved in managing ULTRIX/SQL databases
- Permissions or prerequisites for performing these tasks
- Chapter references for additional information on these tasks

1.2 ULTRIX/SQL Users

ULTRIX/SQL recognizes four different types of users:

- ULTRIX Operating System Administrator
- ULTRIX/SQL System Administrator
- ULTRIX/SQL Database Administrator
- End user

1.2.1 ULTRIX Operating System Administrator

The Operating System Administrator sets up the operating system environment in which ULTRIX/SQL is installed. This person logs in as **root** and is the owner of the root account, which provides all permissions available from the operating system.

The Operating System Administrator helps the ULTRIX/SQL System Administrator by performing all installation and initialization tasks that require root permissions. These include loading ULTRIX/SQL with the **setld** utility and setting up directories, ownerships, and permissions for use by ULTRIX/SQL. Once the ULTRIX/SQL environment is running, the Operating System Administrator is responsible for adding new data areas, expanding system resources, and performing system backups.

1.2.2 ULTRIX/SQL System Administrator

The ULTRIX/SQL System Administrator logs in as **ingres** (or uses **su** to become the **ingres** user) and is owner of the **ingres** account, which provides permissions in the ULTRIX/SQL environment that are needed to initialize and maintain ULTRIX/SQL. As owner of the **ingres** account, this person has the primary responsibility for initializing and maintaining ULTRIX/SQL. Additionally, he or she is the primary holder of the ULTRIX/SQL superuser permission.

1.2.2.1 ULTRIX/SQL Superuser Permission

When ULTRIX/SQL is initialized, both the Operating System Administrator (**root**) and the ULTRIX/SQL System Administrator (**ingres**) automatically receive the ULTRIX/SQL superuser permission. (It is important not to remove this permission from these persons, because they will need it for many of the tasks they are required to perform.)

ULTRIX/SQL superuser permission enables the Operating System Administrator and the ULTRIX/SQL System Administrator to impersonate other users when accessing databases. They do this by using the **-u** flag with ULTRIX/SQL operating system commands. The **-u** flag enables them to temporarily become the Database Administrator for any database. Additionally, they can confer the ULTRIX/SQL superuser permission on any other ULTRIX/SQL user. There is no limit on the number of ULTRIX/SQL superusers that can exist at a site.

ULTRIX/SQL superuser permission is required to perform many ULTRIX/SQL tasks, especially those necessary for initializing and maintaining the ULTRIX/SQL installation. Although Database Administrators do not automatically receive superuser permission, some of the tasks described in this manual require it. If you do not have superuser permission, you will need to perform these tasks with your ULTRIX/SQL System Administrator.

1.2.2.2 ULTRIX/SQL System Administrator Responsibilities

The ULTRIX/SQL System Administrator has the following responsibilities:

- Authorize ULTRIX/SQL users to access ULTRIX/SQL
- Initialize ULTRIX/SQL
- Define ULTRIX/SQL environment variables in the symbol table
- Start, stop, configure, and monitor servers
- Disconnect or suspend a session connected to a server
- Shut down the ULTRIX/SQL installation or parts of it

For instructions on performing these tasks, see the *ULTRIX/SQL Operations Guide.*

1.2.3 ULTRIX/SQL Database Administrator

Anyone who creates a database becomes the Database Administrator for that database. There is no limit on the number of database administrators that can exist at a site. However, before you can create a database, you must be authorized to do so by an ULTRIX/SQL superuser, usually the ULTRIX/SQL System Administrator. Authorization to create databases is provided through the **accessdb** utility; see Chapter 2 for instructions.

As Database Administrator, you determine who can access your database and the objects in it. You do this by:

- Creating a database as public or private. A public database is accessible to all ULTRIX/SQL users; a private one is accessible to its owner only. After a database has been created as private, only an ULTRIX/SQL superuser, using the **accessdb** utility, can grant a user access to it.

- Creating database objects and then giving users permission to use them. Database objects created by any other user are accessible to that user only.

As Database Administrator, you must give users explicit permission to use the following objects, created by you:

- Tables

- Views

As Database Administrator, you can access objects in your database that were created by another user by using the **-u** flag with ULTRIX/SQL operating system commands. This enables you to temporarily become that user.

Note

If a user changes any object owned by the Database Administrator, ULTRIX/SQL creates a new copy of the object, which is owned by that user. This object is private to the user who created it; it is *not* accessible to other ULTRIX/SQL users or to the Database Administrator.

The ULTRIX/SQL Database Administrator has the following responsibilities:

- Create and destroy ULTRIX/SQL databases

- Create public database objects

- Give users access to data through grants on tables and views

- Maintain database and query performance

- Manage locking strategies

- Back up and recover the database

The tasks listed below are sometimes considered the responsibility of the ULTRIX/SQL System Administrator, because they are performed with the **accessdb** utility, which requires ULTRIX/SQL superuser permission. However, you may need to perform them as part of your database administration responsibilities:

- Authorize users to access ULTRIX/SQL and to create ULTRIX/SQL databases
- Define alternate locations for database files
- Authorize (extend) databases to use alternate locations

For a summary of database administration tasks described in this manual and the permissions or prerequisites required to perform them, see the section that follows entitled "Managing ULTRIX/SQL Databases—A Summary."

1.2.4 End User

An end user is anyone who uses ULTRIX/SQL and is not an Operating System Administrator, ULTRIX/SQL System Administrator, Database Administrator, or ULTRIX/SQL superuser. Since ULTRIX/SQL was designed for a wide variety of users, this can be anyone from an application developer with many years of experience to a data entry clerk with little or no computer experience.

End users can:

- Use any public database and any private one to which they have been authorized access.
- Create database objects in any database to which they have access. These objects can be viewed, updated, and destroyed *only* by the user who created them or by a superuser, who is impersonating the creator by using the **-u** flag with ULTRIX/SQL operating system commands.
- Use existing tables and views to which the Database Administrator has granted them access.

1.3 Managing ULTRIX/SQL Databases—A Summary

The following table provides a summary of the database administration tasks described in this manual. It lists:

- Tasks that are required to create and manage ULTRIX/SQL databases
- Commands, statements, or utilities used to perform these tasks
- Required permissions or prerequisites
- Chapter references for additional information

In the following table, the term "superuser" always refers to the ULTRIX/SQL superuser and "root" refers to the ULTRIX Operating System Administrator.

Table 1-1: Tasks Required to Manage ULTRIX/SQL Databases

Task	Command, Statement, or Utility	Prerequisite	For more information, see
Authorize users to access ULTRIX/SQL	**accessdb**	superuser	Chapter 2
Create databases	**createdb**	authorization by a superuser to create databases	Chapter 3
Destroy databases	**destroydb**	Database Administrator for the database or a superuser	Chapter 3
Define ULTRIX/SQL locations	**accessdb**	superuser, root	Chapter 4
Authorize (extend) a database to use alternate locations	**accessdb**	superuser, root	Chapter 4
Create public tables and views	**create table, create view**	Database Administrator for the database or a superuser	Chapter 5
Unload/copy data from one database into another	**unloaddb, copydb**	Database Administrator for the database or a superuser	Chapter 6
Populate tables with data	**copy, insert**	table owner or permission from Database Administrator to update the table	Chapter 7
Manage locking strategies implemented by application developers	**set lockmode**	authorization to access the database	Chapter 8
Back up and recover the database	**ckpdb, rollforwarddb**	Database Administrator for the database or a superuser	Chapter 9

Authorizing User Access 2

2.1 Overview

This chapter tells you how to allow users access to ULTRIX/SQL. It provides instructions on using the **accessdb** utility to:

- Give a new user access to ULTRIX/SQL and specific ULTRIX/SQL databases
- Modify an existing user's access to ULTRIX/SQL and specific ULTRIX/SQL databases
- Delete an existing user's access to ULTRIX/SQL and specific ULTRIX/SQL databases
- Find out an existing user's authorizations

It also tells you how to authorize more than one user at a time by building a users file when ULTRIX/SQL is initialized.

2.2 Using the accessdb Utility

The **accessdb** utility lets you provide, modify, and query user access to ULTRIX/SQL for one user at a time. To use **accessdb**, you need:

- ULTRIX/SQL superuser permission

  See Chapter 1 for a description of the ULTRIX/SQL superuser permission.

- An understanding of how to use a forms-based program, such as **isql** or **catalogdb**.

  The **accessdb** utility is a forms-based program. If you do not know how to use an ULTRIX/SQL forms-based program, refer to the *ULTRIX/SQL Reference Manual.*

- A cursor-addressable terminal that you must identify with the environment variable TERM_INGRES or TERM.

If you have problems when you are using **accessdb**, you can access a **Help** screen from the **accessdb** menu by selecting **Help**.

2.2.1 Identifying Your Terminal to accessdb

You can only run **accessdb** on a cursor-addressable terminal (for example, a VT100) that you must identify. To identify the terminal, you set the environment variable TERM_INGRES or TERM to a terminal definition contained in the ULTRIX/SQL $II_SYSTEM/sql/files/termcap file.

If you do not specify a terminal with TERM_INGRES, the value in TERM is used by default. If that value is not a valid ULTRIX/SQL terminal definition, ULTRIX/SQL fails with an error. Often you may want to identify terminal capabilities using TERM_INGRES to access features not activated by ULTRIX TERM values.

For example, if you are using a VT100 terminal with function keys active in the C shell environment, you identify your terminal to ULTRIX/SQL by including the following command in your .login or .cshrc file:

```
setenv TERM_INGRES vt100f
```

To identify your VT100 terminal in the Bourne shell environment, include the following commands in your .profile file:

```
TERM_INGRES=vt100f
export TERM_INGRES
```

You can specify any of the valid terminal codes listed in the *ULTRIX/SQL Reference Manual.*

2.2.2 Authorizing New Users One at a Time

You must have access to the **ingres** account or have ULTRIX/SQL superuser permission to grant new users access to ULTRIX/SQL or old users new privileges. Using the **accessdb** utility, you can add, modify, or delete ULTRIX/SQL users, one at a time, and authorize them to access ULTRIX/SQL.

Authorizing access to ULTRIX/SQL is different from granting permission to view and manipulate the data in ULTRIX/SQL databases. Only the Database Administrator can grant permission to use the data in your tables with the SQL **grant** statement.

To authorize a new user to use ULTRIX/SQL:

1. Start **accessdb** by issuing the **accessdb** command at the operating system prompt.

2. Select **User** from the **accessdb** main menu.

3. Type the login for the user you wish to add at the following prompt:

   ```
   User name:
   ```

   Note

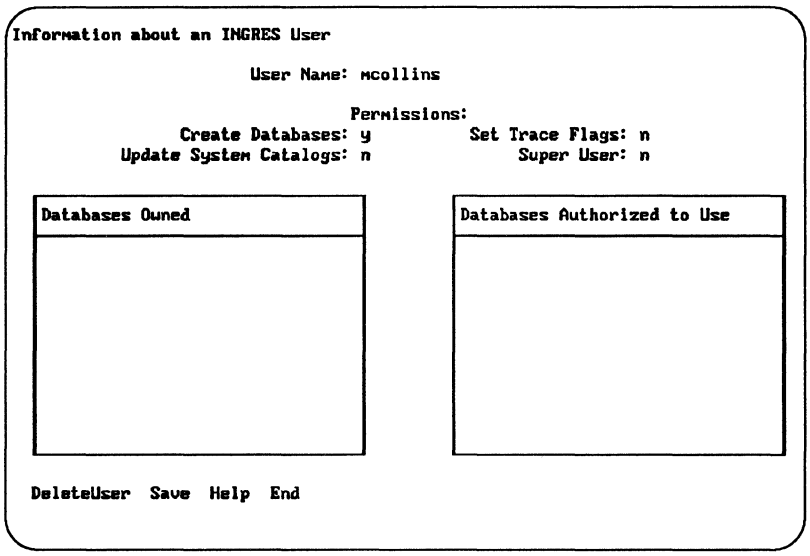
   Because ULTRIX is case sensitive, to avoid problems, we strongly recommend that only lower case ULTRIX logins be specified for ULTRIX/SQL users.

4. Type **y** (yes) at the following prompt:

```
Unknown user - do you want to create a new user?
```

A form that contains information describing the new user appears. It contains a list of four types of permission that you can assign to the user by changing the default values that are displayed.

Figure 2-1: Information About a User Frame



5. Change the default value of the Create Databases permission from **y** to **n** if you want to deny the user permission to create new databases.

6. Change the default value of the Update System Catalogs permission from **n** to **y** if you want to permit the user to update system catalogs with a query language. (This should be granted with caution.)

7. Change the default value of the Set Trace Flags permission from **n** to **y** if you want to permit the user to set the debugging trace flags within ULTRIX/SQL.

8. Change the default value of the Super User permission from **n** (no) to **y** (yes) if you want to give ULTRIX/SQL superuser permission to the user. See Chapter 1 for a description of the ULTRIX/SQL superuser permission.

9. Select **Save** from the menu to save the changes you just entered, and return to the **accessdb** main menu.

10. Repeat steps 2-9 for each new user you wish to authorize.

11. Select **Quit** to leave **accessdb**.

2.2.3 Modifying an Existing User's Access Rights

To modify an existing user's (that is, either your own or another's) access to ULTRIX/SQL and user capabilities:

1. Start **accessdb** by issuing the **accessdb** command at the operating system prompt.

2. Select **User** from the **accessdb** main menu.

3. At the following prompt, type the login of the user you wish to modify:

```
User name:
```

4. Reassign any of the privileges on the form by changing their current values. Type **y** (yes) or **n** (no).

   Do not attempt to add or delete any database names in the read-only table field called Databases Owned. You can only use the **createdb** or **destroydb** commands to add or delete databases.

5. Select **Save** from the menu to save the changes you just entered and return to the **accessdb** main menu.

6. Repeat steps 1-5 for each existing user whose authorization you wish to modify.

7. Select **Quit** to leave **accessdb**.

2.2.4 Deleting an Existing User

To delete an existing user's access to ULTRIX/SQL:

1. Repeat steps 1-4 in the procedure "Modifying an Existing User" above.

2. When the user's information form appears on the screen, select **Delete** from the menu.

3. Return to the **accessdb** menu and repeat steps 1-3 for each user you want to delete.

4. Select **Quit** to leave **accessdb**.

Since there are many places in the system catalogs where a user's name could appear, you cannot delete more than one user at one time.

Note

You are not allowed to delete any user who is the owner of a database, table, or view.

2.2.5 Listing Authorized Users

To list the names of the ULTRIX/SQL users at your installation and their corresponding permissions:

1. Start **accessdb** by issuing the **accessdb** command at the operating system prompt.

2. Select **Catalog** from the **accessdb** menu.

3. Select **Users** from the **Catalog** menu.

   The Catalog Users screen appears. It contains a scrollable table that displays the logins and user permissions of all the ULTRIX/SQL users at your installation.

   You cannot modify the information displayed in this table. To change information about a particular user you must return to the **accessdb** main menu and select the **User** function. To modify or delete a user, see the procedures in the sections "Modifying an Existing User's Access Rights" or "Deleting an Existing User" earlier in this chapter.

4. To return to the **accessdb** main menu, select **End.**

5. Select **Quit** to leave **accessdb.**

2.2.6 Other Uses for accessdb

The **accessdb** utility can also be used to:

- Create nondefault locations for databases, checkpoints, and journals
- Extend an existing database to new locations
- Authorize access to private databases
- Change a database from private to public or from public to private

These functions are described in Chapter 3 and Chapter 4.

2.3 Using the Users File

With the $II_SYSTEM/sql/files/users file, you can authorize a large number of new users at initialization time. This section tells you how to create a new users file.

2.3.1 User Validation

ULTRIX/SQL keeps track of valid ULTRIX/SQL users in two places:

- An ULTRIX/SQL system catalog
- The $II-SYSTEM/sql/files/users file, located in the $II_SYSTEM/sql/files directory

The users file is submitted to the ULTRIX/SQL **iibuild** procedure, when ULTRIX/SQL is initialized for the first time. The file provides the initial entries for one of the ULTRIX/SQL system catalogs that is created when the master database (**iidbdb**) is created. That system catalog contains the logins of users authorized to use ULTRIX/SQL. Each time a user tries to use ULTRIX/SQL, his or her login is compared against the entries in the catalog.

Because ULTRIX/SQL uses both the system catalog and the users file, the catalog and file must remain consistent with each other.

2.3.2 Creating the Users File to Add a Large Number of New Users

To add a large number of new users all at once when ULTRIX/SQL is initialized, create a users file and then have your ULTRIX/SQL System Administrator run **iibuild** using this file instead of the version of the users file that **iibuild** creates by default.

When you run **iibuild** for the first time, you are prompted for the pathname for the users file. If you do not supply the pathname, **iibuild** creates the users file in the $II_SYSTEM/sql/files directory. To build the users file, add an entry for each new user. The entry consists of a line containing four fields, each of which is separated by an exclamation point (!). The format for each field is as follows:

- The username in lowercase in the first field
- A zero (0) in the second field (currently not used)
- A zero (0) in the third field (currently not used)
- A fourth field, the status code field, which contains a number representing the permissions you can assign to the user

The status code field can contain a single number representing a single permission or it can contain a sum, representing any two, three, or all four permissions. The codes for each permission are shown in the following table:

Table 2-1: Permission Codes

Status Code	Permission
100000	Superuser permission
20	Ability to set trace flags
4	Ability to update system catalogs
1	Ability to create databases

The following table provides examples of status codes representing more than one permission.

Table 2-2: Codes Representing More Than One Permission

Status Code	Permission
100001	Superuser and the ability to create databases
100025	All privileges
24	Set trace flags and update system catalog

An example of an entry for user "samiam" with the superuser, **update system catalogs,** and **create databases** permissions is:

```
samiam!0!0!100005
```

2.3.3 Using a Copy of Another Installation's Users File to Add New Users

If you are creating a new installation, you can add new users by using a copy of the users file of another ULTRIX/SQL installation.

If you edit the users file, do so with the names of new users, their accounts, and permissions *before* the **iibuild** command is run to initialize ULTRIX/SQL. Then, when the master database is created during the **iibuild** (initialization) procedure, the ULTRIX/SQL system catalogs will be current and consistent with the users file.

If you edit the users file after the master database is created, the listing in the system catalog is not updated. Thus, editing the users file after the master database is created is not a way to add new users. For information on adding new users, refer to the section entitled "Authorizing New Users One at a Time" earlier in this chapter.

2.3.4 Restoring the Users File

If your installation's users file is destroyed, you can recover it. You must create a temporary users file in the $II_SYSTEM/sql/files directory with the following lines in it:

```
$ingres!0!0!100025
ingres!0!0!100025
root!0!0!100025
```

These lines comprise the template with which a new ULTRIX/SQL installation begins.

Then use the following procedure to run **accessdb** from the **ingres** account:

1. Start **accessdb** by issuing the **accessdb** command at the operating system prompt.
2. Select **User** from the **accessdb** main menu.
3. At the following prompt, specify the root account by typing **root**:

```
User name: root
```

4. Select **Save** from the menu to save the changes you just entered. You do not actually need to make a change to this screen. The users file will get updated with information from the appropriate system catalog and contain an entry for every ULTRIX/SQL user account that exists in the system catalog.

5. Select **Quit** to leave **accessdb**.

Creating and Destroying Databases 3

3.1 Overview

This chapter tells you how to create a database using the default locations for database files that are established when ULTRIX/SQL is initialized. It describes the **createdb** command and tells you how it works. It provides instructions for listing information about databases with the **accessdb** and **catalogdb** utilities. Lastly, it tells you how to destroy a database with the **destroydb** command.

3.2 ULTRIX/SQL Database Files

Each ULTRIX/SQL database contains a configuration file, data files, checkpoint files if you checkpoint your database, and journal file, if you journal your database. These files are described below.

The *configuration file* contains administrative details about the database, such as the location of data, checkpoint, and journal files.

Data files consist of the following:

- User tables and indexes that you or another authorized user create in the database.
- The system catalogs, which are tables that contain information about the database such as descriptions of its tables, columns, and views. System catalogs exist for each database and can be viewed by any ULTRIX/SQL user. For a complete description of the system catalogs, see the *ULTRIX/SQL Reference Manual.*

Checkpoint files contain a static copy of your entire database. A checkpoint file is created each time you use the **ckpdb** command to take a checkpoint of your database.

Journal files contain dynamic records of changes committed to the journaled tables in your database. Both checkpoint and journal files are used to recover your database if a disk crashes. For information about backing up and recovering your database, see Chapter 9.

When ULTRIX/SQL is initialized, default locations are established for each of these types of files. Chapter 4 provides a list of these locations as well as instructions for creating a database using alternate locations.

3.3 The Master Database

The master database, called the **iidbdb**, contains information about all ULTRIX/SQL databases, their locations, and the users who can access them. The master database is sometimes referred to as the ULTRIX/SQL system database.

3.4 Types of Databases

There are two types of databases:

- A *public* database, which is accessible to all ULTRIX/SQL users.
- A *private* database, which is accessible only to its creator, to an ULTRIX/SQL superuser, and to other users designated by an ULTRIX/SQL superuser with the **accessdb** utility.

When a database is public, it means that all ULTRIX/SQL users can access the database. It does *not* mean that all ULTRIX/SQL users can access its data. As Database Administrator, you control access to the data in your database by granting users permission to access tables and views that you own. See the *ULTRIX/SQL Reference Manual* for details.

3.5 Creating an ULTRIX/SQL Database

Before you can create a database in ULTRIX/SQL, you must be authorized to do so by the ULTRIX/SQL System Administrator or another superuser. See Chapter 2 for instructions.

3.5.1 How Many Databases Can Be Created

ULTRIX/SQL does not limit the number of databases that you can create. You can create as many databases as your operating system configuration allows. Additionally, you can use alternate locations to spread your databases across disks. For information about using alternate locations, see Chapter 4.

3.5.2 Rules for Naming Databases

Database names must follow these rules:

- They must be unique within an ULTRIX/SQL installation.
- They can be up to 24 alphanumeric characters, including an underscore. No special characters other than the underscore are allowed.
- They must begin with any alphabetic character except the underscore.

ULTRIX/SQL stores database names in lowercase. If you use uppercase to name your database, ULTRIX/SQL automatically changes the name to lowercase.

3.5.3 The createdb Command

The syntax of the **createdb** command is:

createdb [**-u***username*] [**-p**] *dbname* [**-d***location*] [**-c***location*] [**-j***location*]

For a complete description of the flags and parameters for **createdb**, see the *ULTRIX/SQL Reference Manual.*

After you issue **createdb**, ULTRIX/SQL:

- Updates the system catalogs in the master database (**iidbdb**).
- Creates a new subdirectory, with the name of the database, under the database location for the database. If you use the default location for your database, this subdirectory will be under $II_DATABASE/ingres/data/default.
- Copies the appropriate files from the database template directory ($II_SYSTEM/sql/dbtmplt) to the new database directory.
- Creates the system catalogs for the new database.
- Modifies the system catalogs for the new database.
- Creates the database's configuration file (aaaaaaaa.cnf) in the database's data directory.
- Updates the system catalogs for the new database.

3.5.4 Creating a Public Database

To create a public database that uses default locations, issue the **createdb** command at the operating system prompt:

createdb *dbname*

To locate a database's data, checkpoint, or journal files in locations other than the default location, see Chapter 4.

3.5.5 Creating a Private Database

When you create a database, it is automatically public, unless you use the **-p** flag with the **createdb** command. To create a private database that uses the default locations, issue the following command at the operating system prompt:

createdb -p *dbname*

If an unauthorized user tries to access a private database, ULTRIX/SQL returns an error message.

To locate a private database's data, checkpoint, or journal files in locations other than the default location, see Chapter 4.

3.5.5.1 Authorizing Access to Private Databases

To authorize a user to access a private database, use the **accessdb** utility. You must be an ULTRIX/SQL superuser to use **accessdb**.

To authorize a user to access a private database:

1. Start **accessdb** by issuing the **accessdb** command at the operating system prompt.
2. Select **Database** from the menu.
3. At the following prompt, enter the name of the private database:

   ```
   Database:
   ```

4. When the Information About a Database frame appears:
   a. Tab to the Authorized Users table field.
   b. Enter the logins of the users you want to authorize to use this database.
5. Select **Save** from the menu.
6. Select **Quit** to leave **accessdb**.

3.5.5.2 Changing a Database from Private to Public

To change a database from private to public, or from public to private, use the **accessdb** utility. You must be an ULTRIX/SQL superuser to use **accessdb**.

To change database access from private to public, or public to private:

1. Start **accessdb** by issuing the **accessdb** command at the operating system prompt.
2. Select **Database** from the menu.
3. At the following prompt, enter the name of the database:

   ```
   Database:
   ```

4. When the Information About a Database frame appears, the cursor will be in the Access field. Type **private** over **public** (or **public** over **private**).
5. If you changed the access from public to private,
   a. Tab to the Authorized Users field.
   b. Enter the logins of the users who you want to access this database. Remember to enter your (the Database Administrator's) login.
7. Select **Save** from the menu.
8. Select **Quit** to leave **accessdb**.

3.6 Listing Information about ULTRIX/SQL Databases

You can list ULTRIX/SQL database information with the following utilities:

- **accessdb**
- **catalogdb**

Instructions for using these utilities appear in the following sections.

3.6.1 Listing Database Information with accessdb

With the **accessdb** utility, you can list information about all ULTRIX/SQL databases. To use **accessdb**, you must be an ULTRIX/SQL superuser.

To list database information with **accessdb**:

1. Start **accessdb** by issuing the **accessdb** command at the operating system prompt:
2. When the Accessdb frame appears, make one of the following selections:

To see	Select
All ULTRIX/SQL databases, their owners, access status, and type	**Catalog** and then **Databases**
All ULTRIX/SQL users and their permissions	**Catalog** and then **Users**
All locations and their ULTRIX directories	**Catalog** and then **Locationnames**
All databases using a particular location	**LocationNames** and then enter the location name.
Information about a particular ULTRIX/SQL user	**User** and then enter the user name
Information about a particular database	**Database** and then enter the database name.

3. After viewing the information, select **End**.
4. Select **Quit** to leave **accessdb**.

3.6.2 Listing Database Information with catalogdb

With the **catalogdb** utility, you can list information about all of the ULTRIX/SQL databases *that you own*. Any ULTRIX/SQL user can use **catalogdb**; unlike **accessdb**, you do not need to be an ULTRIX/SQL superuser.

To list database information with **catalogdb**:

1. Start **catalogdb** by issuing the **catalogdb** command at the operating system prompt.
2. When the Catalogdb frame appears, make one of the following selections:

To see	Select
All databases that you own, their access status, and type	Catalog and then **Databases**
Locations and their ULTRIX directories	Catalog and then **Locationnames**
Your databases and their extended locations	Catalog and then **DbExtension**
Information about a particular database (its access status, authorized users, and data, ckp, and jnl locations)	**Database** and then enter the name of the database
All databases that you own, all databases to which you have access, and your permissions	**User**

3. After viewing the information, select **End**.
4. Select **Quit** to leave **catalogdb**.

3.7 Destroying a Database

To destroy a database, use the **destroydb** command. The syntax for this command is:

destroydb [**-u***username*] *dbname*

To use this command, you must be the Database Administrator for the database or an ULTRIX/SQL superuser impersonating the Database Administrator for the database.

The **destroydb** command:

- Deletes the database, checkpoint, and journal directories for the database.
- Removes all traces of the database from the master database (**iidbdb**).

For a complete description of the **destroydb** command, see the *ULTRIX/SQL Reference Manual*.

Using Alternate Locations 4

4.1 Overview

When ULTRIX/SQL is initialized, your ULTRIX/SQL System Administrator creates default locations for the files that comprise a database (data, checkpoint, and journal files.)

Chapter 3 provides instructions for creating a database using the default locations. However, you do not have to place your database files in the default locations; you can use alternate locations for them. Using alternate locations enables you to organize your ULTRIX/SQL installation, databases, and tables across multiple disks.

This chapter tells you how to:

- Create alternate locations
- Create a database using alternate locations
- Extend an existing database to an alternate location. Extending a database means putting the database's tables and indexes in more than one location.

4.2 What Is an ULTRIX/SQL Location?

An ULTRIX/SQL location is a logical entity. A location associates a directory path (area) on a disk with a label, or *locationname*. The distinction between the logical location and the physical path allows databases to be defined independent of operating system or site. *locationname*

Note

The ULTRIX/SQL **accessdb** form refers to the directory structure as an **area**. It is possible to use the **ingsetenv** command to set any variable name you choose to a directory structure (area). If you do this, you can enter the variable name on the Accessdb form instead of the full path name. (See the section "Creating an ULTRIX/SQL Location" in this chapter for specific instructions.) However, since ULTRIX/SQL will expand the variable to the full path name in the configuration file, do not change the variable after you have set it.

For information about **ingsetenv**, see the *ULTRIX/SQL Operations Guide*.

When you use certain commands, such as **createdb**, **create table**, or **create index**, you specify the location with the *locationname*, not the directory specification.

4.3 Default Locations for Database Files

When ULTRIX/SQL is initialized, your ULTRIX/SQL System Administrator creates the following default locations for data, checkpoint, and journal files:

Table 4-1: Default Locations for Database Files

Type of File	Default Location	Directory Created for Default
data	ii_database	$II_DATABASE/ingres/data/default
checkpoint	ii_checkpoint	$II_CHECKPOINT/ingres/ckp/default
journal	ii_journal	$II_JOURNAL/ingres/ckp/default

4.4 Guidelines for Using Alternate Locations

The following list contains guidelines for using alternate locations:

- You can use alternate locations for a database's data, checkpoint, and journal files.
- You can store any of a database's data files, except system catalogs, in more than one location; this is called *extending* a database. After you have extended a database to an alternate location, you can place new data files in it or relocate existing data files to it. For more information, see the section "Extending a Database to an Alternate Location" below.
- You *must* store all of a database's checkpoint files in a single location. The location is determined when you create the database.
- You *must* store all of a database's journal files in a single location. The location is determined when you create the database.
- By default, the data, checkpoint, and journal files for a database are stored in the same locations and on the same disk. We recommend that you store data files in a different location and on a different disk from those used to stored checkpoint and journal files. Doing so helps to protect your data in the event of disk failure.
- After creating a database:
    - You can change the location of the user tables and indexes, but not the system catalogs. For instructions, see Chapter 5.
    - You *cannot* change the location of its checkpoint and journal files. Since these locations are written permanently in the configuration file, you cannot change them later.

The following table summarizes the guidelines for using alternate locations:

Table 4-2: Summary of Guidelines for Using Alternate Locations

Database File	Can use alternate location	Can extend to multiple locations	OK to change location
data (user tables, indexes)	yes	yes	yes
data (system catalogs)	no	no	no
checkpoint	yes	no	no
journal	yes	no	no

4.5 Creating Alternate Locations for a New or Existing Database—An Overview

Before using an alternate location for a new or existing database, the location *must* exist. The following steps provide an overview of the process for creating a new database in an alternate location or extending an existing database to an alternate location:

1. Create the directory structure for the new location.

   For instructions, see the section "Creating a Directory Structure for a New Location" below.

2. Create the new location and map it to the directory structure.

   For instructions, see the section "Creating an Alternate Location" below.

3. Use the new location when you create a new database or extend an existing one. See the section "Creating a Database in an Alternate Location" or "Extending a Database to an Alternate Location" below.

4.5.1 Creating a Directory Structure for a New Location

You can create a directory structure for a new location:

- Within the ULTRIX/SQL installation area
- Outside the ULTRIX/SQL installation area

You need ULTRIX/SQL system permission to create the directory structure within the ULTRIX/SQL installation area. You need ULTRIX operating system permission to create the directory structure outside the ULTRIX/SQL installation area.

4.5.1.1 Creating a Directory Structure Within the ULTRIX/SQL Installation Area

To create a directory structure within the ULTRIX/SQL installation area:

1. Log in or **su** to the ULTRIX/SQL System Administrator's account (**ingres**).

   The **ingres** account must own the subdirectories that you will create in this procedure.

2. Change directory to the appropriate default directory:

   - $II_DATABASE/ingres/data to create a new location for data files
   - $II_CHECKPOINT/ingres/ckp to create a new location for checkpoints
   - $II_JOURNAL/ingres/jnl to create a new location for journal files

   For example, if you are going to create a new location for data files, issue the following command:

   ```
   cd $II_DATABASE/ingres/data
   ```

3. Make a new subdirectory. For example, to make a directory named new_area, issue the following command:

   ```
   mkdir new_area
   ```

4. Change the permissions of the new directory to 777. For example:

   ```
   chmod 777 new_area
   ```

5. You have now created a new subdirectory inside the ULTRIX/SQL installation area. To create a new location and map it to this area, see the section "Creating a New Location" below.

4.5.1.2 Creating a Directory Structure Outside the ULTRIX/SQL Installation Area

To create a new area outside the ULTRIX/SQL installation:

1. Log in or **su** to the ULTRIX operating system account, **root**.

2. Change location to the directory where you will create the new directory structure. For example, to create the new directory structure in the /otherplace directory, issue the command:

   ```
   cd /otherplace
   ```

3. Create a new subdirectory owned by the **ingres** account. For example:

   ```
   mkdir new_area
   chown ingres new_area
   ```

4. Log in or **su** to the **ingres** account. The **ingres** account must own the subdirectories that you will create in this procedure.

5. Create subdirectories for the types of database files that use the new area.

   For example, to create a subdirectory for data files in new_area, issue these commands:

```
mkdir new_area/ingres
mkdir new_area/ingres/data
mkdir new_area/ingres/data/default
```

To make subdirectories for checkpoints, substitute **ckp** or **jnl** for **data** when issuing the above commands.

6. Place the appropriate permissions on the new directories and subdirectories, as shown in this example below:

```
chmod 755 new_area
chmod 755 new_area/ingres
chmod 700 new_area/ingres/data
chmod 777 new_area/ingres/data/default
```

6. You now have created a new directory structure outside the ULTRIX/SQL installation area. To create a new location and map it to the new directory structure, see the section "Creating a New Location" below.

4.5.1.3 Directory Structure Summary

The following table provides a summary of the directory structure created in the examples in the sections above. Additionally, it shows the sample location to which you will map the directory structures, as described in the next section "Creating a New Location."

Table 4-3: Directory Structure for Alternate Locations

Location is	Name of New Location	Name of New Area	Directory Structure (path)
Within the ULTRIX/SQL installation	new_loc	new_area	$II_DATABASE/ingres/data/new_area $II_CHECKPOINT/ingres/ckp/new_area $II_JOURNAL/ingres/jnl/new_area
Outside the ULTRIX/SQL installation	new_loc	/otherplace/new_area	/otherplace/new_area/ingres/data/default /otherplace/new_area/ingres/ckp/default /otherplace/new_area/ingres/jnl/default

4.5.2 Creating an Alternate Location

When you create a new location, you must map it to an *existing* directory structure. If the directory structure does not exist, create it before you perform the following procedure. See the section "Creating a Directory Structure for a New Location" above.

When creating a new location, the *locationname*:

- Must be 24 characters or less
- Must be alphanumeric
- May contain the underscore character

- Must begin with a letter or the underscore
- Will be translated to lowercase, if you enter it in uppercase

To create a new location, you use **accessdb**, which requires ULTRIX/SQL superuser permission.

To create a new location:

1. Issue the **accessdb** command at the operating system prompt.
2. Select **LocationName** from the menu.
3. At the following prompt, type the new location name:

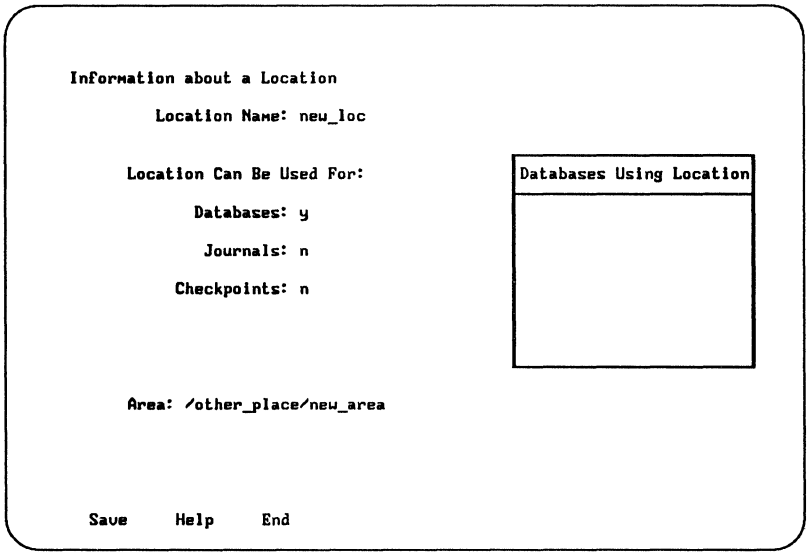
   ```
   Location name:
   ```

   Remember that the *locationname* is the name of a logical entity. It is not equivalent to the path with which the locationname is later associated.
4. Press **Return.**
5. When the following message appears, type **y** (yes):

   ```
   Unknown location - do you want to create a new one?
   ```
6. Press **Return.**

   The Information about a Location frame (Figure 4-1) appears.

Figure 4-1: Information About a Location



When you create or extend a database to this location, ULTRIX/SQL automatically puts the database name in the Databases Using Location table field on this frame. This field is read-only. You cannot specify databases that will use this location by entering them in this field.

7. Type the name of the directory to which you are mapping this location in the Area field.

   Enter the directory pathname in the Area field. When the location points to a directory structure within the ULTRIX/SQL installation area, enter only the name of the new directory. When the location points to a directory structure outside the ULTRIX/SQL installation area, you must specify the full pathname.

   Table 4-4 summarizes how to enter directory pathnames in the Area field.

Table 4-4: How to Enter Directory Pathnames

Location is	Sample Pathname	Enter in the Area Field
Within the ULTRIX/SQL installation	$II_DATABASE/ingres/data/new_area	new_area
Outside the ULTRIX/SQL installation	/otherplace/new_area/ingres/ data/default	/otherplace/new_area

Note

If you used **ingsetenv** to set a variable to the directory structure, you can enter the variable name instead of the pathname. See the section "What is an ULTRIX/SQL Location" for additional information.

8. Press **Return.**

9. By default, ULTRIX/SQL creates alternate locations for data files. To change this, enter **n** (no) in the Databases field.

10. To use this location for checkpoints or journals, enter **y** (yes) in the Checkpoints or Journals fields.

    You must have created subdirectories for the types of database files that use this location. For example, if you enter **y** next to checkpoints, then you already must have created a ckp subdirectory in the area to which this location is mapped.

11. Select **Save** from the menu.

12. Select **Quit** to leave **accessdb**.

You can now use the new location for a new or existing database. For instructions on creating a new database in the new location, see the section "Creating a Database in an Alternate Location" below. For instructions on extending a database to the new location, see the section "Extending a Database" below.

4.5.3 Creating a Database in an Alternate Location

To place database files in the new location, use the **createdb** command with any of the following flags followed by the name of the alternate location:

- **-d** for data files
- **-c** for checkpoints
- **-j** for journals

You can place these flags in any order. If you do not use these flags, ULTRIX/SQL places your database's data, checkpoint, and journal files in the default locations.

You must create the alternate location before you attempt to place database files in it. See the section "Creating Alternate Locations for a New or Existing Database—An Overview."

To create a database named "sample," whose checkpoints and journals are in the default location and whose data is in an alternate location named "data_location," issue this command at the operating system prompt:

```
createdb -d data_location dbname sample
```

4.5.4 Extending a Database to an Alternate Location

When you want to store a database's tables and indexes in more than one location, you must first *extend* the database to these locations. You use the **ExtendDB** option of the **accessdb** utility to do this.

You cannot use **ExtendDB** to place database files for a new database in an alternate location. In this case, you create the database in the alternate location; follow instructions given above in "Creating a Database in an Alternate Location."

To use **accessdb**, you must have ULTRIX/SQL superuser permission.

To extend a database to an alternate location:

1. Before extending a database to a *new* location, you must create the location. For instructions, see the sections "Creating a Directory Structure for a New Location" and "Creating a New Location" above.

   To extend a database to an *existing* location, start at Step 2.

2. Issue the **accessdb** command at the operating system prompt.

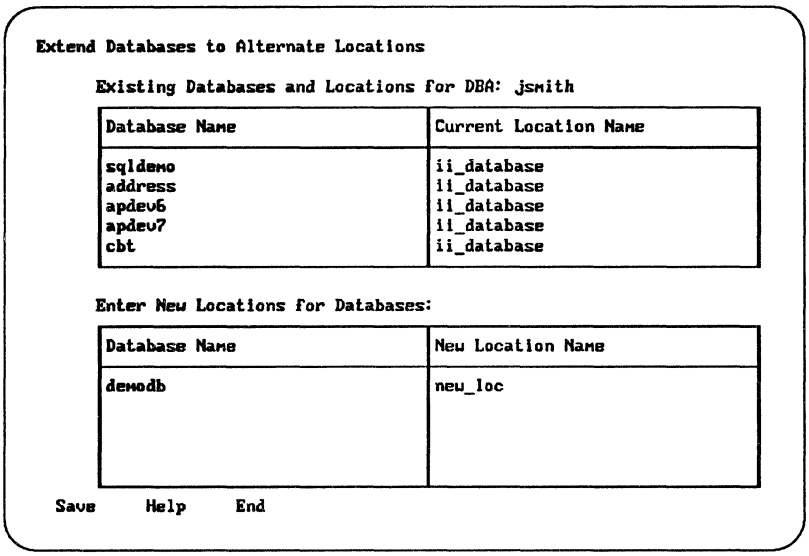
3. Select **ExtendDB** from the menu.

4. At the following prompt, enter the name of the Database Administrator (DBA) for the database you are extending:

```
DBA name:
```

5. When the Extend Databases to Alternate Locations frame appears (see Figure 4-2, "Extending Databases"), type the name of the database you want to extend in the Database Name field.

Figure 4-2: Extending Databases



6. In the New Location Name field type the name of the location to which you are extending the database.

   You *cannot* change the current locations of an existing database by typing a new locationname over an existing one. The database's system catalogs are in the location for data files that you specified when you created the database, and you cannot move them. This location, as well as any locations to which you have extended the database, is saved permanently in the database's configuration file. You only can enter additional locations for an existing database.

7. Select **Save** from the menu.

8. Select **Quit** to leave **accessdb**.

After extending a database to a new location, you can place new or existing tables and indexes in the new location. See Chapter 5 for instructions.

4.5.5 Adding Types of Files that Can Use an Existing Location

After you create a new location and use it to create a new database or extend an existing one, you cannot change the mapping between the location and the directory structure. When you use a location, ULTRIX/SQL writes the locationname and directory structure to the database's configuration file. After this has occurred, the *only* change you can make to the location is to add to the types of files (data, checkpoint, or journal) that can use the location.

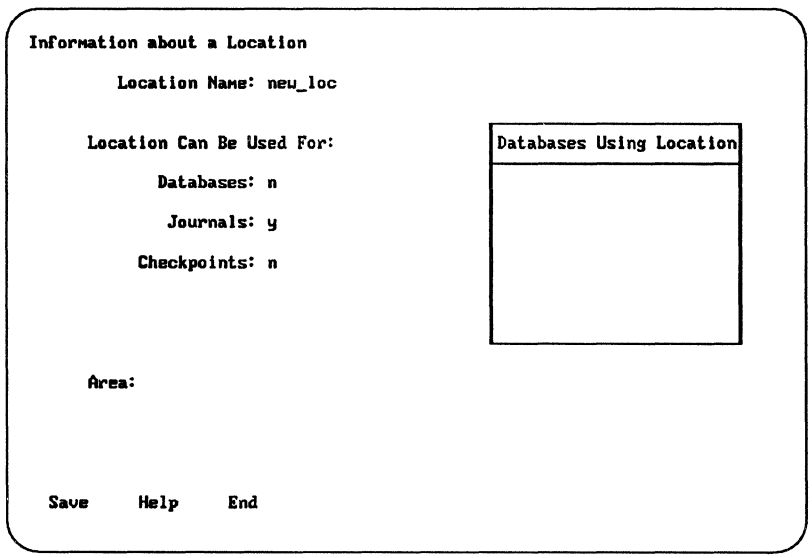
To add types of database files that can use an existing location:

1. Create subdirectories for the database files to be stored in this location. See the section "Creating an Alternate Location" above.
2. Issue the **accessdb** command at the operating system prompt.
3. Select **LocationName** from the menu.
4. At the following prompt, type the location name:

   ```
   Location name:
   ```

5. Press **Return.**
6. When the Information about a Location frame appears (see Figure 4-3, "Changing Location Information"), type y (yes) next to the additional types of files that you will store in this location.

Figure 4-3:Changing Location Information



7. Select **Save** from the menu.
8. Select **Quit** to leave accessdb.

Creating Tables and Views 5

5.1 Overview

This chapter tells you how to create tables and views. It also provides information on table limits, how to handle duplicate rows in tables, how to manipulate columns, and how to move tables to new locatons.

For instructions on creating database indexes, refer to the *ULTRIX/SQL Reference Manual.*

5.2 Creating Shareable Objects

Any user who is authorized to use a database can create private objects in that database. However, only the Database Administrator (DBA) can create shareable objects.

As DBA, you give users permission to access your tables and views. See the *ULTRIX/SQL Reference Manual* for instructions.

5.3 Creating Tables

The **create table** statement creates a new base table that is owned by the user issuing the command.

You can issue the **create table** statement:

- In an embedded SQL program
- In the ULTRIX/SQL Terminal Monitor
- In interactive ULTRIX/SQL

5.3.1 The create table Statement

To create a table, use one of the following **create table** statements:

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

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

A *with-clause* consists of the word **with** followed by a list of any number of the following items, separated by commas:

- **[no]journaling**
- **location** = (*locationname* [,*locationname*])
- **[no]duplicates**
- **structure** = *storage_structure*

For the syntax of *subselect*, see the **select** statement in the *ULTRIX/SQL Reference Manual.*

For a complete description of the **create table** statement and its parameters, see the *ULTRIX/SQL Reference Manual.*

The **create table** statement creates entries in the ULTRIX/SQL system catalogs for the table created.

5.3.2 Table Limits: Number of Rows, Number of Columns, Width of Rows

While there is no limit on the number of tables in an ULTRIX/SQL database, disk space is the limiting factor. The table below describes the limits on the contents of a table. For a discussion of each limit, see the sections immediately following the table.

Table 5-1: Table Limits

Type of Limit	Size of Limit
Number of Rows	4,194,304 multiplied by the number of rows per page
Number of Columns	127
Width of Rows	2000 bytes

5.3.2.1 Limit on the Number of Rows Stored in One Table

The maximum number of ULTRIX/SQL pages in a table is 4,194,304.

A page is a block of physical storage space. Each page is 2048 bytes in size.

Each row must fit within one ULTRIX/SQL page. The length of a row in bytes is determined by the format declared in the **create table** statement.

5.3.2.2 Limit on the Number of Columns in One Table

The number of columns allowed in one table is limited to 127. This limit applies to queries and views as well. The **select** list of a query may have up to 127 columns, and selecting from or creating a view is also limited to 127 columns. The following **select** statement could not join two tables, each with 100 columns, because the target list in a query is limited to 127 columns:

```
select * from a,b;
```

5.3.2.3 Limit on the Width of a Row

Each row can use a maximum of 2000 bytes. To derive this figure, subtract the space used by ULTRIX/SQL from the 2048 bytes on an ULTRIX/SQL page.

A **select** is also limited to 2000 bytes. Since you cannot select more than 2000 bytes in one target list, dividing a table in two and rejoining the pieces does not **select** more than 2000 bytes.

5.3.3 Duplicate Rows in Tables

A table contains duplicate rows when two or more rows are identical.

When you create a table, you can specify the handling of duplicate rows. To do this, use the **with [no]duplicates** clause of the **create table** statement. For example:

```
create table emps(
        name char(10),
        id integer
        ) with noduplicates;
```

When you create a table **with duplicates**, duplicate rows are allowed. (This is the default for ULTRIX/SQL.) When you specify a table **with noduplicates**, no duplicate rows are allowed in the table, except if the table has a **heap** storage structure. (With a **heap** structure, duplicate rows are always allowed, even if **no duplicates** was specified. See the *ULTRIX/SQL Reference Manual* for a description of storage structures.) If a user attempts to insert a duplicate row into a table where duplicate rows are not allowed, ULTRIX/SQL generates an error.

ULTRIX/SQL checks for duplicate rows only when the table has a keyed storage structure. Duplicate rows are always allowed in **heap** or **cheap** tables, since these structures are not keyed, so there is no implicit order in the table.

Depending on whether you specify **with duplicates** or **with noduplicates**, ULTRIX/SQL performs the following tasks differently:

- Adding new records into a table with the **insert** statement
- Revising existing records in a table with the **update** statement

5.3.3.1 Duplicate Rows When Adding New Records to or Modifying a Table

If a table is created **with duplicates**, duplicate rows can always be inserted.

If a table is created **with noduplicates**:

- Duplicate rows can be added if the storage structure of the table is either **heap** or **cheap**.
- Single row inserts (**insert ... values**) are silently discarded if a duplicate row occurs on a keyed structure.
- Multiple row inserts (**insert ... select**) generate an error if a duplicate row occurs on a keyed structure. The entire statement is rolled back.
- When a table is modified from a **heap** or **cheap** structure to a keyed structure, duplicates are silently eliminated.

5.3.3.2 Duplicate Rows when Bulk Copying Records in a Table

If a table is created **with duplicates**, duplicate rows can always be loaded.

If a table is created **with noduplicates**:

- Duplicate rows can be loaded if the table storage structure is **heap or cheap**.
- Duplicate rows are silently removed if the table has a keyed structure.

5.3.3.3 Duplicate Rows in Updated Tables

If a table is created **with duplicates**, duplicate rows can always be updated to duplicate other rows.

If a table is created **with noduplicates**:

- Rows can be updated to duplicate other rows if the table storage structure is either **heap** or **cheap**.
- Rows cannot be updated to duplicate other rows if the table is a keyed structure. The update is rejected and an error is generated.

In the following bulk increment update example, specified columns in all rows selected from the table are updated. For example:

```
update table1
    set info = info+1;
```

(where Info has values of 1, 2, 3, 4 ...)

If **with noduplicates** is being enforced on the table, this update fails due to duplicate rows being created, as the following explains:

The new values for the first row are prepared, changing "info" from 1 to 2. Before inserting the new values, ULTRIX/SQL checks to see if they violate any integrity constraints. Since the new value, 2, duplicates the value in an existing row, thus violating the **with noduplicates** criterion, an error is generated and the update is rolled back.

To solve this problem, you can use either of the following methods:

- Create the table **with duplicates**
- Change the table storage structure to **heap** or **cheap** before performing the update, since **with noduplicates** is not enforced on these structures

5.3.3.4 Removing Duplicate Rows from Tables

If a table was originally created **with duplicates**, ULTRIX/SQL preserves duplicate rows, even when the table is modified to another structure. If you decide that duplicate rows are not wanted, there are several techniques for removing them, as described in the following paragraphs:

- Use **with noduplicates** with a **heap** structure. This is the recommended way of handling tables that sometimes contain duplicate rows and sometimes must be unique:

  1. Create the table **with noduplicates**.
  2. Modify the table to **heap** or **cheap**.
  3. Insert the rows that contain duplicates.
  4. Modify the table to a keyed structure to remove duplicates.

- Remove duplicate rows using a temporary table. If you create a table **with duplicates** and later decide that you do not want duplicate rows, you can remove them by using a temporary table. In the example below, the table named "has_dups" has duplicate rows that are deleted using the following SQL statements:

```
create table temp as
    select distinct * from has_dups;

drop has_dups;

create table has_dups as
    select * from temp
    with noduplicates;

drop temp;
```

Note

This technique may not be appropriate for large tables because of disk space constraints.

5.3.4 Creating a Table with Journaling

When you create a table, you can enable journaling on it. See Chapter 9 for instructions.

5.3.5 Creating a Table in an Alternate Location

When you create a table, ULTRIX/SQL places it in the default location for the database unless you specify otherwise. Before you place a table in an alternate location:

- The location must exist and be set up and specified as a location that can hold data files
- The database must be extended to the alternate location

For instructions on creating a location and extending a database, see Chapter 4.

To place a table in an alternate location, use the **with location** option of the **create table** statement. For example:

```
create table emp (name char(10)) with location = (altloc1);
```

To create a table that spans multiple locations, use the **with location** option, specifying multiple *locationnames*. For example:

```
create table emp (name char(10)) with location = (altloc1,altloc2);
```

After creating a table, you can change its location. For instructions, see the section "Moving a Table to an Alternate Location" later in this chapter.

5.3.6 Additional Examples of create table Statements

The following examples of creating tables illustrate the syntax variations of the **create table** statements discussed earlier in this chapter:

- Creating a table named "employee" with no special options

```
create table employee
     (
     emp_number          integer2,
     last_name           varchar(30),
     first_name          varchar(20),
     birth_date          date,
     );
```

- Creating a table named "employee" with all **varchar** columns **not null with default**

```
create table employee
     (
     emp_number          integer2,
     last_name           varchar(30) not null with default,
     first_name          varchar(20) not null with default,
     birth_date          date,
     );
```

- Creating a table named "employee" with **integer** and **date** columns **not null not default**

```
create table employee
     (
     emp_number          integer2 not null not default,
     last_name           varchar(30),
     first_name          varchar(20),
     birth_date          date not null not default,
     );
```

- Creating a table named "employee" that spans two alternate locations

```
create table employee
     (
     emp_number          integer2,
     last_name           varchar(30),
     first_name          varchar(20),
     birth_date          date,
     )
     with location = (altloc1,altloc2);
```

- Creating a table named "employee" at an alternate location, with journaling

```
create table employee
     (
     emp_number          integer2,
     last_name           varchar(30),
     first_name          varchar(20),
     birth_date          date,
     )
     with location = (altloc1), journaling;
```

- Creating a table named "employee" with no duplicates

```
create table employee
     (
     emp_number          integer2,
     last_name           varchar(30),
     first_name          varchar(20),
     birth_date          date,
     )
     with noduplicates;
```

- Creating a table named "employee" from table "emp_trans" with a **hash** structure keyed on "emp_number"

```
create table employee
     as select * from emp_trans
     with structure = hash, key = (emp_number);
```

5.4 Manipulating Columns: Adding, Changing and Deleting

While ULTRIX/SQL does not provide a one-step method for adding or deleting a column from an existing table, the methods described here are very easy to use. You can use the same methods for changing the data type of an existing column or renaming it.

Deleting a column in a table or changing its data type does not change anything else that is dependent on the column. You need to re-create or edit all views, permits, programs, and so forth, that refer to the old column in any way. In particular, make sure views that were destroyed are re-created. Any views or indexes on a table are automatically destroyed when the table is destroyed.

5.4.1 Adding a Column

The method for adding a column to an existing table is to use the **create table ... as** statement as described below. Note that this procedure requires twice the disk storage as that required by the original table.

1. Create a temporary table containing the existing columns from the original table and the new columns. See the following section, "Data Types for New Columns," for an explanation of assigning data types to new columns.

```
create table temp as
     select test.*, varchar('  ') as newcol from test;
```

2. Drop the original table.

```
drop test;
```

3. Rename the temporary table with the name of the original table using a **create table ... as** statement with a **subselect** statement.

```
create table test as
     select * from temp;
```

4. Drop the temporary table.

```
drop temp;
```

If the column named "newcol" is to be located in the middle of the table structure, you must list all the columns individually. For example, Step 1 could look like:

```
create table temp as
     select col1,col2,varchar('      ') as newcol,
          col3,col4 from test;
```

5.4.1.1 Data Types for New Columns

Use the associated conversion function for the data type assigned to the new column. If the new column is created with no conversion function, the defaults are:

- **varchar** for character strings
- **float (float8)** for floating point numbers
- **smallint (integer2)** or **integer** (integer4) for integer numbers (depending on the size of the number)

5.4.1.2 Default Column Values

Instead of specifying a typical default value of 0 or quoted spaces (' '), you may want to substitute a particular value as the default value for the new column.

You can also specify the default value of **null** within any of the numeric conversion functions or the **date** function to initialize a column's value to **null**. This makes the column nullable. Do not use **null** as a default value for character fields, as ULTRIX/SQL attempts to create a character field of null length, which cannot be done, and returns an error.

The following table lists examples of the available data types and their associated conversion functions for creating a column with each of those data types:

Table 5-2: Data Types and Their Conversion Functions

Data Type	Conversion Function
integer1	**int1(0)**
smallint (integer2)	**int2(0)**
integer (integer4)	**int4(0)**
float4	**float4(null)**
float (float8)	**float8(0)**
money	**money(0)**
date	**date(' ')** or **date(null)**
char(1)	**char(' ')**
c1	**c(' ')**
varchar(7)	**varchar(' ')**
text(7)	**text(' ')**

If you wish to specify a very large **varchar** or character string, instead of typing the number of spaces you need between quotes use the following SQL statements:

```
create table temp2 (colx varchar(1000));
insert into temp2 (colx) values (NULL);
create table temp1 as
    select a.*, b.* from test a, temp2 b;
drop test,temp2;
create table test as
    select * from temp1;
drop temp1;
```

This also works for adding new character columns with **null** as the initial value.

5.4.2 Deleting a Column

To delete a column:

1. Create a temporary table containing all the columns except the column you wish to delete.

```
create table temp as
    select name, addr from test;
```

2. Drop the original table.

```
drop temp;
```

3. Recreate the original table from the temporary table.

```
create table test as
    select * from temp;
```

4. Drop the temporary table.

```
drop test;
```

5.4.3 Changing Data Types

If you wish to change the data type of an existing column (perhaps increasing the size of a **varchar** column or changing a **float4** column to a **float (float8)** column), you can use the same method as for adding a column. For example, to change the data type of the column Salary from **float4** to **float**, do the following:

1. Create a temporary table containing the existing columns from the original table and the new columns.

```
create table temp as
    select name, addr, float8(salary)
        as salary from test;
```

2. Drop the original table.

```
drop test;
```

3. Rename the temporary table with the name of the original table using a **create table ... as** statement with a **subselect** statement.

```
create table test as
    select * from temp;
```

4. Drop the temporary table.

```
drop temp;
```

To make a **varchar** or **char** column larger, do the following:

1. Create a temporary table and **select** the column to be made larger, as follows:

    a. Add the number of blanks necessary to yield the larger column width. (You can use the method shown earlier, in the section "Default Column Values.")

b. Use the **pad** function (which pads the column out to the declared size).

c. Use the **squeeze** function to remove the trailing blanks added by the **pad**. The **squeeze** function is important because **char** columns treat blanks as visible characters; they must be specified in an exact match query.

See the *ULTRIX/SQL Reference Manual* for information on string functions.

The following example illustrates the use of the **create table ... as** and **select** statements to make the column "addr" larger:

```
create table temp as
    select squeeze(pad(addr)+'      ') as addr
        from test;
```

2. Drop the original table (automatically dropping dependent integrities, views, permits, and index structures).

```
drop test;
```

3. Rename the temporary table with the name of the original table.

```
create table test as
    select * from temp;
```

4. Drop the temporary table.

```
drop temp;
```

5.4.4 Using the create table ... as Statement to Rename a Column

In the following example, you rename a column using the **create table ... as** statement:

1. Create a temporary table with the columns from the original table that are to be renamed.

```
create table temp as
    select name as employee, addr as address
        from test;
```

2. Drop the original table.

```
drop test;
```

3. Re-create the original table with the columns from the temporary table.

```
create table test as
    select * from temp;
```

4. Drop the temporary table:

```
drop temp;
```

Make sure you update any ULTRIX/SQL objects dependent on the old column name, such as programs. Re-create integrities, views, permits, and index structures, which are automatically deleted when the original table is dropped.

5.4.5 Additional Examples of Manipulating Columns

The following examples of modifying columns illustrate additional uses for the **create table** statements discussed earlier in this chapter. (It is a good idea to backup or duplicate the table before trying any of the following, in case something goes wrong.)

- Adding the column named "manager" with the data type of **varchar(20)** to the table named "employee"

```
create table temp as
    select emp_number,last_name,first_name,
        birth_date,job_title,department,salary,
        varchar('                    ') as manager
    from employee;
drop employee;
create table employee as
    select * from temp;
drop temp;
```

- Deleting the column named "manager" from the table named "employee"

```
create table temp as
    select emp_number,last_name,first_name,
        birth_date,job_title,department,salary
    from employee;
drop employee;
create table employee as
    select * from temp;
drop temp;
```

- Renaming the column "emp_number" to "ss_number" in the table named "employee"

```
create table temp as
    select emp_number as ss_number,
        last_name,first_name,birth_date,
        job_title,department,salary
    from employee;
drop employee;
create table employee as
    select * from temp;
drop temp;
```

- Changing the order of the columns in the table named "employee"

```
create table temp as
    select birth_date,salary,first_name,
        emp_number,job_title,department,
        last_name
    from employee;
drop employee;
create table employee
    select * from temp;
drop temp;
```

- Changing the data type for the column named "salary" from **float4** to **money**

```
create table temp as
    select emp_number,last_name,
        first_name,birth_date,job_title,
        department,money(salary)as salary
    from employee;
drop employee;
create table employee
    select * from temp;
drop temp;
```

5.5 Moving a Table to a New Location

As a database grows, it may become necessary to move some of its tables (or indexes) to an alternate location.

Perhaps a table has grown so large that you can no longer modify it at the current location, or perhaps the table needs to be distributed across multiple disk drives for optimal performance. In either case, the solution is to move the table to one or more locations. The maximum number of locations that one table can occupy is 255.

Use the **modify ... to relocate** statement whenever the number of new locations is the same as the number of old locations. Use the **modify ... to reorganize** statement only if the number of locations is changing, since there is extra overhead if you use this statement.

To move a table to an alternate location:

- You must be the table owner (or a Database Administrator impersonating the table owner in his or her database).
- The alternate location must exist.
- The database must be extended to the alternate location.

For details on defining locations and extending a database to a location, see Chapter 4.

5.5.1 Moving a Table to a Single Location

To move a table from one location to another, use the **modify** statement. The syntax for the **modify** statement is as follows:

modify *table_name* **to relocate**
with oldlocation = (area1),
newlocation = (area2);

For example, to move the table named "employees" from location "loc_1" to the location "loc_2," the table owner enters:

```
modify employees to relocate
    with oldlocation = (loc_1),
         newlocation = (loc_2);
```

For a description of the options and parameters for the **modify** statement, see the *ULTRIX/SQL Reference Manual.*

5.5.2 Moving a Table to Multiple Locations

When moving a table to multiple locations, there are two cases to keep in mind:

- Moving where the number of locations used by a table increases or decreases. This case involves table reorganization.
- Moving where the number of locations for a table remains unchanged, but one or more locations change.

5.5.2.1 Moving a Table to a Different Number of Locations

Whenever you want to change the number of locations over which a table's data is spread, use the **modify ... to reorganize** statement:

modify *tablename* **to reorganize**
with location = (*location_name*, *location_name* {,...});

For example, to reorganize the table "employees," which was contained in one location, over three locations, "loc_2," "loc_3," and "loc_4:"

```
modify employees to reorganize
    with location = (loc_2,loc_3,loc_4);
```

Keep in mind that a multiple-location table can also be reorganized to be spread across fewer locations.

5.5.2.2 Moving a Table to Different Multiple Locations

If you want to move a table that is already in multiple locations, keeping the same number of locations, use the **modify ... to relocate** statement, as follows:

modify *table_name* **to relocate**
with oldlocation = (*location_name*, *location_name* {,...}),
newlocation = (*location_name*, *location_name* {,...});

where each part of the table in **oldlocation** is moved to the corresponding **newlocation.**

For example, to relocate the multi-location table named "employees" to locations "loc_4," "loc_5," and "loc_6," use the **modify ... to relocate** statement, as follows:

```
modify employees to relocate
    with oldlocation = (loc_2,loc_3,loc_4)
         newlocation = (loc_4,loc_5,loc_6);
```

Notice that the **modify ... to relocate** statement literally moves the data from one location to another. There is no internal reorganization of the data in the table, as occurs with **modify ... to reorganize.**

Internal reorganization of the data occurs when some data in the table is moved to a different location, while other data remain in the original location. In that case, the table is reorganized to be spread across two locations.

If you just want to move part of a multi-location table to a different location, specify the locations you want to move. For example, if the "employees" table is located at "loc_4," "loc_5," and "loc_6," use the following command to move the part of the table at location "loc_5" back to location "loc_3":

```
modify employees to relocate
    with oldlocation = (loc_5),
         newlocation = (loc_3);
```

For more information on the **modify** statement, see the *ULTRIX/SQL Reference Manual.*

5.6 Creating Views

Views can be thought of as virtual tables. They do not make copies of the data, but refer to the base tables involved in the view. The main use of a view is to store a commonly used query.

Another use for views is to provide security by limiting access to specific columns in selected tables, without compromising database design.

As the name suggests, a view is a device designed primarily for selecting data. Although update rules are presented in the section "Updating Views," later in this chapter, we do not recommend updating a database by means of a view.

A view is destroyed when the base table is destroyed.

5.6.1 The create view Statement

A view is created with the **create view** statement:

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

For further details on the **create view** statement, see the *ULTRIX/SQL Reference Manual.*

5.6.2 Examples of the create view Statement

The following examples illustrate the **create view** statement:

- Creating a view called "emp_view"

```
create view emp_view
        (
        view_col1,
        view_col2,
        view_col3
        )
        as select emp_number, job_title, salary
            from employee
            where department = 'Technical Support';
```

- Creating a view called "emp_view" utilizing the check option

```
create view emp_view
        (
        view_col1,
        view_col2,
        view_col3
        )
        as select emp_number, job_title, salary
            from employee
            where department = 'Technical Support'
with check option;
```

5.6.3 Additional Information about Views

A table on which a view operates is called a *base table*. For an ULTRIX/SQL view that the DBA has created on tables he or she owns, access is controlled by permissions on the view, not the base table. By granting permissions on a view, you can allow users to read some of the data in a base table while denying them permission on other, more sensitive data.

Permissions may be granted by the DBA, using the **grant** statement.

Only the definition for the view is stored, not the data. A view definition can encompass from one to thirty-two tables. A view can be created on other views or on physical database tables.

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

If a view definition involves multiple tables, those tables should be joined together in the **where** qualification by their common columns. For example:

```
create view empmgr as
    select e.name, e.salary, d.dept, d.mgr
    from emp e, department d
    where e.dept = d.dept;
```

To use the view, refer to "empmgr" just as you would any other table in a query. For example:

```
select * from empmgr
    where dept = 'Technical Support';
```

To see the definition of a view that has been created, use the **help view** *viewname* SQL statement.

All **selects** on views are fully supported. Simply use a *view_name* in place of a *tablename* in the **select** statement.

5.6.4 Updating Views

Only a limited set of updates on views are supported because of anomalies that can occur. ULTRIX/SQL does not support updates on views that have more than one base table. If the **with check option** is turned on, no updates are allowed on columns that are in the qualification of the view definition, or on any column whose source is not a simple column name (for example, set functions or computations).

Updating is supported only if it can be guaranteed (without looking at the actual data) that the result of updating the view will be identical to that of updating the corresponding base table. This can be achieved if the rules for updating are observed before running the query.

Updating, deleting, or inserting data in a table using views is not recommended. You can update, delete or insert with ULTRIX/ SQL statements, but you must abide by the following rules. Keep in mind that an ULTRIX/SQL error message appears when you attempt an operation that is not permitted:

- Do not update columns in the qualification of a view definition. Consider the following example:

```
create view admin as
     select name, dept, sal from deptinf
     where dept = 'admin';
```

  If the view has the **with check option** turned on, the column "dept" cannot be updated through this view because it is in the qualification of the view definition.

- Do not update columns that are aggregates or computations. Consider the following example:

```
create view totsal as
     select dept, sum(sal) as tsal
     from deptinf
     group by dept;
```

  The column "tsal" cannot be updated through this view because it is an aggregate.

- Do not update columns that would cause more than one table to be updated. Consider the following example:

```
create view empinfo as
     select e.name, e.dept, e.div, d.bldg, d.floor
     from emp e, deptinf d
     where e.dept = d.dname
     and e.div = d.div;
```

  Updates to this data should be done through the underlying base tables, not through this view.

Loading and Unloading a Database 6

6.1 Overview

This chapter tells you how to unload and reload a database or selected tables with the **unloaddb** and **copydb** commands.

The **Unloaddb** and **copydb** ULTRIX/SQL commands are run from the operating system. When you run them on a database, they generate scripts that enable you to:

- Unload an entire database (**unloaddb**) to external binary or ASCII files
- Copy selected tables, or all the tables and views that you own (**copydb**) to external binary or ASCII files
- Reload the database or objects from these files

Both **unloaddb** and **copydb** are two-phase operations. In phase 1, you run the command on a database and ULTRIX/SQL creates the scripts. In phase 2, you execute the scripts to copy data out of a database and then into a database.

You must be the Database Administrator (DBA) for the database or an ULTRIX/SQL superuser to run **unloaddb**. Any user can run **copydb** to copy selected tables or all the tables and views that he or she owns in the database.

6.2 Uses for unloaddb and copydb

The **unloaddb** and **copydb** commands are most frequently used to copy or move a database or selected tables from one ULTRIX/SQL installation to another. Using these commands, you can copy or move data from one installation to another with the same or different architecture.

In addition, you can also use these commands to:

- Copy a database, or tables from one database, to another on the same installation.
- Document your database or specific tables in it. When you run **unloaddb** or **copydb**, it produces **create** scripts for the tables and views being copied. You can use these scripts as documentation for your database.

- Make static copies of your database or selected tables for the purpose of recovery.

6.3 Using unloaddb

The **unloaddb** command enables you to completely unload a database and then reload it into a new, empty database.

Unloaddb unloads all of the objects and system catalogs in your database, including:

- Tables
- Views
- Associated permissions, integrities, and indexes

To run **unloaddb**, you must be the Database Administrator for the database or a superuser impersonating the Database Administrator by using the **-u** flag.

6.3.1 unloaddb Syntax

The syntax of the **unloaddb** command is:

unloaddb [**-u***username*] [**-c**] [**-d***directory-specification*] *dbname*

For a complete description of flags and parameters for **unloaddb**, see the *ULTRIX/SQL Reference Manual.*

6.3.2 Using unloaddb to Unload and Reload a Database

The following example shows you how to use **unloaddb** to unload and reload a database on the same installation.

Since **unloaddb** creates many files, you may find it useful to create a subdirectory and then run the command while in the subdirectory. If you do this, all of the files created by **unloaddb** will be in this subdirectory.

1. At the operating system prompt, run **unloaddb** on the database you are unloading:

   unloaddb *dbname*

   This creates the **unload.ing, reload.ing, cp***DBA***.in, cp***DBA***.out, cp***USER***.in, cp***USER***.out,** and **cp***DBA***.cat** files described in the table in the following section.

2. To unload the database, execute the **unload.ing** command file:

   unload.ing

3. To reload the database into another database:

a. Create the new database:

   createdb *newdb*

b. Change the name of the database in the **reload.ing** command file to the name of the database you created in Step 3a. For example:

```
sql -s -f4F79.38 -f8F79.38 -uDBA newdb
    </path/cpDBA.in
```

c. If you want to create the new database in a different location than the original, use an editor to change the location name in the **cp***DBA***.in** or **cp***USER***.in** script to the new location name.

d. Execute the **reload.ing** command file:

   reload.ing

4. If you want to destroy the original database, use the **destroydb** command:

```
destroydb original_dbname
```

Caution

Make sure you successfully unloaded the original database and reloaded the new database before you destroy the original database.

6.3.3 How unloaddb Works

When you run **unloaddb** on a database, it creates the files listed and described in the following table.

In this table and the following examples, the characters *DBA* and *USER* are used to represent the logins of the Database Administrator and other users who own objects in the database. The actual names of these files do not contain the characters *DBA* or *USER*, but the logins of the Database Administrator and users. To ensure compatibility across all systems, ULTRIX/SQL truncates the names of the files generated by **unloaddb** to twelve characters.

Table 6-1: Files Generated by unloaddb

File	Contains
unload.ing	Operating system commands to invoke the Terminal Monitor and execute the **cp***DBA***.out** and **cp***USER***.out** files.
cp*DBA***.out**	SQL commands to copy out system catalogs and tables owned by the Database Administrator to external files.
cp*USER***.out**	SQL commands to copy out objects owned by the user (non-DBA). ULTRIX/SQL creates one **cp***USER***.out** file for each user who owns objects in the database.

File	Contains
reload.ing	Operating system commands to invoke the Terminal Monitor and execute the cp*DBA*.**in**, cp*USER*.**in**, and cp*DBA*.**cat** scripts.
cp*DBA*.**in**	SQL commands to re-create the objects owned by the Database Administrator.
cp*USER*.**in**	SQL commands to re-create the objects owned by the user. ULTRIX/SQL creates one cp*USER*.**in** file for each user who owns objects in the database.
cp*DBA*.**cat**	SQL commands to copy system catalogs into the new database.

6.3.3.1 The unload.ing and reload.ing Command Files

After you run the **unloaddb** command on a database, you execute the **unload.ing** and **reload.ing** command files to unload and reload the database.

To unload a database, you execute the **unload.ing** command file at the operating system prompt:

unload.ing

This is an example of an **unload.ing** command file:

```
sql -s -f4F79.38 -f8F79.38 -uDBA dbname </path/cpDBA.out
```

To reload a database, you execute the **reload.ing** command file at the operating system prompt:

reload.ing

This is an example of a **reload.ing** command file:

```
sql -s -f4F79.38 -f8F79.38 -uDBA dbname <path/cpDBA.in

sql -s -f4F79.38 -f8F79.38 -u'$ingres' dbname <path/cpDBA.cat
```

The parameters in these command files have the following meanings:

- **sql** is the operating system level command to invoke the Terminal Monitor.
- **-s** tells ULTRIX/SQL not to print Terminal Monitor messages when you execute these files.
- **-f4F79.38** and **-f8F79.38** are the **float4** and **float8** specifications.
- **-u***DBA* represents the login of the Database Administrator for the database.
- *dbname* is the name of the database that was unloaded and then reloaded.

Note

Never reload into the same database that you unloaded because the copy scripts will append all the unloaded data into the existing system catalogs and tables.

- */path/***cp***DBA.in* **represents the full pathname/filename for the cpDBA.in script**

6.3.3.2 The cpDBA.out and cpUSER.out Files

When you execute the **unload.ing** command file, the scripts contained in the **cp***DBA***.out** and **cp***USER***.out** scripts are invoked.

These **copy out** scripts cause ULTRIX/SQL to copy the extended system catalogs and tables into files. These files reside in the current directory or in the directory that you specified with the **-d** flag when you ran **unloaddb**. The format for the name of these files is:

- II_first_8_letters_or_less_of_the_catalog_name.$IN. (system catalogs)
- first_8_letters_or_less_of_tablename.first_3_letters_of_the_owner's_login (tables)

6.3.3.3 The cpDBA.in, cpUSER.in, cp.DBA.cat Files

When you execute the **reload.ing** command file, the **cp***DBA***.in, cp***USER***.in,** and **cp***DBA***.cat** scripts are invoked. This causes ULTRIX/SQL to :

- Re-create the database's tables in their original location

Note

If you want the tables re-created in another location, you must change the location in the **cp***DBA***.in** or **cp***USER***.in** script.

- Re-create permissions, integrities, and views
- Copy the contents of the files generated by the copy out scripts into the re-created tables and system catalogs
- Grant permission to select to public (everyone authorized to access the database) on the extended system catalogs

6.3.4 Unloading in ASCII Format

When you issue the **unloaddb** command without the **-c** flag, ULTRIX/SQL unloads the files in binary format. To unload the database in ASCII format, use the **-c** flag:

unloaddb -c *dbname*

Unloading in ASCII format allows you to:

- Move databases to an installation with a different machine architecture

- Edit the data files before reloading them into a database

Caution

If you unload the files in binary format, do *not* edit them or you will not be able to reload them.

6.3.5 Changing the Floating Point Specification

The floating point specification defaults to maximum precision and length (-f8F79.38) in the **unload.ing** and **reload.ing** command files. To reduce precision or length, you can edit this specification in these files. If you do not, zeros with no significance may consume disk space in the external data files.

6.3.6 Locking During unloaddb

When you execute the **unloaddb** command or the **unload.ing** command file, ULTRIX/SQL takes shared locks on:

- System catalogs
- Tables being reloaded

When you execute the **reload.ing** command file, ULTRIX/SQL takes exclusive locks on the user tables and system catalogs being unloaded.

6.3.7 Preventing an Inconsistent Database During unloaddb

There are two major ways that a database can become inconsistent during **unloaddb**:

- Since, by default, the database is not exclusively locked while **unloaddb** or **unload.ing** is running, a user can alter tables that are not locked during this time.
- A user can alter the database after you have executed the **unloaddb** command, but before you have executed the **unload.ing** command file.

  If a user drops a table between the time that you execute **unloaddb** and the time that you execute the **unload.ing** command file, ULTRIX/SQL generates an error message. However, if a user either adds or deletes rows from a table or adds a table, ULTRIX/SQL does not generate an error message and you will not know about the change.

To ensure the consistency of the database while it is being unloaded, lock it exclusively. To do this, add the **-l** flag to the **sql** scripts in the **unload.ing** command file:

```
sql -l -s -f4F79.38 -f8F79.38 -uDBA dbname </path/cpDBA.out

sql -l -s -f4F79.38 -f8F79.38 -uUSER dbname </path/cpUSER.out
```

6.4 Using copydb

The **copydb** command enables you (a DBA or non-DBA) to copy:

- Selected tables in a database
- All of the tables and views that *you own* in a database

Even as DBA, you will not be able to copy private tables and views in your database by using **copydb**. To do this, use **unloaddb**.

6.4.1 What copydb Copies

What **copydb** copies in the database depends on whether:

- A DBA or end user issues the command
- Table names are specified in the command

The following table shows what is copied in each situation.

Table 6-2: What copydb Copies

Who issued copydb	Table specified in command?	Command issued	What is copied
DBA	no	**copydb** *dbname*	All tables and views (owned by the DBA) and associated indexes, integrities, and permissions
DBA	yes	**copydb** *dbname* {*tablename*}	Specified tables (owned by the DBA) and associated indexes, integrities, and permissions
User (non-DBA)	no	**copydb** *dbname*	All tables and views (owned by the user who issued the command) and associated indexes and integrities
User (non-DBA)	yes	**copydb** *dbname* {*tablename*}	Specified tables (owned by the user who issued the command) and associated indexes and integrities

6.4.2 copydb Syntax

The syntax of the **copydb** command is:

copydb [**-u***username*] [**-c**] [**-d***directory-specification*] *dbname* {*tablename*}

For a complete description of the flags and parameters for this command, see the *ULTRIX/SQL Reference Manual.*

6.4.3 How copydb Works

To copy all of the tables and views that you own in a database, issue the following command at the operating system prompt:

copydb *dbname*

This produces two scripts:

- **copy.out**, which contains ULTRIX/SQL statements to copy your tables and views to operating system files
- **copy.in**, which contains ULTRIX/SQL statements to re-create your tables, views, and indexes, and associated permits and integrities, and copy the table's data from the operating system files into an ULTRIX/SQL database

To copy the tables out of the database, you run the **copy.out** script. To copy them into the same or another database, you run the **copy.in** script. For instructions, see the section "Copying Tables from One Database to Another" below.

The **copy.out** script contains a **copy** statement for each table being copied.

The **copy.in** script contains the following statements for each of your tables and views if you do not specify a particular table when running **copydb**:

- **create table**
- **create view**
- **copy** to load your tables
- **create index** to re-create secondary indexes, if any
- **grant** to re-create permissions, if any
- **create integrity** to re-create integrities, if any
- **modify** to modify any tables that did not use the **heap** structure to their original storage structures

If you specify a particular table, the **copy.in** script will not contain statements to create views.

The following example shows you the **copy.out** and **copy.in** scripts produced when you run the **copydb** command on the "personnel" database and specify that only the "emp" table should be copied. Since there are no secondary indexes or integrities for this table, the **copy.in** script does not contain statements to re-create them. The **-c** flag causes the table to be copied into a file in ASCII format. This is necessary when transferring the database from a VAX to a RISC system or from a RISC to a VAX system.

```
copydb -c personnel emp
```

It produces this copy.out script:

```
copy emp (
    name= varchar(10),
    salary= c0tab,
    dept= varchar(8),
    div= varchar(3),
    mgr= varchar(10),
    birthdate= c0tab,
    numdep= c0n1,
    n1= d1)
into '/path/emp.USR'
\p\g
```

(where "USR" represents the first 3 characters of the user's login)

And it produces this copy.in script:

```
table emp(
    name text(10) not null with default,
    salary money not null with default,
    dept text(8) not null with default,
    div text(3) not null with default,
    mgr text(10) not null with default,
    birthdate date not null with default,
    numdep smallint not null with default
)
with noduplicates, journaling,
location = (ii_database)
\p\g

copy emp (
    name= varchar(10),
    salary= c0tab,
    dept= varchar(8),
    div= varchar(3),
    mgr= varchar(10),
    birthdate= c0tab,
    numdep= c0n1,
    n1= d1)
from '/path/emp.USR'
 \p\g
```

(where "USR" represents the first 3 characters of the user's login)

```
modify emp to isam on name
with fillfactor = 80
\p\g

grant all on emp to public
\p\g
```

6.4.4 Copying Tables from One Database to Another

To copy tables from one database to another on the same installation:

1. Run **copydb** on the database that contains the tables you want to copy:

   copydb *database1 table1 table2*

This creates **copy.out** and **copy.in** scripts for "table1" and "table2" in "database1." If you had not specified tablenames, ULTRIX/SQL would have copied all the tables and views that you own in "database1."

2. To copy the tables out of the database, execute the **sql** command to run the **copy.out** script:

   sql *database1* **<copy.out**

   This creates data files that the **copy.in** script uses to load the tables into the new database.

3. To copy the tables into another database, execute the **sql** command to run the **copy.in** script. Use the name of the database that you are copying the table(s) into:

   sql *database2* **<copy.in**

6.4.5 Copying in ASCII Format

When you issue the **copydb** command without the **-c** flag, ULTRIX/SQL copies the files in binary format. To copy the files in ASCII format, use the **-c** flag:

copydb -c *dbname*

Copying the files in ASCII format allows you to:

- Move the tables you own to an installation with a different machine architecture
- Edit the data files before copying them into a database

Caution

If you copy the files in binary format, do *not* edit them; doing so will cause your data to be corrupted.

6.4.6 Changing the Floating Point Specification

When you execute the **sql** command to run the **copy.out** and **copy.in** scripts, the floating point specification defaults to 10 positions with 3 to the right of the decimal point. To change this specification, use the **-f** flag with the **sql** command when you run the **copy.out** and **copy.in** scripts. See the *ULTRIX/SQL Reference Manual* for a description of the floating point (**-f**) flag parameters that you can use with the **sql** command.

6.4.7 copydb and Locking

When you execute the **copydb** command or the **copy.out** script, ULTRIX/SQL takes shared locks on the tables being copied.

When you execute the **copy.in** script, ULTRIX/SQL takes exclusive locks on the tables being copied.

6.4.8 Preventing an Inconsistent Database During copydb

There are two major ways that the database can become inconsistent during **copydb**:

- Since ULTRIX/SQL takes shared locks on the tables being copied while **copydb** or **copy.out** is running, a user can alter the tables that are not locked during this time.
- A user can alter the tables being copied after you run the **copy.out** script, but before you have run the **copy.in** script.

  If a user drops a table between the time that you execute **copydb** and run the **copy.out** script, ULTRIX/SQL generates an error message. However, if a user adds or deletes rows from a table during this time, ULTRIX/SQL does not generate an error message, and you will not know about the changes.

To ensure the consistency of the tables being copied, exclusively lock them while they are being copied. To do this, add the **-l** flag to the **sql** command used with the **copy.out** script:

sql -l *dbname* **<copy.out**

6.5 Moving/Copying Databases and Tables Between ULTRIX VAX and ULTRIX RISC Systems

One of the most common uses for **unloaddb** and **copydb** is copying or moving databases from one installation to another. This section provides instructions for copying/moving an ULTRIX/SQL database or selected tables from an ULTRIX VAX system to an ULTRIX RISC system or from an ULTRIX RISC system to an ULTRIX VAX system.

If you plan to destroy the original database or tables after moving them to a new installation, be sure you have a good operating system backup of them first.

6.5.1 Copying/Moving a Database Between ULTRIX VAX and ULTRIX RISC Systems

To perform the following steps, you must be either:

- The DBA for the database being copied or moved
- An ULTRIX/SQL superuser impersonating the DBA by using the **-u** flag with the **unloaddb** command

1. Issue the following command at the operating system prompt of the system where the database is currently located:

   unloaddb -c *dbname*

   This creates scripts to unload the database into ASCII files.

2. Execute the **unload.ing** command file by issuing the following command at the ULTRIX system prompt:

   unload.ing

   This unloads the database into files into the current directory or into the directory that you specified when you ran **unloaddb**.

3. Use a file transfer utility to transfer the files created by the **unloaddb** command and the **unload.ing** command file to the target system.

4. Log on to the other ULTRIX system with a differing architecture.

5. Create a new database on the other system using the **createdb** command at the ULTRIX system prompt.

   Be sure to use a database name that does not currently exist. If you do not, ULTRIX/SQL returns an error message.

6. Edit the **reload.ing** file:

   a. Change the name of the database to the name you specified in Step 5.

   b. Convert the ULTRIX directory path to the correct path for the new system.

7. Edit the **cp***DBA*.**in,** **cp***USER*.**in,** and **cp***DBA*.**cat** files:

   a. Convert the ULTRIX directory path to the correct path for the new system.

   b. Convert the directory path specification for the iiud.scr file to the correct path for the new system.

8. Issue the following command at the ULTRIX prompt on the new system:

   reload.ing

9. Back up the new database. See Chapter 9 for instructions.

10. If you want to destroy the original database:

    a. Make sure you have successfully moved the database to the new installation before destroying the original one.

    b. Return to the original ULTRIX system.

    c. Use the **destroydb** command at the ULTRIX system prompt to destroy the original database:

       destroydb *original_dbname*

6.5.2 Copying/Moving Tables Between ULTRIX VAX and ULTRIX RISC Systems

Users can copy or move tables *that they own* from one system to another. To do this:

1. Issue the following command at the operating system prompt:

   copydb -c *dbname table1 table2*

   This command creates scripts to unload the tables into ASCII files.

2. Run the **copy.out** script by issuing the following command at the system prompt:

   sql *dbname* **<copy.out**

3. Use a file transfer utility to transfer the files created by the **copydb** command and the **copy.out** script to the target system.

4. Log on to the other ULTRIX system.

5. Edit the **copy.in** script to convert the directory path to the correct path on the other system.

6. Run the **copy.in** script by issuing the following command at the ULTRIX prompt on the new system:

   sql *dbname* **<copy.in**

7. If you want to destroy the original table:

   a. Make sure you have successfully moved the table to the new installation before destroying the original one.

   b. Return to the original ULTRIX installation.

   c. Execute the **drop** statement in the ULTRIX/SQL Terminal Monitor or in interactive SQL:

      drop *table1, table2*;

6.6 Avoiding Problems with unloaddb and copydb

To avoid problems with **unloaddb** and **copydb**:

- Be careful not to make syntax errors if you edit the scripts generated by these commands.

- If the data files are in binary format, do not edit them. This will cause problems when ULTRIX/SQL tries to read the files into the new database.

- Do not try to change the copy scripts to skip fields in a binary file because of the difficulty of counting bytes in a binary file.

  If you need to change the copy scripts, copy or unload the data in ASCII format by using the **-c** flag with either **copydb** or **unloaddb**. See the sections "Unloading in ASCII Format" or "Copying in ASCII Format" for additional information.

- Be sure to copy or unload data in ASCII format (using the **-c** flag) if you are transferring data between ULTRIX VAX and ULTRIX RISC systems.

Populating Tables 7

7.1 Overview

This chapter presents methods for loading data into tables using the **copy** statement. It discusses considerations for ensuring the data's integrity and validity, and ways to avoid data errors. The chapter also explores advanced use of the **copy** statement including its use in unloading and reloading data.

7.2 Methods of Loading Data into Tables

There are several ways to get data into a database. Which method you choose depends on several considerations:

- Where is the data coming from?
- How much of it is there?
- What level of action is needed?

The methods for loading data into tables include:

- Using the SQL **copy** statement if the data is already in a file, or if it will be put in one by a program. The **copy** statement loads large quantities of data quickly from files and is flexible in dealing with various record formats. This statement is discussed in the sections titled "Using the Copy Statement" and "Advanced Use of the Copy Statement" in this chapter.
- Writing an embedded language program for data entry
- Using the SQL **insert** statement in the ULTRIX/SQL Terminal Monitor or in interactive SQL. This choice is appropriate for entering a very small amount of test data.

7.3 Using the copy Statement

This section tells you how to use the **copy** statement to load data into tables.

7.3.1 The copy Statement

To load data into a table or unload data back into a file, use the following **copy** statement:

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

A *with-clause* consists of the word **with** followed by a comma-separated list of any number of the following items:

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

For details on the **copy** statement, see the *ULTRIX/SQL Reference Manual.*

7.3.2 The copy Statement and Locking

The **copy statement** takes an exclusive lock on a table while data is being copied into the table and a shared lock on the table while data is being copied out of it.

7.3.3 Specifying a Filename

The **copy** statement is not able to expand ~$HOME or to recognize the ULTRIX variables set in your environment. Using these variables to specify a pathname for the **copy** statement does not work.

Always enclose the pathname in single quotation marks.

The following **copy** statements work:

```
copy emp() from '/usr/fred/emp.lis';
copy emp() from 'subdir/emp.lis';
```

The following **copy** statements do not work:

```
copy emp () from '~fred/emp.lis';
copy emp () from '$HOME/emp.lis';
```

If you are copying files from your current directory, you can specify just the name of the file rather than a full path.

7.3.4 Speed of the copy Statement

The **copy** statement is the fastest way to load a table with data. It executes most quickly when the table into which it copies is an empty, unjournaled, unindexed **heap**, since no transaction logging is necessary. If you are loading a substantial amount of data, it is much faster to load a table as a **heap** and modify it to another storage structure after it is loaded.

Avoid copying large amounts of data into a table with an **isam** structure. Doing so could result in overflow chains that can degrade performance. Copying data to a **hash** structure can be quick if enough space is pre-allotted for the new data (using the **fillfactor** or **minpages** clauses with the **modify** statement).

7.3.5 The copy Statement and Nulls

When you copy data from a table to a file or from a file to a table, the **with null** clause of the **copy** statement allows you to substitute a value for nulls.

When you use variable length data types, you *must* replace the nulls with some string that represents nulls; for example:

```
copy table personnel (name=c20,
    salary=c0 with null ('N/A'),
    title = c0 with null ('xxx'),
    dummy=d0nl)
    into 'pers.data';
```

After executing this statement, the pers.data file will contain "N/A" for each null salary and "xxx" for each null title.

With other data formats, you are not required to substitute a value for nulls. However, if you do not, you will get an ULTRIX/SQL binary data file containing unprintable characters.

When substituting a value for nulls, the value:

- Must not be one that occurs in your data
- Must be compatible with the format of the field in the file:
    - Character formats require quoted values
    - Numeric formats require unquoted numeric values

Do not use a null if you are copying to a numeric format. The file will not accept an actual ASCII null character or the word **null** for numeric format.

For a complete explanation of the **copy** statement, see the *ULTRIX/SQL Reference Manual.*

7.3.6 Invalid Data Errors

When copying from file to table, the following problems are the most frequent causes for errors:

- Using data that does not match the specified format
- Miscounting fixed-length field widths
- Neglecting the "nl" delimiter in the **copy** statement format
- Omitting delimiters between fields in the **copy** statement format
- Including too many delimiters in the **copy** statement format

7.3.6.1 Invalid Data

If you try to load invalid data into a field, ULTRIX/SQL rejects the row. For example, suppose you had the following record in a file:

```
559-58-2543,31-feb-1945,Weir,100000.00,Executive
```

Since February has only twenty-eight or twenty-nine days, ULTRIX/SQL will return errors.

7.3.6.2 Miscounting Fixed-Length Field Widths

If the widths of fixed-length fields are not correct, the **copy** statement may try to include data in a field that it cannot convert to the appropriate format. For example, here is a record to be copied into a table which will generate an error:

```
554-39-2699 1-oct-1943 Quinn 28000.00 Assistant
```

The **copy** statement is:

```
copy table personnel (ssno=c20,
    birthdate=c12,name=c11,salary=c9,
    title=c0n1)
    from 'pers.data';
```

Because "c20" was specified incorrectly for the "ssno" field, the **copy** statement includes part of the birthdate in the value for the Social Security number. The **copy** statement then reads "943 Quinn " for the birthdate. For information on variable-length field width formats, refer to the *ULTRIX/SQL Reference Manual.*

7.3.6.3 Neglecting the "nl" Delimiter in the copy Statement

When using fixed-length specifications in the **copy** statement, the **Newline** (nl) character at the end of the record must be accounted for. This is an example with an error:

```
554-39-2699 1-oct-1943 Quinn 28000.00 Programmer
335-12-1452 23-jun-1931 Smith 79000.00 Sr Analyst

copy table personnel (ssno=c20,birthdate=c12,
    name=c11,salary=c9,title=c10)
    from 'pers.data';
```

The format specified for the "title" field is **c10**, which does not account for the **Newline** character. As a result, ULTRIX/SQL returns an error.

The **Newline** characters are converted to blanks, as specified in the warning message. The extra characters force the **copy** statement to begin reading a third record that ends abnormally with an unexpected end of file.

7.3.6.4 Omitting Delimiters Between Fields

Omitting delimiters between fields in the data file causes another error message. For example, the first record below has no delimiter between the employee's name and her salary:

```
123-45-6789,1-jan-1960,Garcia33000.00,Programmer
246-80-1357,2-jan-1960,Smith,43000.00,Coder
```

The **copy** statement

```
copy table personnel (ssno=c0,birthdate=c0,name=c0,salary=c0,
        title=c0n1)
    from 'pers.data';
```

results in an error.

The **copy** statement attempts to read the value "Programmer" into the "salary" field, resulting in the error messages.

7.3.6.5 Including Too Many Delimiters

Accidental inclusion of too many delimiters in the data file occurs most often when the comma is used as a delimiter and also appears in the data. For example, in the first record, the salary value contains a comma:

```
123-45-6789,1-jan-1960,Garcia,33,000.00,Programmer
246-80-1357,2-jan-1960,Smith,43000.00,Coder
```

The following **copy** statement will cause ULTRIX/SQL to return errors:

```
copy table personnel (ssno=c0,birthdate=c0,
    name=c0,salary=c0,title=c0)
    from 'pers.data';
```

The **copy** statement reads the value 33 into the "salary" field, 000.00 into the "title" field, and "Programmer" as into the "ssno" field. It then attempts to read "246-80-1357" as the "birthdate," which produces the error.

If the **copy** statement had specified **title=c0n1,** there would have been no error messages. The **copy** statement would have again read 33 as the "salary," but would have read "000.00,Programmer" as the "title," since it is only looking, at that point, for a **Newline** at the end of the "title." The **copy** statement would have read the second record correctly. Although no error message would have been generated, the "title" field for one record would have been incorrect.

7.3.7 What To Do If You Are Having Trouble Loading Data

If you are having trouble loading your data into the designated tables:

- Take two rows from the data file you are using and work with the **copy** statement until you succeed with those two rows. Check the database table to be sure the results are what you require.
- Use the options available in the **copy** statement syntax to continue on error and log records that fail for later examination.

If attempts to load data from binary files are not successful:

- Check to make sure the data comes from exactly the same machine architecture. Integer and floating point formats can differ between machines.

- Pick apart your data column by column, using dummy delimiters for the rest of the row until the **copy** statement succeeds.
- If all else fails, get an ASCII copy of the data so you can correct errors.

7.4 Data Integrity and Validity

There are two ways to ensure that your data is consistent and error-free:

- Integrity constraints, imposed by the SQL **create integrity** statement
- Program customized data checking with embedded SQL

Sometimes the data you want to put in an ULTRIX/SQL table is already in a file, ready to be loaded with the **copy** statement. Because **copy from** is a high-performance utility, it is not subject to integrity constraints defined by the **create integrity** statement. Errors in the data must be handled, nonetheless.

These situations always terminate the **copy** statement:

- When ULTRIX/SQL cannot read the input file when doing a **copy ... from**
- When ULTRIX/SQL cannot write the output file when doing a **copy ... into**
- When server errors occur

The following errors do not necessarily terminate the **copy** statement:

- Data type errors in which non-numeric characters are scattered throughout ASCII (character) representations of numeric fields. This is the most common type of error.
- Errors unrelated to data type; for example, a syntax error where the "name" column fails to start with a capital letter, when it is required to do so.

7.4.1 Using the copy Statement's with Clause to Control Error Handling

You can control the way the **copy** statement handles errors by specifying options in its **with** clause.

The options allow you to decide:

- Whether the **copy** should stop or continue when an error occurs
- Whether the rows already copied should be undone, assuming the **copy** will not continue
- Whether the **copy** should stop when a certain number of errors occur
- Whether the records not copied should be logged into a log file for analysis

If you do not specify any of the options, the default is to stop copying when the first error is encountered and to rollback or undo whatever was copied up to that point.

Typical combinations of options used in the **copy** statement follow:

- If you want to load data from a file, but stop and rollback if more than ten errors occur:

```
copy table personnel (name=c0,dept=c0n1)
    from 'pers.data'
    with on_error=terminate,error_count=10,
    rollback = enabled;
```

- If you want to load data from a file, throwing away any invalid rows and continuing to process good ones:

```
copy table personnel (name=c0,salary=c0n1)
    from 'file.data'
    with on_error=continue,rollback=disabled;
```

- If you want to load data from a file, logging any bad rows in badrows.data:

```
copy table personnel (name=c0,address=c0n1)
    from 'hr.data'
    with on_error=continue,rollback=disabled,
    log = 'badrows.data';
```

For more information about the **with clause**, see the **copy** statement in the *ULTRIX/SQL Reference Manual.*

The **copy** statement will *write lock* a table while data is being copied into it, but will only *read lock* a table while data is being copied out of it. For a discussion of locking, see Chapter 8.

7.4.2 Checking for Data Type Errors

As the example below illustrates, using the **with clause** of the **copy** statement when loading tables can be very helpful if the data files contain data type errors:

- The **copy** statement continues to copy good records into the database table even after encountering errors.

- Those records with errors are logged in a log file that you can examine and use to correct the records in the data files.

Otherwise, the **copy** statement aborts with error messages indicating type-mismatch.

Suppose, for example, that you want to load some personnel data from a file with the following record format:

```
social_security_number,name,jobtitle,salary
```

Although "salary" is a numeric item, it is stored in character format.

Suppose further that in the ULTRIX/SQL "personnel" table, into which you want to load the data, "salary" is the name of a column of type **money.**

The **copy** statement handles this conversion easily, provided each "salary" value consists only of digits and (optionally) a dollar sign, minus sign, and/or decimal point. Nevertheless, there might be spurious characters in the "salary" data, such as those contained in the following two records:

```
123-45-6789,Garcia,Programmer,3Z000
559-58-2543,Weir,Executive,I80000
```

The data contains a Z instead of a 3, an uppercase I instead of 1, and three letter Os instead of zeros. If the **copy** statement encountered this data while trying to load a numeric table column, it would abort with an error message, indicating type-mismatch.

While ULTRIX/SQL provides the row number where it failed, if there are many problem rows, visually scanning a large amount of raw data is tedious and seldom productive.

You can solve this problem by using the **with** clause of the **copy** command to log invalid records. The **copy** statement would look like this:

```
copy table personnel (ssnum=c0,name=c0,
        jobtitle=c0,salary=c0n1)
    from 'pers.data' with on_error=continue,
        log='logfilename';
```

The **with clause** has two parts in this case:

- The first part, **on_error=continue,** causes the **copy** command to continue copying data after a data format error has occurred.

- The second part, **log='logfilename,'** causes the records with data type errors to be logged in the log file you name. When the **copy** statement is done, you can examine the log file and use it to correct errors in the original data. All good records will have been copied into the database table.

Once you have corrected the errors in the data logged in the log file, you can run the **copy** statement again using the log file to insert the corrected data into the "personnel" table.

7.4.3 Checking for Integrity Errors Unrelated to Data Type

Other kinds of errors unrelated to data type, such as syntax errors, can exist in the data to be loaded. With the **create integrity on ...** is statement, you can do the following, provided you own the table (or, with ULTRIX/SQL superuser status, impersonate the table owner):

- Impose on the table integrity constraints that the data must satisfy.

  Note

  The **copy from** statement, designed for high-speed data loading, ignores these constraints.

- Catch any exceptions as errors

Consider again the example in which the **copy** statement loads data into the ULTRIX/SQL Personnel table. (See the section, "Checking for Data Type Errors" above.) Now suppose the values in the "name" column of that table must begin with an uppercase letter. Suppose also that the record for the executive contained the name "weir" misspelled with a lowercase "w," but was, nevertheless, added to the table since the **copy** ignored that constraint.

The following procedure, used prior to loading the data, would enable you to catch the exception as an error, correct it, and impose the integrity constraint so that future updates with that kind of error would not occur. If you already had an integrity definition, it was ignored, so remove it with the **drop integrity** statement and re-create the integrity after copying.

1. Execute the **create integrity on ...** is command to impose the integrity constraint as follows:

```
create integrity on personnel
is name like '\[A-Z\]%' escape '\';
```

   If the search condition is not true for every existing row in the table when the command is issued, an error message appears and the integrity constraint is rejected.

2. Find the incorrect rows with the following query:

```
select name from personnel where name not
like '\[A-Z\]%' escape '\';
```

3. Use interactive SQL to browse the "personnel" table, spot errors, and correct them.

4. Repeat Step 1 to impose the integrity constraint.

7.5 Unloading and Reloading Data

The **copy** statement is bi-directional; in addition to loading data into a table, it also unloads data from a table. A common use of **copy** is to unload a table for backup to tape or for transfer to another database. In either case, there is the possibility of reloading the data into a database later.

7.5.1 Bulk Copy

The quickest form of the **copy** statement to code and execute is called a *bulk copy*. This statement has the following format:

copy table *tablename* () **into** *'output filename'*;

The bulk **copy** statement copies all table data, byte for byte, into a file. It places no delimiters between fields or between records. It performs no type conversions; thus, all data items retain the type they had in the table. *If any columns have a type other than* **char** or **varchar**, *they are not readable as characters in the output file.*

Unloading data in binary format for backup may be inconvenient if you need to inspect the data later. On the other hand, the binary format may be better for maintaining floating point accuracy.

Another concern when you use the bulk copy is that it is designed for reloading data into tables that have exactly the same record layout as those from which the data was unloaded. Those tables must be on a machine with the same architecture as that from which they were unloaded. The ULTRIX/SQL utilities designed for quick unloading and reloading of tables or databases, **copydb** and **unloaddb**, both use the bulk form of **copy** by default. They automate the process so you can easily re-create and reload tables of identical layout.

You cannot use unloaded binary data to load, at a later time, tables that have a different column order, number of columns, data types, or table structure. For that reason, you may wish to have your external files in character format, with **newlines** separating records, so that you can inspect them.

7.5.2 Unloading in Readable Format

There are several important things to remember to include in the **copy** statement when unloading your data into readable files.

- You must specify each column in the target list; omitting the column names is not acceptable.
- You must describe the external file format, *not* the table column format, in the target list of the **copy** statement.
- There are two major types of copying for unloading into files: one for fixed-length fields and another for variable-length fields.

7.5.2.1 Unloading into Files with Fixed-Length Fields

Fixed-length fields can use automatic or explicit specification of the field length, as indicated below:

- If you use **c0** or **char(0)**, character fields are printed using their full length. Other types of columns are formatted using the field formats **integer1**, **smallint**, **integer**, **float4**, **date**, and **money**..
- If you use **cN** or **char(N)**, the **copy** statement prints exactly N characters. Excess characters are discarded and shorter columns are padded with blanks.
- Use of the **text(N)** format has the same results as use of the **cN** or **char(N)**, except that the padding is with null bytes. The **varchar(N)** format prints exactly N characters with a leading length indicator in binary format, discarding any excess characters. Shorter columns are padded with null bytes.

Variable length data items are written to a file by the **copy** statement as follows:

- **text(1)**, ... , **text(2000)**—no leading length indicator is written to the output file with the data item

- **text(0)** delim—writes a two-byte binary length, followed by the data, followed by the delimiter
- **varchar(0)**—an ASCII length is written preceding **varchar(0)**

7.5.2.2 Unloading Into Files with Variable-Length Fields

There are two types of variable-length fields, those with a leading length indicator and those that end with a delimiter:

- The **varchar(0)** format prints the data in the column with a leading length indicator.
- The **text(0)** format prints the data in the column with an optional trailing delimiter and trims trailing blanks.

The **c** and **char** data type formats trim trailing blanks, whereas the **text** and **varchar** data type formats retain them.

7.6 Advanced Use of the copy Statement

You can find extensive documentation on the **copy** statement in the *ULTRIX/SQL Reference Manual.* Be sure to review this material before using the **copy** statement. The current section discusses variations on the use of this command.

The examples in this section use a database illustrated by the following table:

Table 7-1: Example Database

Table Name	Column Name	Data Type
Header	Orderno	Integer2
	Date	Date
	Suppno	Integer2
	Status	Char(1)
Suppinfo	Suppno	Integer2
	Suppinfo	Char(35)
Detail	Orderno	Integer2
	Invno	Integer2
	Quan	Integer2
Iteminfo	Invno	Integer2
	Descript	Char(20)
Princeinfo	Invno	Integer2
	Suppno	Integer2
	Catno	Integer2
	Price	Money

7.6.1 How to Use Multiple Files to Populate Multiple Database Tables

Suppose that the information for the database described in the last section was stored in data files outside ULTRIX/SQL, and that those files, file1 and file2, have the record formats shown below:

```
orderno,date,suppno,suppinfo,status

orderno,invno,catno,descript,price,quan
```

The **copy** statement can be used to load the data from these files into a five-table database. Assume that the files are entirely in ASCII character format, with fields of varying length terminated by commas, except for the last field, which is terminated by a **Newline**. The following copy statement loads the "header" table from the first file:

```
copy table header (orderno=c0comma, date=
    c0comma,suppno=c0comma,
    dummy=d0comma,status=c0nl)
    from 'file1';
```

Each column of the "header" table is copied from a variable-length character field in the file. All columns except the last are delimited by a comma; the last column is delimited by a **Newline**.

Specification of the delimiter, although included in the statement, is not needed because the **copy** statement looks for the first comma, tab, or **Newline** as the field delimiter by default.

The notation **d0,** used in place of **c0,** tells the **copy** statement to ignore the variable-length field in that position in the file, rather than copy it. The **copy** statement ignores the column name (in this case "dummy") associated with the field described as **d** format.

Note

The order of column names in a **copy** statement *must* correspond to the order in the file from which the data comes. The data type indicates the data type in the file, not in the table. The order of columns in the table does not have to be the same as the order of fields in the file.

7.6.1.1 Loading a Table from Multiple Files

Loading the "priceinfo" table presents special difficulties. The **copy** statement can read only one file at a time, but the data needed to load the table resides in two files.

The solution to this kind of problem varies with the file and table designs in any particular situation. In general, a good solution is to copy from the file containing most of the data into a temporary table containing as many columns of information as needed to complete the rows of the final table.

To load data from the files into the "priceinfo" table, do the following:

1. Create a temporary table named "pricetemp" that contains the "orderno" column in addition to all the columns of the "priceinfo" table:

```
create table pricetemp (orderno integer2,
    invno integer2,
    suppno integer2,
    catno, integer2 price money);
```

The reason you need to add the "orderno" column to the temporary table is that it enables you to join the temporary table to the "header" table to get the supplier number for each row.

2. Copy the data from the order detail file into the "pricetemp" table.

```
copy table pricetemp (orderno=c0,invno=c0,
    catno=c0,dummy=d0,price=c0,dummy=d0)
    from 'file2';
```

3. Insert into the "priceinfo" table all rows that result from joining the "pricetemp" table to the "header" table.

```
insert into priceinfo
        (invno,suppno,catno,price)
    select p.invno,h.suppno,p.catno,p.price
    from header h, pricetemp p
    where p.orderno=h.orderno;
```

7.6.1.2 Multi-Line File Records

Another feature of the **copy** statement is that it can read multi-line records from a file into a single row in a table. For instance, suppose that for viewing convenience, the detail file was formatted so that each record took three lines. That file might look like this:

```
1, 5173
10179A, No.2 Rainbow Pencils
0.29
1, 5175
73122Z, 1986 Rainbow Calendars
4.90
```

You could load these values into the "pricetemp" table with the following **copy** statement:

```
copy table pricetemp (orderno=c0comma,
    invno=c0nl,catno=c0comma,
    descript=d0nl,price=c0nl)
    from 'file2';
```

It does not matter that **Newlines** have been substituted for commas as delimiters within each record. The only requirement is that the data fields be uniform in number and order, the same as for single-line records.

7.6.2 Loading Fixed-Length and Binary Records

The **copy** statement can also load data from fixed-length records without any delimiters within or between the data. In addition, numeric items in the file may be stored in true binary format. For example, the value 256 may be stored in a 2-byte integer instead of 3 characters.

For example, if the order header file is laid out as follows:

```
orderno  date  suppno  suppinfo  status
```

With field formats of:

2-byte integer, 8 characters, 2-byte integer, 35 characters, 1 character

The following **copy** statement loads the "header" table:

```
copy table header (orderno=integer2,date=c8,
        suppno=integer2,dummy=c35,
        status=c1)
    from 'file1';
```

It is also possible to copy data from files that contain both fixed- and variable-length fields.

ULTRIX/SQL Locking 8

8.1 Overview

ULTRIX/SQL is a shared system; it allows many users to access the same database at the same time. To ensure that their transactions do not interfere with each other, ULTRIX/SQL uses locking.

In any database management system with multiple users, there is a tradeoff between consistency and concurrency. Ideally, you want all of your users to be able to access the database at the same time (concurrency) and you want all of the data to be accurate (consistency).

To ensure consistency, ULTRIX/SQL automatically places locks on tables or pages of tables when users submit queries to change or select data.

But locking can cause some users to wait for other users to finish their queries, if not handled carefully. The purpose of this chapter is to provide you, the Database Administrator (DBA), with the knowledge you need to maximize concurrency, while minimizing the time that users wait for locks to release.

This chapter describes the following:

- Types and levels of locks
- How ULTRIX/SQL locking works
- How long locks are held
- Locking examples
- Locking parameters
- How to change locking parameters with **set lockmode**
- How to avoid deadlock situations
- How to monitor locks
- How to improve concurrency

8.2 The ULTRIX/SQL Locking System

The ULTRIX/SQL locking system, which uses shared memory and semaphores, controls all locks. The shared memory and semaphores used by your ULTRIX/SQL installation are configured in the operating system when your ULTRIX kernel is configured.

Your ULTRIX/SQL System Administrator initially configures the locking and logging system when ULTRIX/SQL is initialized. The parameters that he or she selects are installation-wide. These parameters may be changed, but only by the ULTRIX/SQL System Administrator. For a complete discussion of these locking parameters, see the *ULTRIX/SQL Operations Guide.*

The ULTRIX/SQL locking system controls locking by:

- Managing and queuing lock requests
- Detecting deadlock situations

For a discussion of deadlock, see the section "Avoiding Deadlock" later in this chapter.

8.3 Lock Types

ULTRIX/SQL locks can be of various *types*. The term *type* refers to how powerful the lock is; for example, whether it prevents other users from reading the data it protects, or only from changing that data. ULTRIX/SQL uses the five types of lock that are described in the following table.

Table 8-1: ULTRIX/SQL Locks

Type of Lock	Description
X	*Exclusive* locks or *write* locks. Only one user may hold an exclusive lock on a resource at any given time. (A resource is a data page or table on which a lock can be taken. A user of this lock is called a *writer.*)
S	*Shared* locks or *read* locks. Multiple users may hold shared locks on the same resource at the same time. No user can update an object which is read locked. A user of this lock is called a *reader.*
IX, IS	*Intended exclusive* and *intended shared* locks. Whenever ULTRIX/SQL takes an exclusive (X) or shared (S) lock on a page within a table, ULTRIX/SQL takes an intended exclusive (IX) or intended shared (IS) lock on the table. ULTRIX/SQL uses these locks to determine whether it is possible to take a lock on the table. These locks do not actually lock users out of the table.
SIX	*Shared intended exclusive* locks. The ULTRIX/SQL database management system server's buffer manager uses these locks to manage its page cache.
N	*Null locks.* An ULTRIX/SQL lock that does not block any action but preserves the number in the value block of the lock.

8.4 Locking Levels

ULTRIX/SQL locks can be of various *levels*. The *level* of a lock refers to the scope of the resource on which the lock is taken; for example, an entire table or a single page. (A page is a block of physical storage space; it is the smallest unit on which a user can take a lock. Each page is 2048 bytes in size.)

Although there are many other levels of lock besides the page and table, these are the only two levels subject to the user's control. For details on user-controlled locking, see the section "User-Controlled Locking" later in this chapter.

8.5 How ULTRIX/SQL Locking Works

When a user issues a query, implicit requests for locks are made. In response, ULTRIX/SQL considers the following factors to determine what kind of lock, if any, to take on behalf of that user:

- Are there any locks available?
- Does the query involve reading or changing data?
- Should the locking level be for a page or an entire table?
- What kind of resource is affected by the query?
- Do any other users already hold locks on the affected resource?

8.5.1 How ULTRIX/SQL Determines Whether a Lock Is Available

When your ULTRIX/SQL System Administrator configures the ULTRIX/SQL logging and locking system, he or she sets the total number of available locks. As each lock is taken, a counter is decremented by one to reflect the number of locks still available. If ULTRIX/SQL receives a lock request after all available locks have been taken, the request cannot be satisfied until a lock is freed. If this happens frequently, your ULTRIX/SQL System Administrator can reconfigure the maximum number of locks. For details on configuring the number of ULTRIX/SQL locks, see the *ULTRIX/SQL Operations Guide*.

8.5.2 How ULTRIX/SQL Determines the Appropriate Type of Lock

When a user issues a **select** statement to access some data, a *shared lock* is required, since the user only wants to read the resulting data.

When a user issues a statement that writes to the database, such as an **update**, **insert**, or **delete** statement, ULTRIX/SQL knows that an *exclusive lock* is required.

8.5.3 How ULTRIX/SQL Determines the Appropriate Level of Lock

The default locking level for ULTRIX/SQL is the page level. Page-level locks are taken whenever possible.

Using the **set lockmode** statement, you can change parameters that determine how ULTRIX/SQL handles locking during a session. One of these is the **maxlocks** parameter. Using **maxlocks,** you can reset the maximum number of page-level locks ULTRIX/SQL can take per table per query before it escalates to table-level locking.

The default for **maxlocks** is ten. For a discussion of the **set lockmode** statement, see the section "User-Controlled Locking" below. For more details on the **maxlocks** parameter, see the section "Changing maxlocks" below.

If the ULTRIX/SQL optimizer facility (OPF) estimates that a query will be touching more than the number of pages to which **maxlocks** is set, the query will start out with a table-level lock. This saves the overhead of accumulating multiple page-level locks. For example, on a query that is not restrictive or does not use a key to locate affected records, where the optimizer decides that scanning the entire table is required, ULTRIX/SQL takes a table-level lock at the beginning of query execution.

If the optimizer estimates that no more than **maxlocks** pages will be needed, ULTRIX/SQL takes page-level locks.

If a query involves a single table with only a primary key, the optimizer is not used and page-level locks are taken on the appropriate pages of the table.

If the number of pages in a table on which ULTRIX/SQL holds locks reaches **maxlocks** during the processing of a query, ULTRIX/SQL escalates to table-level locks to complete the query, by doing the following:

- Stops accumulating page-level locks
- Escalates to a table-level lock
- Drops all page-level locks it has accumulated

ULTRIX/SQL also escalates to table-level locks in an attempt to complete a query if the transaction or the installation has run out of locks. To avoid this situation in the future, the ULTRIX/SQL System Administrator can shut down the ULTRIX/SQL installation and reconfigure the locking system for the number of locks allowed per transaction. The locking system cannot be reconfigured in the middle of processing a query if the transaction or the installation has run out of locks. ULTRIX/SQL simply returns an error and backs out the transaction.

The following table describes what type and level of lock ULTRIX/SQL invokes by default when a query is issued. For more information on the optimizer facility and ULTRIX/SQL lock levels, refer to the *ULTRIX/SQL Operations Guide.*

Table 8-2: ULTRIX/SQL Locking Levels

ULTRIX/SQL Operation	Comment	Lock Type	Lock Level
select	For each table involved in the **select**	IS and S	Table lock Page lock(s) on pages in table
	If query touches > **maxlocks** pages, ULTRIX/SQL takes a shared table lock rather than page locks	S	Table lock
update, insert, or **delete**	Table **update, insert,** or **delete**	IX and X	Table lock Page lock(s) on pages in table
	If query touches > **maxlocks** pages	X	Table lock
	For other tables used in query but not being changed	S	See lock for **select** statement
Create index	On base table	X	Table lock
	On index	X	Table lock
Create table	On table	X	Table lock
Drop table	On table	X	Table lock

8.5.4 How ULTRIX/SQL Determines Whether to Take a Lock

Whether ULTRIX/SQL can take a lock depends on whether any other user holds a lock on that resource, and if so, what type of lock the other user holds. If no locks are available or another user already holds an exclusive lock on the resource in question, ULTRIX/SQL cannot take a new lock. The user will have to wait.

As mentioned earlier, ULTRIX/SQL looks for intended shared and intended exclusive locks on a table to determine quickly whether a table-level lock can be taken on that table, as follows:

- An intended shared lock on the table means that a shared lock has been taken on at least one page of the table; nevertheless, a shared lock, if available, can still be taken at either the page or table level.
- An intended exclusive lock on the table means that an exclusive lock has been taken on at least one page of the table; no table-level lock can be taken on the table *on behalf of another user* until the current exclusive page-level locks have been released.

The default setting of the ULTRIX/SQL locking system ensures that no user can read data being changed and no user can change data being read. However, users may read data that is being read by other users. This means that:

- ULTRIX/SQL can take an S lock for User2 on resource R provided User1 does not already hold an X lock on R.
- ULTRIX/SQL can take an X lock on resource R for User2 provided User1 does not already hold an S or X lock on R.
- ULTRIX/SQL can take an S lock on resource R for User2 even if User1 already holds an S lock on R.

This default strategy is adequate for most situations. When it is not, you can establish a different strategy using the **set lockmode** statement. For details, see the sections "Uses for the set lockmode Statement" and "Considerations when Setting readlock to nolock" later in this chapter.

8.5.5 How Long Locks Are Held

In ULTRIX/SQL, locks are held until a transaction is committed. When a transaction is committed, its results are written to the database and all locks accumulated during the transaction are dropped. Transactions are committed:

- By issuing the **commit** statement after one or more SQL queries.
- By ending your ULTRIX/SQL session. This is not recommended.

After a **commit** is executed, the current transaction is terminated and you are in a new transaction as soon as the next SQL statement is issued.

Note

If you do not issue the **commit** statement during a session, all locks taken on the resources affected by your queries are held until your session ends. Your entire session will be treated like one transaction and will cause concurrency problems.

8.5.6 A Single-User Locking Example

In the following example, a user issues an SQL query to read data on the employee named Jeff from the table named "emp." The user then issues a **commit** statement so that ULTRIX/SQL releases the locks taken on the table after executing the query.

1. The user issues a **select** statement followed by a **commit** statement:

```
select * from emp where name = 'Jeff';
commit;
```

2. ULTRIX/SQL takes an IS lock on the "emp" table and a page-level S lock on the second page of that table.

Since the query is restrictive (only the row specified in the **where** clause is to be retrieved), and the table itself has an **isam** structure indexed on the "name" column, ULTRIX/SQL does not have to scan the entire table. Thus, an S on the entire table is not necessary. ULTRIX/SQL can use the index to go directly to the row for Jeff. For this reason, an IS lock on the table and an S lock on the page containing the row for Jeff are sufficient.

3. ULTRIX/SQL retrieves the Jeff row.

4. ULTRIX/SQL releases the locks held.

If the user, upon retrieving the row for Jeff, were to decide that he wanted to update that record, he could issue an **update** statement before issuing the **commit**.

If so, and there are no other shared locks on the page containing the Jeff row, ULTRIX/SQL would escalate the shared lock taken on the page to an exclusive lock and the IS lock on the table to an IX lock.

8.5.7 A Multi-User Locking Example

In the previous section, the example illustrated the use of locking when a single user initiates a transaction.

The next example illustrates how ULTRIX/SQL uses locks when multiple users run queries against the same tables. (A brief summary of the example precedes two figures showing what locks are taken on the tables involved. A scenario with the details of both users' transactions follows.)

User1 initiates a multi-query transaction to update the salary of each employee in the Techsup department to 30000. Shortly thereafter, User2 issues a query to read the salary and floor of the employee named Dan. Both users end their transactions with a **commit** statement.

Both users' transactions affect the tables named "emp" and "dept." The former table is keyed on "name," with a secondary index on "dept," while the latter is keyed on "dname." Because of the way these tables are indexed, page-level locking is used, since only a few pages within the tables need to be accessed. If the query were likely to span many pages (either because there were many employees in the Techsup department or because the row for each employee was very wide), table-level locking might have been required.

The following tables illustrate the first four pages of the "emp" table and the "dept" tables, with the locks ULTRIX/SQL takes on behalf of both users.

Table 8-3: The "Emp" Table

Page	Name	Salary	Deptno
1	Andy	55000	9
	Candy	50000	6
	Dan	25000	7
2	Ed	20000	2
	Fred	20000	8
	Jeff	35000	4
3	Kevin	40000	3
	Lenny	30000	6
	Marty	25000	8
4	Penny	50000	9
	Susan	20000	1
	Tami	15000	6

Table 8-4: The "Dept" Table

Page	Deptno	Dname	Floor
1	1	Accting	5
	2	Admin	4
	3	Develop	4
2	4	Mgr	3
	5	Prod	2
	6	Sales	3
3	7	Shipping	2
	8	Techsup	1
4	9	VP	5
	10	WP	5

Table 8-5: Locks Taken on the "Emp" and "Dept" Tables

Table	Page	Locks Taken for User 1	Locks Taken for User2
Emp Table	Entire table	IX	IS
	1		S
	2	X	
	3	X	
	4		
Dept Table	Entire table	IS	IS
	1		
	2		
	3	S	S
	4		

The scenario looks like this:

1. User1 issues the following statements:

```
update emp set salary = 30000 where deptno in
     (select deptno from dept
     where dname = 'Techsup');
commit;
```

2. User2 issues the following statements:

```
select e.salary, d.floor from emp e, dept d
     where d.deptno = e.dept
     and e.name = 'Dan';
commit;
```

3. On behalf of User1, ULTRIX/SQL takes the following locks:

   a. An IS lock on the "dept" table

   b. An S lock on the third page of the "dept" table where the record for the Techsup department is located

   and starts executing the subselect statement to retrieve the Techsup record.

4. On behalf of User2, ULTRIX/SQL takes the following locks:

   a. An IS lock on the "emp" table

   b. An S lock on the first page of the "emp" table where the record for the employee named Dan is located

c. An IS lock on the "dept" table

d. An S lock on the third page of the "dept" table where the Shipping record is located

and starts executing the **select** statement to retrieve the salary for employee Dan from the "emp" table and the floor on which he works from the "dept" table, using the "dname" value "Shipping."

5. On behalf of User1, ULTRIX/SQL takes the following locks:

   a. An IX lock on the "emp" table

   b. An IX on the second and third page of the "emp" table where the updates will be made and begins executing the **update** statement, setting the value of the "salary" column for all employees in the Techsup department to 30000.

6. On behalf of User2, ULTRIX/SQL executes the **commit** statement, dropping all locks held on her behalf.

7. On behalf of User1, ULTRIX/SQL executes the **commit** statement, committing all updates and dropping all locks held on his behalf.

8.5.8 Waiting for Locks

Now look at what would have happened in the previous example if User2 had issued instead the following query:

```
select * from emp;
commit;
```

Because ULTRIX/SQL took an IX lock on the second and third pages of the "emp" table on behalf of User1, User2 would have waited to retrieve all the values from the "emp" table until User1 completed his query and released all locks. The reason is that if one user is updating at least one page in a table, no other user can read the entire table.

In this simple case, the waiting time would have been negligible, but had User1 issued a complicated update on a large number of rows in the "emp" table, User2 might have waited a long time.

To prevent delays, there are several approaches:

- Keep all transactions as short as possible. This is the best approach. (Note that the **set lockmode** statement is not allowed in an open transaction.)

  For details on the **set lockmode** statement, see the section "Uses for the set lockmode Statement" later in this chapter.

- Use the **set lockmode** statement with **readlock = nolock** (when possible) to avoid having to wait for read locks.

- Use the **set lockmode** parameter **timeout** to indicate how long to wait for a lock. (The default is to wait forever.) ULTRIX/SQL returns an error when the timeout is reached. The current statement (not the transaction) is aborted. It is up to the program to trap the error.

8.6 User-Controlled Locking

User-controlled locking is available in ULTRIX/SQL through the **set lockmode** option of the **set** statement. The syntax for the set **lockmode** statement is as follows:

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]**

Note

You cannot issue the **set lockmode** statement within a transaction. You can issue it as the first statement in a session or after a **commit** statement.

For a complete description of the set lockmode statement, see the *ULTRIX/SQL Reference Manual.*

8.6.1 How to Use the set lockmode Statement

There are several ways to use the set lockmode statement:

- Type the statement in the interactive SQL Terminal Monitor.
- Include it in an embedded ULTRIX/SQL program as you would any other ULTRIX/SQL statement. This affects only the session of the user running the program.
- Specify the ULTRIX/SQL set statement with any of the following environment variables, each of which has a different scope.

Note

When typed interactively or included in an embedded SQL program, **set lockmode** overrides the locking parameters specified by these environment variables.

- ING_SET

  This affects all users.

- ING_SET_*DBNAME*

  This affects only users in the specified database.

- *DBNAME*_SQL_INIT

  This affects only the SQL session.

- ING_SYSTEM_SET

  This affects all ULTRIX/SQL users.

The **set** statements pointed to by the environment variables are executed whenever an ULTRIX/SQL user connects to the server. The environment variables can be set in the ULTRIX/SQL symbol table as installation-wide variables. They may also be set locally in each user's environment. (See the *ULTRIX/SQL Operations Guide* for details, including the syntax, for each of these variables.)

For example, to specify **readlock** = **nolock** for a user's session with the **set lockmode** option using **ING_SET**, use:

C shell:

```
setenv ING_SET "set lockmode session where readlock=nolock";
```

Bourne shell:

```
ING_SET="set lockmode session where readlock=nolock"

export ING_SET
```

8.6.2 Uses for the set lockmode statement

With the **set lockmode** statement, you can:

- Set locking parameters for a particular table. For example:

  ```
  set lockmode on emp where readlock = nolock;
  ```

- Set locking parameters for the duration of an ULTRIX/SQL session. For example:

  ```
  set lockmode session where readlock = nolock;
  ```

For more discussion on the use of locking parameters, see the following sections in this chapter:

- "Changing the Locking Level"
- "Changing Maxlocks"
- "Setting a Timeout"
- "Setting Readlock to Nolock"

8.6.3 Changing the Locking Level

By default, ULTRIX/SQL locks at the page level. Page-level locks will be taken whenever possible. If the optimizer estimates that not more than **maxlocks** pages per table, per transaction need be locked, ULTRIX/SQL takes page-level locks; otherwise, ULTRIX/SQL takes table-level locks.

To specify table-level locking, use the following:

```
set lockmode session where level = table;
```

There are several situations where the page locking default might not be appropriate:

- If a query is not restrictive or does not make use of the key for a table, scanning the entire table is required. In that case, ULTRIX/SQL will automatically start with a table-level lock; you don't need to specify it.

- If there are a number of unavoidable overflow pages, it might be preferable to set table-level locking for reasons of efficiency. Refer to the *ULTRIX/SQL Reference Manual* for a discussion on storage structures.

- If, during execution of a query, ULTRIX/SQL must lock more than **maxlocks** pages on a table (often because of an overflow chain), ULTRIX/SQL will escalate to a table-level lock. ULTRIX/SQL then drops the page locks that have been accumulated. Since accumulating page locks when a table lock was really necessary is a waste of resources, table locking from the outset would be preferable.

- If multiple users are concurrently running queries to change data, *deadlock* can occur.

  Deadlock occurs when multiple users are waiting for each other to release locks so they can escalate locking to a higher level, and none of them can complete their transactions. For example, if User1 is holding exclusive page locks on "table1" and trying to get an exclusive table lock on "table1" while User2 is holding exclusive page locks on "table1" and trying to get an exclusive table lock on "table1," neither user can get the table lock until all the page locks are released. This results in deadlock. If ULTRIX/SQL had taken an exclusive table lock on behalf of one of the users at the onset, this deadlock could have been avoided.

For a discussion on deadlock, see the section "Avoiding Deadlock" later in this chapter.

8.6.4 Changing Maxlocks

By default, ULTRIX/SQL escalates to a table-level lock after locking ten (**maxlocks**) pages within a table. But lock escalation can lead to deadlock.

Set the number of locks taken before escalation occurs to a number higher than ten. For example:

```
set lockmode on emp where maxlocks = 20;
```

changes the number of pages in the "emp" table that can be locked from 10 to 20.

This requires more locking system resources. The ULTRIX/SQL lock limit may have to be raised. But, this can provide better concurrency in a table with unavoidable overflow chains. Be forewarned that **btree** tables use twice as many locks as **isam** and **hash** tables, since index pages, leaf pages (pages containing a unique key and tuple identifiers for every row in the table), and data pages are all locked.

8.6.5 Setting a Timeout

By default, ULTRIX/SQL waits for a lock indefinitely. (The default is timeout 0—that is, no timeout.) For instance, if User1 is running a report and User2 attempts to insert into the table used for the report, the insert will appear to "hang" while waiting for a lock. User2 will wait for the lock, no matter how long that takes.

If you aren't certain how long users in your database will have to wait for locks, you may want to limit the period of time (expressed in seconds) a user waits for a lock. This can be done using the **timeout** option of the **set lockmode** statement.

To set the time limit for which a lock request should remain pending to thirty seconds, issue the following statement:

```
set lockmode session where timeout = 30;
```

If a lock is not granted in the amount of time specified, the query is rolled back (not the entire transaction) and ULTRIX/SQL returns an error. This error may be trapped and handled in embedded SQL programs.

If you embed a timeout in an application, timeout must be carefully handled by the application. If timeout occurs while processing a statement in a multi-query transaction, only the query that timed out is rolled back. The entire transaction is not rolled back unless the user specifies **rollback**. Previous statements are not backed out and the next query in the transaction is processed. For this reason, the application must contain code to trap the error, roll back the entire transaction, and retry it, starting with the first query.

8.6.6 Setting Readlock

Pages locked for reading are normally locked with a shared lock. A shared lock on a page does not prevent multiple users from reading that data concurrently. However, a user trying to change data on the locked page will have to wait for all shared locks to be released, since changing data requires exclusive locks.

8.6.6.1 Setting Readlock to Nolock

By setting the lockmode on the table to **readlock = nolock**, one user can read data while another user modifies data. To set **readlock = nolock**, issue the following statement:

```
set lockmode session where readlock = nolock;
```

Using **readlock = nolock** does not affect any query that updates, deletes, or inserts rows in a table.

But if this is done, the user retrieving data may end up with inconsistent data if changes were made by others during the retrieval. Before using this strategy, consider how important the consistency and accuracy of the data is.

To ensure that a **readlock = nolock** user is reading accurate pages, ULTRIX/SQL uses a table control lock. A *table control lock* is a special lock on a table that is implemented prior to normal table locking. This lock ensures that no reader of any type (including those for whom **readlock=nolock**) can look at a table:

- When it is being loaded using the **copy** or **the create table ... as select** statement
- When its schema is being created or changed, using either the **create table, create index, create view, create integrity, drop,** or **modify** statement

8.6.6.2 Considerations when Setting Readlock to Nolock

Whereas readlocks prevent other users from obtaining writelocks and slow down their performance, setting **readlock = nolock** can improve concurrent performance and reduce the possibility of deadlocks.

To summarize, setting **readlock = nolock** is beneficial when updates, inserts, or deletes to a table involve isolated operations on single rows rather than multi-query transactions or iterative operations on multiple rows.

In contrast, setting **readlock = nolock** is undesirable when using multi-query transactions that include updates that reference data from other tables. Here one cannot guarantee the consistency of data between the tables with no readlocks.

8.7 Avoiding Deadlock

Deadlock is different from waiting for locks, but the two situations are often confused. Deadlock occurs when User1 is being blocked from continuing his transaction by User2, and at the same time User2 is being blocked from continuing her transaction by User1. Both are unable to proceed until one transaction is aborted, allowing the other to continue.

8.7.1 A Deadlock Situation

The next example depicts a situation that produces deadlock:

User1 initiates a multi-query transaction to read all the data from the "employee" table and then insert a record with the department name "Sales" into the "department" table. Shortly after, User2 initiates a multi-query transaction to read all the data from the "department" table and then to insert a record with the employee name "Bill" into the "employee" table.

The scenario goes like this:

1. User1 issues the following statement:

```
select * from emp;
```

2. On behalf of User1, ULTRIX/SQL takes a shared lock on the "emp" table and starts executing his **select** statement.

3. User2 issues the following statement:

```
select * from dept;
```

4. On behalf of User2, ULTRIX/SQL takes a shared lock on the "dept" table and starts executing her **select** statement.

5. User1 enters the following statement:

```
insert into dept (dname) values 'Sales'
```

6. User2 enters the following statement:

```
insert into emp (name) values 'Bill'
```

7. ULTRIX/SQL blocks User1's implicit request for an IX lock on the "dept" table because of the shared lock already on the table.

8. ULTRIX/SQL blocks User2's implicit request for an IX lock on the "emp" table because of the shared lock already on the table.

User1 must wait for User2 to release his shared lock on the "department" table, but this will never happen unless User2 can finish her transaction. To finish her transaction, she needs to obtain an exclusive lock on the "employee" table, which she cannot get until User1 releases his shared lock on it.

Thus, both users are waiting for each other, so neither would ever finish his or her transaction if it were not for the fact that periodically, the ULTRIX/SQL locking system checks on all processes waiting for locks to make sure deadlock has not occurred.

When a deadlock is discovered, the locking system aborts the transaction of one of the users, allowing the other user to continue. The user whose transaction was aborted receives an ULTRIX/SQL error.

All his queries from the last **commit** up to the point of the deadlock will be backed out, not just the last query. For this reason, the ULTRIX/SQL deadlock error should be trapped and the transaction retried in an application program.

Deadlock should not happen frequently if transactions are concise and no lock escalation occurs (either page to table or shared lock to exclusive lock.) A deadlock is always logged to the error log.

8.7.2 Deadlock in Single-Query Transactions

Because ULTRIX/SQL uses page-level locking, accumulating locks one by one, deadlock can happen even when single-query transactions are being used. At least two users must be accessing the database, since deadlock will not occur when running single-user, and at least one user must be modifying rows. Deadlock generally does not occur when ULTRIX/SQL is only executing **select** statements, since shared locks do not conflict with each other.

It is possible for deadlock to occur during a single-query transaction when:

- Different access paths to pages in the base table are used
- Lock escalation occurs
- A **select** statement opens a cursor without the **for read only** option

Lock escalation deadlock can be caused by any of the following:

- Read to shared lock to exclusive lock
- Overflow chains
- System lock limits exceeded
- **maxlocks** exceeded
- Btree index splits

8.7.2.1 When Different Access Paths Are Used

Multiple users updating table data using different access paths can cause single-query deadlocks. Consider the following example in which the "emp" table has an **isam** structure indexed on "name" and a **hash** secondary index on "empno."

1. User1, accessing the "emp" table through the secondary index, takes an exclusive lock on the fourth page of the table.
2. User2, accessing the "emp" table by way of the **isam** key on the base table, takes an exclusive lock on the third page.
3. User1 needs an exclusive lock on the third page, but cannot get one because User2 already has a lock on it.
4. User2 needs an exclusive lock on the fourth page, but cannot get one because User1 already has a lock on it.

8.7.2.2 When Lock Escalation Occurs

When multiple users are updating a table and lock escalation occurs, they can deadlock. This is probably caused by one of three things:

- A user has run into a lock limit and can only continue by escalating to table-level locks.
- More than **maxlocks** pages need to be locked during the course of a query.
- There are long overflow chains.

If you are running into locking limits, either raise these limits or shorten the multi-query transactions.

If lock escalation deadlock is occurring, consider using the **set lockmode** statement to force table-level locking on the table or to increase **maxlocks**.

To understand how lock escalation can produce deadlock, consider the following example in which two users are trying to insert into the same table that has many overflow pages:

User1 tries to insert a record, and because of the long overflow chain exclusively locks ten pages. Meanwhile, User2 also tries to insert a record and takes locks down another overflow chain.

During the processing of User1's query, the transaction reaches **maxlocks** pages and needs to escalate to an exclusive table-level lock. But, because User2 is still exclusively locking pages in the table, User1's request must wait.

User2's query also needs to lock more than **maxlocks** pages, so a request is made to escalate to an exclusive table-level lock. User2's request is also blocked, because User1 is holding exclusive page-level locks.

Deadlock occurs in that neither user can proceed because each is blocking the other.

When many concurrent users are inserting to a small **btree** table, index splits are likely to occur and deadlock can happen as the locking level in the index must be escalated to exclusive.

8.7.2.3 When Locking Occurs down an Overflow Chain

Tables with many overflow pages can cause locking problems because ULTRIX/SQL must search all overflow pages. It locks each page individually and keeps locks all the way down the overflow chain. Escalation to table-level locking while locking an overflow chain can cause deadlock in heavily concurrent environments, as well as slow down the query processing time.

If you have a table with many unavoidable overflow pages, you may wish to use the **set lockmode** statement to do the following:

- Establish table-level locking as the default for that table
- Increase **maxlocks**

8.7.3 Handling Deadlock in Applications

The following program sample checks for deadlock after every query. If deadlock happens when a query is issued and that query is the victim, the entire transaction of which the failed query was a part aborts, the application is sent back to line 100, and the transaction is retried until it successfully completes without deadlock. This sample program is written in embedded FORTRAN.

```
    EXEC SQL INCLUDE SQLCA;
    EXEC SQL WHENEVER SQLERROR GOTO 100;
    EXEC SQL WHENEVER NOT FOUND CONTINUE;
    EXEC SQL BEGIN DECLARE SECTION;
    integer*4 x;
    EXEC SQL END DECLARE SECTION;
    x = 0;
10  continue;

    EXEC SQL SELECT MAX(empno) INTO :x
         FROM emp;
```

```
    EXEC SQL INSERT INTO emp (empno)
         VALUES (:x + 1);

    EXEC SQL COMMIT;

    goto 200;

100 if (sqlcode .eq. -4700) then goto 10

200
    .
    .
    .
```

In this example, if deadlock occurs, there is no need to issue the **rollback** statement, since ULTRIX/SQL has already aborted the transaction.

If deadlock was not checked for and handled, and the **select** statement to retrieve the maximum employee number failed with a deadlock, the program flow would continue and the next statement issued, the **insert** statement, would complete.

```
INSERT INTO emp (empno)
VALUES (:x + 1)
```

Since the **select** statement did not complete, this statement would insert the value 1, which very probably is not the maximum employee number. If there is no check for deadlock in this situation and deadlock does occur, incorrect data is inserted into the "emp" table.

Programmers using embedded SQL should note that the default behavior is to continue when an error occurs and that errors are not printed by default. To handle an error, the programmer needs to specify the desired behavior in the **whenever sqlerror** statement or to check the **sqlca.sqlcode** manually after each SQL statement.

8.8 Monitoring Locking

If you are having problems with concurrency, you can use one of the following tools to see how ULTRIX/SQL places and releases locks:

- The **set lock_trace** statement, which displays the locks that ULTRIX/SQL places each time a user executes a query.
- The **lockstat** utility, which provides a summary listing and a "snapshot" of *all* the locking activity in your installation. For details on the **lockstat** utility, see Appendix B in this manual.

8.8.1 set lock_trace

You can use the **set lock_trace** option of the **set** statement as you use the **set lockmode** statement. For details on the **set lockmode** statement, see the section "How to Use the set lockmode Statement" earlier in this chapter.

The **set lock_trace** statement enables you to start and stop lock tracing at any time during a session.

To start tracing locks, issue the following statement:

```
set lock_trace;
```

To stop tracing locks, issue the following statement:

```
set nolock_trace;
```

When you use **set lock_trace** during a session, you receive a list of the locks placed during the execution of your query. This list is displayed on your terminal with the results of your query.

You should *only* use **set lock_trace** as a debugging or tracing tool. Since **set lock_trace** output is not guaranteed to remain the same across releases, you should not base applications on it.

8.8.2 Environment Variables

You can use any of the following environment variables to start the **lock_trace** option:

- **ING_SET**
- **ING_SET**_*DBNAME*
- *DBNAME*_**SQL_INIT**
- **ING_SYSTEM_SET**

(See the *ULTRIX/SQL Operations Guide* for the information about these variables.)

If you use any of the environment variables to start the **set lock_trace** option, you receive output for ULTRIX/SQL user interface startup queries as well as for query language statements.

To set environment variables in your local environment, issue a statement like the following at the operating system prompt, specifying the environmental variable you want to set:

C shell:

```
setenv ING_SET 'set lock_trace'
```

Bourne shell:

```
ING_SET='set lock_trace'
export ING_SET
```

8.8.3 lock_trace Output

Following is some sample output from the set **lock_trace** statement. Table 8-5 provides an explanation of the information in this listing. To help you locate the appropriate explanation, column numbers have been added to the sample output.

```
  1       2    3    4          5          6
LOCK:    PAGE  PHYS Mode: S   Timeout: 0 Key: (inv,iiattribute,21)
UNLOCK: PAGE   Key: (inv,iiattribute,21)
LOCK:    PAGE  PHYS Mode: S   Timeout: 0 Key: (inv,iiindex,11)
UNLOCK: PAGE   Key: (inv,iiindex,11)
LOCK:    TABLE PHYS Mode: IS  Timeout: 0 Key: (inv,parts)
LOCK:    PAGE       Mode: S   Timeout: 0 Key: (inv,parts,0)
```

The following table provides an explanation of each column in the **set lock_trace** output.

Table 8-5: Explanation of set lock_trace Output

Column	Name	Explanation
1	LOCK, UNLOCK	A lock was taken or released.
2	PAGE, TABLE	Page or table level lock.
3	PHYS	Physical lock, which is either a table-level lock or a lock internal to ULTRIX/SQL.
	BLANK	Indicates that lock is on a user table. The lock is held until the transaction is committed.
	Key	Lockname assigned by ULTRIX/SQL. See Column 6 for a description of the lock name.
4	Mode	S = shared lock X = exclusive lock IS = intended shareed lock IX = intended exclusive lock N = null lock SIX = shared intended exclusive lock
5	Timeout	Default timeout (0) or timeout set with **set lockmode**.
6	Key	Lock name assigned by ULTRIX/SQL. The lock name consists of the database name, table name, and page number.

8.8.4 A lock_trace Example

This section shows the **set lock_trace** output for the following transaction:

```
select * from parts where color = 'red';
update parts set price = 10 where partno = 11;
commit;
```

The **set lock_trace** output for this transaction appears below. It contains line numbers, which are explained on the page following the **set lock_trace** output.

Note

If you run the same query several times, you may begin to receive less **set lock_trace** output. This is because ULTRIX/SQL is caching the system catalogs information.

lock_trace Output:

```
select * from parts where color = 'red'

+------+-------------+------+-----------+-----+
|partno|partname     |color |wt         |price|
 ------+-------------+------+-----------+------
*************************************************************
(1) LOCK:   PAGE  PHYS Mode: S  Timeout: 0 Key: (inv,iirelation,11)
(2) LOCK:   PAGE  PHYS Mode: S  Timeout: 0 Key: (inv,iiattribute,21)
(3) UNLOCK: PAGE  Key: (inv,iiattribute,21)
(4) LOCK:   PAGE  PHYS Mode: S  Timeout:   Key: (inv,iiattribute,19)
(5) UNLOCK: PAGE  Key: (inv,iiattribute,19)
(6) UNLOCK: PAGE  Key: (inv,iirelation,11)
(7) LOCK:   PAGE  PHYS Mode: S  Timeout: 0 Key: (inv,iiindex,11)
(8) UNLOCK: PAGE  Key: (inv,iiindex,11)
(9) LOCK:   TABLE PHYS Mode: IS Timeout: 0 Key: (inv,parts)
(10)LOCK:   PAGE      Mode: S  Timeout: 0 Key: (inv,parts,0)
*************************************************************
|1A12 |Truck      |red      |    290.000|  $16.00|
|1B5  |Bean bag   |red      |    198.000|  $18.00|
|20G  |Laser      |red      |    165.000|  $15.80|
+-----+-----------+---------+-----------+--------+
   (3 rows)

update parts set price = 10 where partno = 20G
*********************************************
(11)LOCK:   TABLE PHYS Mode: IX  Timeout: 0 Key: (inv,parts)
(12)LOCK:   PAGE      Mode: X   Timeout: 0 Key: (inv,parts,0)
*************************************************************
(1 row)

commit
*************************************************************
(13)UNLOCK: ALL     Tran-id: 092903CB0A7
*************************************************************
End of Request
```

Explanation of set lock_trace output :

(1) ULTRIX/SQL took a shared physical lock on page 11 of the **iirelation** table of the "inv" (inventory) database.

Remember that physical locks are internal to ULTRIX/SQL and are released as soon as possible.

(2) ULTRIX/SQL took a shared physical lock on page 21 of the **iiattribute** table of the "inv" database.

(3) ULTRIX/SQL released the lock on page 21 of the **iiattribute** table.

(4) ULTRIX/SQL took a shared physical lock on page 19 of the **iiattribute** table of the "inv" database.

(5) ULTRIX/SQL released the lock on page 19 of the **iiattribute** table.

(6) ULTRIX/SQL released the lock on page 11 of the **iirelation** table.

(7) ULTRIX/SQL took a shared physical lock on page 11 of the **iiindex** table of the "inv" database.

(8) ULTRIX/SQL released the lock on page 11 of the **iiindex** table.

(9) ULTRIX/SQL took an intended shared lock on the "parts" table.

This is the first lock in this example that was placed on a user table.

(10) ULTRIX/SQL took a shared lock on page 0 of the "parts" table.

(11) ULTRIX/SQL took an intended exclusive lock on the "parts" table.

(12) ULTRIX/SQL took an exclusive lock on page 0 of the "parts" table.

(13) ULTRIX/SQL released all locks taken during this transaction.

8.9 Improving Concurrency

When evaluating performance in a situation where multiple users are performing selects, updates, inserts, and deletes on the same set of tables concurrently, consider the following:

- If there are no users changing data in a set of tables, multiple, concurrent users reading data have no performance problems associated with concurrency. There are no deadlock problems either.

  Once a writer mixes with the readers of a table, concurrency is affected, since the writer will be acquiring exclusive write locks on pages or on entire tables. Deadlocks may occur, degrading performance for users who are "backed out" from the deadlock.

- Remember that locks acquired during a multi-query transaction are held until the **commit** statement is executed. This will reduce concurrency.

- Whenever possible, users should work in their own tables or download into their own tables with **create table as select** statements. Doing so offloads tables where there is heavy concurrent activity.

In a heavy concurrent usage situation, there are two approaches:

- The "never-escalate-at-any-cost" approach:

  Concurrent users are working in different regions of the table. Extreme care is taken by the person whose role it is to deal with concurrency problems, (the ULTRIX/SQL System Administrator, the Database Administrator, or both) to make sure *nobody* escalates to a table lock.

- The "table lock" approach:

  This approach, which minimizes the occurrence of deadlock, is appropriate when there is much concurrent activity on smaller tables or in one part of a larger table.

8.9.1 The "Never Escalate" Approach

This approach is appropriate when the users are working in different parts of the table, running simple queries and updates, and making full use of primary and secondary indexes. Here, the goal is to have users coexist as much as possible within the same tables, where no one impedes another user's performance by acquiring table locks. Considerations of the "never escalate" approach include:

- A single-table keyed query starts with page locking, unless the **set lockmode** statement has been issued. Page locks are acquired until **maxlocks** is reached, at which point lock escalation occurs. By checking the *tuple identifiers* (tids) of rows visited, you can estimate the number of pages visited in a specific table.

- More complex queries may take a table-level lock right away, if the ULTRIX/SQL optimizer thinks that **maxlocks** pages will be used.

- Make sure that you are using primary and secondary indexes effectively. Check how many pages are returned from a keyed, primary or secondary lookup to check that it is less than **maxlocks** for that table. The **optimizedb** statement should be run at least on primary and secondary keys to help the optimizer make estimates.

- Monitor overflow levels in tables with **isam** and **hash** primary and secondary indexes.

- It is advisable to reduce fillfactors to lower than the default if tables with **isam** or **hash** storage structures are used, since this provides more room in the table after the **modify**.

- Make sure **maxlocks** is set to an appropriate figure, such as ten percent of table size.

When choosing storage structures while using the "never escalate" approach, the basic principle is that **isam** or **hash** structures with little or no overflow are better than small **btrees** in a concurrent environment. The reason is that growing **btrees** involve some locking when index pages split.

However, as the percentage of overflow builds up in the **hash** or **isam** structure, they become inferior to **btrees**, because locks are held within overflow chains. In particular, if any overflow chain being visited is greater than **maxlocks**, escalation to table locks will occur. This may increase the risk of deadlocks when there are multiple users in the same table.

At what point the tradeoff occurs depends on the circumstances, such as how frequently **modify** statements can be performed. Experimentation is advised. Overflow buildup should be checked in secondary indexes as well as primary indexes.

Concurrent performance is much more difficult to analyze than single-user performance. Be prepared to experiment using the guidelines presented.

8.9.2 The "Table Lock" Approach

The "table lock" approach is used only when there are unsolvable bottlenecks. The philosophy behind the approach is that it is better for users to queue up in an orderly manner to get into a table, thereby avoiding the risk of deadlock, than to waste time backing out of deadlock situations.

Note

Before using this approach, ensure that lock escalation and transaction size are minimized.

This approach is appropriate when extensive table scanning is needed, as with set functions such as **max** and **min**. In these cases it may be advisable to keep an extra table around containing **max** and **min** values, or to search for **max** and **min** values directly in a secondary index without reference to the base table.

In multi-query transactions, table locks reduce the likelihood of deadlocks, but do not eliminate them. The following statement reduces the likelihood of deadlock in a multi-query transaction:

set lockmode on *tablename*
where level = table;

To do the same for secondary indexes, where necessary, use the following statement:

set lockmode on *indexname*
where level = table;

This also applies to **btrees** when they are small.

Under some circumstances setting **readlock = exclusive** might be useful. For example, when executing an open, fetch, update sequence in a program or the subquery that is part of an **update** statement, there is little advantage to taking shared locks for the retrieval operation and immediately escalating the same locks to exclusive for the update. In this case, setting **readlock = exclusive** helps ensure completion of the update quickly and without interference.

Backup and Recovery 9

9.1 Overview

Databases, or individual tables, can be damaged accidentally by hardware failure or human error. For instance, a disk crash, power failure or surge, operating system bug, or system crash can destroy or damage your database or tables in it. For this reason, it is important to back up your database regularly, so that you can recover your data if necessary.

This chapter describes the following methods for backing up and recovering ULTRIX/SQL databases:

- Using checkpoints and journals to back up your database
- Using the **unloaddb** command to back up your database
- Using the **copydb** command to back up particular tables or all of the objects you own in a database
- Using operating system backups to replace current or destroyed tables in a database
- Using the **rollforwarddb** command to recover a database from checkpoints and journals

9.2 The ULTRIX/SQL Logging System

The ULTRIX/SQL logging system keeps track of all database transactions automatically. It is comprised of:

- The logging facility, which includes:
  - A transaction log file
  - Shared memory that contains the logging database
- A recovery process (dmfrcp)
- An archiver process (dmfacp)

9.2.1 The Logging Facility

Each ULTRIX/SQL installation has one installation-wide transaction log file that keeps track of all ULTRIX/SQL transactions for all users. This log file is identified by the environment variable II_LOG_FILE.

The logging facility logs ULTRIX/SQL transactions and manages the logging file. It ensures that log records are written in a way that makes them accessible to the recovery and archiver processes. These processes manipulate the data in the transaction log file when certain events occur. For example, after a transaction is committed, the logging facility moves the log buffer, which resides in shared memory, to the transaction log file.

9.2.2 The Recovery Process

The recovery process handles on-line recovery from system failures and transaction aborts caused by user actions. ULTRIX/SQL writes consistency points into the transaction log file to ensure that all databases are consistent up to that mark and to allow on-line recovery to take place when a problem is detected. When ULTRIX/SQL rolls back a transaction, users may continue working in the database.

9.2.3 The Archiver Process

The archiver process removes completed transactions from the transaction log file and writes them to the corresponding journal files for the database, for journaled tables. Each database has its own journal files, which contain a record of all the changes made to the database since the last checkpoint was taken. The archiver process "sleeps" until sufficient portions of the transaction log file are ready to be archived or until the database is removed from the logging system.

9.3 Verifying the Accessibility of Your Data

To verify that the data in your database is accessible before backing it up, use one of the following methods:

- Run **sysmod** on your system catalogs and **modify** on the user tables. (See the *ULTRIX/SQL Reference Manual* for a description of these commands.)
- Use any procedure that will touch all the rows in each table being backed up; for example, **select** all the rows from the tables.

If rows in a table are not accessible, you will receive an error message. If this happens, restore the table from an earlier backup before doing a new backup.

There is no ULTRIX/SQL or ULTRIX utility that can verify the accessibility of your tables for you. You must do this by using one of the methods listed above. You can, however, write a script that will automatically check each of the tables and system catalogs in your database.

Before you back up your database, it is a good idea to scan the error log for access method failures. If you find any such errors, please submit a Software Performance Report (SPR).

9.4 Backing Up a Database with Checkpoints

By using the **ckpdb** (checkpoint) command, you can make a static backup of your entire database. This enables you to restore all data up to the last checkpoint using the **rollforwarddb** command.

For an up-to-the-minute backup of your database, use the **ckpdb** command in combination with journaling. See the section "Using the ULTRIX/SQL Journaling System" below for instructions.

Each time you run the **ckpdb** command, it creates a new checkpoint for the named database. When the checkpoints are being created, the database is exclusively locked and is not available to any user until the checkpoint is complete.

To checkpoint a database, you must be the Database Administrator for the database or an ULTRIX/SQL superuser impersonating the Database Administrator by using the **-u** or **-s** flags on the **ckpdb** command line.

Note

Taking checkpoints and journaling tables is not a substitute for normal operating system backups.

9.4.1 The ckpdb Command

The syntax of the **ckpdb** is:

ckpdb [**-d**] [**+j**|**-j**] [**-m***device*] [**-u***username*] [**-s**] [**+w**|**-w**] {*dbname*}

For a complete description of the flags and parameters, see the *ULTRIX/SQL Reference Manual.*

9.4.2 Checkpointing a Database

To checkpoint a database, issue the following command at the operating system prompt:

ckpdb *dbname*

This command causes a checkpoint of the database to be made without affecting the state of journaling.

For instructions on enabling and disabling journaling with a checkpoint, see the sections that follow entitled "Starting Journaling" and "Stopping Journaling."

9.4.3 Cleaning Up Outdated Checkpoints

To delete all previous checkpoints and journals when you take a new checkpoint, use the **-d** flag with the **ckpdb** command:

ckpdb -d *dbname*

If you need to delete older checkpoints that have not been removed by using the **-d** flag, use the ULTRIX **rm** command. In this case, delete all but the most recent checkpoint. You can identify the most recent checkpoint by its version number.

When you checkpoint a database, ULTRIX/SQL creates a checkpoint file for each location on which the database is stored. The names of the checkpoint files are in the following format:

C000*v*00*l*.ckp

where *v* is the version number of the checkpoint sequence and *l* is the location number of the data directories. The most recent checkpoint file has the highest version number.

9.4.4 Checkpoints and Destroyed Databases

A checkpoint is a backup of an *existing* database. If you destroy the database, you will not be able to re-create it from a checkpoint because the checkpoint has been destroyed.

If you want to destroy your database and then re-create it, use **unloaddb**. See Chapter 6 for instructions.

9.4.5 Putting Checkpoints on Tape

The **ckpdb** command allows you to checkpoint directly to magnetic tape.

ULTRIX/SQL uses an operating system utility, such as **tar** or **cpio**, to create checkpoints. Both **cpio** and **tar** are limited to handling files that will fit on a single tape. Since checkpoints of larger databases will abort at the end of the first tape, you must estimate both the checkpoint size and the tape capacity before checkpointing these databases. If you estimate that the checkpoint will exceed the tape size, follow instructions in "Checkpointing to Multiple Tapes" later in this chapter.

The following sections provide instructions for estimating checkpoint and tape size, checkpointing to a single tape, and checkpointing to multiple tapes.

9.4.5.1 Estimating Checkpoint File Size

ULTRIX/SQL creates a separate checkpoint file for each location of a database. To estimate the size of checkpoint files:

1. Issue the following command at the ULTRIX system prompt:

   du $II_DATABASE/ingres/data/default/{*dbname***}**

   For other locations, substitute the name of the directory associated with the location name.

2. For **tar**, increase the resulting block size of the directory by 5%.

3. To get the file size in bytes, multiply the block size by the number of bytes in a block on your operating system.

See your operating system manual for information on the number of bytes in a block on your system.

9.4.5.2 Estimating Tape Capacity

The capacity of a tape depends on the:

- Density at which the tape is written
- Length of the tape
- Size of the blocks written on the tape
- Length of the inter-record gap (IRG)

Standard drives write tapes at either 800, 1600, or 6250 bits per inch (bpi) on 9 tracks, so the bits per inch specification is the same as saying bytes per inch. The standard tape length is 2,400 feet.

Block sizes, which are not standardized, are important because of what is between the blocks—the IRG. A typical IRG is .75 inches of empty tape separating each block from the next. With this information, you can use the following formula to estimate the size of the file in bytes that a tape can accommodate:

$$F=\frac{12*B*D*L}{B+(I*D)}$$

where:

- F is the file size in bytes
- B is the block size in bytes
- D is the density in bpi
- L is the length of the tape in feet
- I is the IRG in inches

The file sizes in the following table were calculated for a standard 2400 foot tape, assuming an IRG of .75.

Table 9-1: Examples of File Sizes

Tape Size	IRG	Block Size	Density	File Size in Mbytes
2400	.75	512	1600	13.8
2400	.75	512	6250	17.7
2400	.75	8192	1600	40.2
2400	.75	8192	6250	114.5

After using this formula to calculate the file size, you need to add an arbitrary amount to allow for miscalculations. You do not want a tape to run off the reel because you miscalculated the size of the file that ought to fit. A reasonable amount to add is 5% of a tape's capacity.

If your system uses a cartridge tape or other storage media, contact the vendor for the specifications that will allow you to make the calculations described above.

9.4.5.3 Checkpointing to a Single Tape

To checkpoint a database to a single tape:

1. Mount a tape reel.
2. For a tape drive whose device special file is named /dev/rmt8, issue the following command at the operating system prompt:

```
ckpdb -m/dev/rmt8 dbname
```

The backup created by this checkpoint writes over everything that was on the tape previously.

9.4.5.4 Checkpointing to Multiple Tapes

There are two cases to consider when checkpoint files exceed tape size.

Case 1: The checkpoint file exceeds the size of the tape, but will fit on a disk.

In this case, follow this procedure:

1. Follow normal procedures for checkpointing to disk.
2. Have your Operating System Administrator move the checkpoints from disk to tape. Use a standard system backup method, such as **cpio** or **dump**.

   If some of the database's tables are stored in alternate locations, the **ckpdb** command creates separate checkpoint files for them in the checkpoint location. These files may be small enough to move to single tapes.

   Caution

   It is possible that large checkpoints will exceed the **ulimit** on your system. (The **ulimit** is a tunable operating system parameter that sets a limit on file size.) It can be set using **limit** in the **csh(1)** shell, **ulimit** in the **sh5(1)** shell, or the **ulimit(2)** system call.

Case 2: The checkpoint file exceeds the size of the tape and will NOT fit on a disk.

In this case, you must checkpoint the database manually. To successfully checkpoint a database manually, you have to lock all users out during the entire process.

To lock out all users and take the checkpoint, follow this procedure:

1. To synchronize journaling, checkpoint the database to a null device:

ckpdb +w -d -m/dev/null *dbname*

The **+w** flag causes the **ckpdb** command to wait until all user locks have been released before beginning the checkpoint.

The **-d** flag removes all previous checkpoints and journals.

The **-m** flag causes the checkpoint to be placed in /dev/null, which is a nonexistent device. This makes the database "think" it is being checkpointed and causes journaling to be correctly synchronized. At this time, all changes to the database are guaranteed to be on disk.

2. To lock the database, start a new process:

   a. C shell:

      After the first message from **ckpdb** is printed, press **Control-Z.**

      Bourne shell:

      Log in at another terminal immediately after the checkpoint begins.

   b. Start the new process:

      sql -l +w *dbname*

      The **-l** flag requests a lock on the entire database.

      The **+w** flag tells ULTRIX/SQL to wait until that lock is granted.

3. After the checkpoint finishes:

   C shell:

   If the checkpoint process is stopped (csh job control), put the job back in the foreground; then wait for the process to complete.

   Bourne shell:

   Wait for the process to complete.

4. Have your Operating System Administrator use standard system backup methods to back up the database directory to tape.

   Make sure that the backup method used allows you to save the files and recover them to their original places on the system. Some backup methods have limitations.

5. C shell:

   Leave the second process stopped (csh).

   Bourne shell:

   Leave the second process at the interactive ULTRIX/SQL Terminal Monitor prompt (*) until the backup is complete.

6. Quit from the ULTRIX/SQL Terminal Monitor prompt held by the second process.

9.5 Using the ULTRIX/SQL Journaling System

For a dynamic backup of your database, use journals in combination with checkpoints. Checkpoints provide you with a snapshot of the database at the time you took the checkpoint. Journals keep track of all changes made to journaled tables since the last checkpoint.

When you are journaling a database:

- Take regular checkpoints of your database to minimize recovery time.
- Periodically verify that your journaling data is correct by auditing the database. See the section "Producing Audit Trails with Journals" below.

9.5.1 Starting Journaling

In ULTRIX/SQL, journaling is selective on a table-by-table basis. That is, you must identify those tables in a database that you want to journal.

To start journaling, you:

1. Select the tables you want journaled and enable journaling on them.
2. Enable journaling on the database by using the **+j** flag with the **ckpdb** command.

For instructions on enabling journaling on tables, see the sections "Enabling Journaling on New Tables" and "Enabling Journaling on Existing Tables" below.

If you have not enabled journaling on the database, ULTRIX/SQL begins journaling the new tables when you enable journaling on the database by taking a checkpoint and using the **+j** flag:

ckpdb +j *dbname*

9.5.1.1 Enabling Journaling on New Tables

There are two ways to enable journaling on new tables:

- With the **with journaling** option of the **create table** statement

  For example, to turn journaling on when you create the "emp" table, issue this command:

```
create table emp
        (name varchar(20),
        age i2,
        salary money,
with journaling;
```

- By setting journaling on for an entire session with the **set journaling** option of the **set** statement, for example:

```
set journaling;
```

  This allows you to enable journaling on *all* the tables you create during a session.

If you have enabled journaling on the database, ULTRIX/SQL begins journaling the newly created tables *immediately*.

9.5.1.2 Enabling Journaling on Existing Tables

To enable journaling on an existing table, use the **journaling** option of the **set** statement:

set journaling on *tablename*;

When you enable journaling on a table *after* creating it, ULTRIX/SQL does not begin journaling the table until you take the next checkpoint. If you have not previously enabled journaling on the database, use the **+j** flag when you take this checkpoint:

ckpdb +j *dbname*

After using the **+j** flag to begin journaling, you do not need to use it when you take subsequent checkpoints. ULTRIX/SQL will continue to journal the table until you specifically stop journaling on it.

For complete descriptions of **set** and **ckpdb**, see the *ULTRIX/SQL Reference Manual.*

9.5.2 Stopping Journaling

To stop journaling a particular table, use the **set nojournaling** statement:

set nojournaling on *tablename*;

To stop journaling all the tables in a database, issue the following command at the operating system prompt:

ckpdb -j *dbname*

This will cause ULTRIX/SQL to take a checkpoint of the named database and then stop journaling it. After stopping journaling, you can still take periodic checkpoints of the database.

9.5.3 Producing Audit Trails With Journals

In addition to using journals for recovery, you can use them to produce audit trails of changes to a database. You use the **auditdb** command, described in this section, to produce these audit trails.

Periodically, you should run **auditdb** to verify that your journals are correct.

9.5.3.1 The auditdb Command

The **auditdb** command enables you to produce a listing or file of changes made to journaled tables since the last checkpoint. This listing may not include *all* changes made since the last checkpoint for the following reasons:

- Since **auditdb** does not exclusively lock the database, other users may complete a transaction while **auditdb** is running.
- If other users are using the database when you run **auditdb**, ULTRIX/SQL may not have moved a completed transaction to the journal files.

You must be the DBA for the database or an ULTRIX/SQL superuser to run **auditdb** on a database.

The syntax of **auditdb** is:

auditdb [**-a**] [**-b***dd-mmm-yyyy:hh:mm:ss*] [**-e***dd-mmm-yyyy:hh:mm:ss*] [**-s**] [**-t***tablename*] [**-f**] [**-u***username*] {*dbname*}

For a complete description of the **auditdb** flags and parameters, see the *ULTRIX/SQL Reference Manual.*

9.5.3.2 Loading an Audit Trail as a Table

To make querying the data easier, you can create an audit trail as a file in your current directory and then load the file into a table in your database. To do this:

1. When you create the audit trail, use the **-f** flag to create a file named audit.trl in your current directory. You can use this flag only if the table you are auditing has less than 120 columns and less than 1948 bytes per row.

   In the following example, **auditdb** extracts a record of the changes to the "employee" table from the journal for the "demodb" database. It automatically places the changes in the current directory in a file named audit.trl.

   ```
   auditdb -temployee -f demodb
   ```

2. To rename the audit.trl file, use the ULTRIX **mv** command.

   In the following example, the audit.trl file is renamed to empaudit.trl.

   ```
   mv audit.trl empaudit.trl
   ```

3. To copy the file into a database table, create a table to hold the audit trail data.

   When creating the table, include the audit trail table columns shown below. Enter the audit trail columns before the table's columns, in the order shown. If you don't, the **copy** statement will fail when you try to copy the audit trail data into the table.

Table 9-2: Audit Trail Table Columns

Column Name	Data Type	Description
date	**date** not null with default	Date and time of the beginning of the multi-query transaction that contained the operation
username	**char(24)** not null with default	ULTRIX/SQL username of the user who performed the operation
operation	**char(8)** not null with default	Select, insert, update, delete
tranid1	**integer** not null with default	Transaction identification number. Concatenated with tranid2.
tranid2	**integer** not null with default	Transaction identification number. Concatenated with tranid1.
table_id1	**integer** not null with default	Table identification number. Corresponds to the value in the **table_reltid** column of the **iitables** system catalog for the specified table.
table_id2	**integer** not null with default	Table identification number. Corresponds to the value in the **table_reltidx** column of the **iitables** system catalog for the specified table.

In the following example, a table named "empaudit" is created to hold the data from the empaudit.trl file.

```
create table empaudit
    (date date not null with default,
    username 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,
    name varchar(20),
    age integer,
    salary money,
    dname varchar(10),
    manager varchar(20))
```

The last five columns are columns from the employee table.

4. Use the **copy** statement to load the new table with the data from the empaudit.trl file.

   In the following example, the data in the empaudit.trl file is copied to the "empaudit" table:

```
copy empaudit() from '/usr/joe/empaudit.trl';
```

The "empaudit" table will contain a row for each row added to the "employee" table, a row for each row removed, and two rows for each update: one showing the row before the update and the other showing the row after the update.

9.6 Backing Up with copydb

The **copydb** command, which is explained in detail in Chapter 6, can be used to back up the tables that *you own* in a database.

If you specify tablenames with **copydb**, only those tables will be copied. If you do not specify tablenames with **copydb**, *all* of the tables, views, and procedures that you own in the database will be copied. See Chapter 6 for a complete explanation of what **copydb** copies.

Since any user authorized to use a database can use **copydb**, this is a useful backup method for a non-Database Administrator, who can use it to back up his or her own tables, views, and procedures.

Before you use the following procedure, you should understand how **copydb** works; see Chapter 6 for a complete explanation of this command.

To back up tables with **copydb**:

1. Create a temporary working directory for the **copy.in** and **copy.out** scripts and move to this directory.

2. To back up specified tables, issue the following command at the operating system prompt, for example:

   copydb *dbname table1 table2*

   To back up all the tables, views, and procedures that *you own* in the database, issue the following command at the operating system prompt:

   copydb *dbname*

   This creates **copy.out** and **copy.in** scripts for the objects copied.

   Unless you specify otherwise, the **copy.in** and **copy.out** scripts will be copied into the default directory. If you want them stored in another directory, use the **-d** flag:

   copydb -d*directory_name dbname*

4. To copy the data into the default or specified files, issue the following command from the operating system:

   sql *dbname* **<copy.out**

   This creates a copy of the objects copied in your database. You can store these files on tape or leave them on disk.

To restore data from a **copydb** backup, you run the **copy.in** script. See the section "Recovering Data from copydb Backups."

9.7 Backing Up with unloaddb

The **unloaddb** command is a time-consuming method for backing up and recovering your database, because all of your database's files must be unloaded and then reloaded. For this reason, it is recommended that you use the **ckpdb** command instead.

However, **unloaddb** can be useful as a backup tool because it enables you to:

- Generate copy scripts, which can be used to re-create your database
- Recover particular tables by editing the **copy.in** scripts (See Chapter 6 for a description of the **copy.in** scripts.)

To use **unloaddb** to back up a database, run **unloaddb** on the database and then execute the **unload.ing** file. If you need to recover from a backup made with **unloaddb**, execute the **reload.ing** command file . For detailed instructions on using **unloaddb**, see Chapter 6.

9.8 Using Operating System Backups

Do not rely solely on operating system backups to back up your ULTRIX/SQL databases for the following reasons:

- You may not be able to control the frequency of these backups.
- If a user is working in the database during the backup, the backup copy of the database may be inconsistent. To prevent an inconsistent backup, make sure that all users are out of ULTRIX/SQL before doing the backup.

However, system backups can be useful for replacing a current table. Additionally, you can use system backups to restore a destroyed table and to restore the system catalogs, although this is a difficult operation and should be done by experienced ULTRIX/SQL users only.

To preserve your database, make sure that the following directories are backed up during the system backup:

- The directory containing the database location for the database
- Directories containing locations to which the database has been extended
- Directories containing embedded language programs, which are not stored in the database

9.8.1 Mapping File Names to Table Names

The tables in your database are stored as files. The names of these files are not the same as the names of the tables. To determine the file names associated with your tables, issue the following statement:

select * from iifile_info;

Since you will need to know the pathname to restore a table from an operating system backup, it is recommended that you execute this command every night and save the resultant listing, so that you always have a current list of the file names associated with your database's tables.

9.8.2 Replacing a Current Table with a System Backup Copy

When you replace a current table with a copy from backup tape, the columns, column widths, data types, keys and storage structures of the current table and the backup table must be the same. If they are not, this procedure will not work. If you are not sure that they are the same, backing up the entire database is safer than using this procedure.

To perform this procedure, you will need help from both your ULTRIX Operating System Administrator and ULTRIX/SQL System Administrator.

To replace a current user table with a system backup copy:

1. Back up the table that is being replaced.

2. Verify that the storage structure of the current table and the copy of the table on tape are the same.

   If they are not, modify the storage structure of the current table to the same storage structure as the table on tape. Do this even if there is no data in the current table. If you do not, ULTRIX/SQL will not correctly access the data replacing the current table.

3. To lock the database exclusively, execute this command at the operating system prompt:

   sql -l *dbname*

4. Have your Operating System Administrator replace the current file on disk with the backup copy from tape. Make sure you know the location of the table.

   The name of the file on disk and the table name are not the same. To find out the file name, issue the following SQL statement:

   select *file_name*
   from iifile_info where
   table_name = 'the_name_of_your_table';

   This command works only if you have not destroyed and re-created the table since the last operating system backup.

5. Have your ULTRIX/SQL System Administrator shut down the database management system server and start up a new one in order to clear the server's buffer manager. See the *ULTRIX/SQL Operations Guide* for instructions on starting and stopping database management system servers.

6. Log into ULTRIX/SQL and access the database into which you are replacing the table.

7. Run the **modify** statement on the replaced table.

   Although you do not have to modify the table, it is a good idea to do so. This ensures that the row count and the number of pages are correct. Since ULTRIX/SQL uses this information to formulate query execution plans, it must be correct.

8. If the table is journaled, take a new checkpoint and restart the journals.

9.8.3 Replacing a Destroyed User Table from Backup Tape

To replace a destroyed user table from backup tape:

1. Re-create the table exactly as it was created before.

   Make sure the column ordering, widths, and other details are exactly the same. This procedure will not work if there are any differences in the order of columns, column widths, data types, keys, or storage structures.

2. Follow steps 1 through 8 in the preceding section.

9.9 Recovering Databases

This section tells you how to:

- Recover a non-journaled database from a checkpoint
- Recover a journaled database from checkpoints and journals
- Recover a database from a taped checkpoint
- Recover tables from **copydb** backups

9.9.1 Recovering Databases from Checkpoints and Journals

To recover a database from checkpoints and journals or from checkpoints only, use **rollforwarddb**. The **rollforwarddb** command overwrites the current contents of the database being recovered.

When you run **rollforwarddb**, ULTRIX/SQL locks the database to prevent errors from occurring. If the database is busy, **rollforwarddb** waits for the database to be free before recovering it. (If you use the -w flag, **rollforwarddb** will proceed to the next database entered if the first database is busy.)

To use **rollforwarddb**, you must be the DBA for the database or an ULTRIX/SQL superuser (using the **-u** or **-s** flags).

9.9.1.1 The rollforwarddb Command

The syntax of the **rollforwarddb** command is:

rollforwarddb [**-b***dd-mmm-yyyy:hh:mm:ss*] [**+c**|**-c**] [**-e***dd-mmm-yyyy:hh:mm:ss*] [**+j**|**-j**] [**-m***device:*] [**-s**] [**-u***username*] [**-v**] [**+w**|**-w**] {*dbname*}

For a complete description of the flags and parameters, see the *ULTRIX/SQL Reference Manual.*

9.9.1.2 Recovering a Non-Journaled Database

To recover a non-journaled database from the last checkpoint, issue the following command at the operating system prompt:

rollforwarddb +c *dbname*

9.9.1.3 Recovering a Journaled Database

To recover a database from the last checkpoint and journal, where both the checkpoints and journals are stored online, issue the following command at the operating system prompt:

rollforwarddb *dbname*

To recover all of your databases from checkpoints and journals, issue the following command at the operating system prompt:

rollforwarddb

9.9.1.4 Recovering a Database From Taped Checkpoints

To recover a database whose checkpoints are on tape:

1. Mount the tape reel containing the checkpoints.
2. With a tape drive named /dev/rmt8, issue the following command at the operating system prompt:

```
rollforwarddb +c +j -m/dev/rmt8/dbname
```

The **rollforwarddb** command reads the checkpoint from the tape and then applies the appropriate journal files to bring your database up to date.

9.9.1.5 Retracting Changes with rollforwarddb

If a user makes a serious error in a table that is being journaled, you can retract the changes. Use **rollforwarddb** to restore the database up to the beginning of the transaction in which the error occurred.

For example, to restore database "db1" from the previous checkpoint to its condition at 8:00 a.m. on August 17, 1989, issue the following command at the operating system prompt:

```
rollforwarddb -v +c +j -e17-aug-1989:08:00:00 db1
```

This retracts *all* changes made to the database after this time, not just those made to the table with the error.

To ensure that the error is not reintroduced when you run **rollforwarddb** in the future, take a new checkpoint to reset the journals.

9.9.2 Recovering Data from copydb Backups

To recover data from a backup made with **copydb**, execute the **sql** command to run the **copy.in** script. To do this, issue this command from the operating system:

sql *dbname* **<copy.in**

9.9.3 Recovering Inconsistent Databases

Although it is highly unlikely that your database will become inconsistent, certain conditions may cause this to happen. If it does, please submit a Software Performance Report (SPR).

ULTRIX/SQL System Files A

A.1 Overview

When ULTRIX/SQL is initialized, your ULTRIX/SQL System Administrator determines where to place the directories and files for ULTRIX/SQL system code, data files, checkpoints, journals, the transaction log file, and other files. This appendix describes these directories and files and provides the environment variable set for each during the initialization of ULTRIX/SQL.

A.2 ULTRIX/SQL Files and Directories

The following table lists the major ULTRIX/SQL system files and directories, and provides the environment variable set for them when ULTRIX/SQL is initialized. The environment variable for each of these files points to the directory structure where they reside on your system.

ULTRIX/SQL File or Directory	Description	Environment Variable
Code for your ULTRIX/SQL installation	Executable images of ULTRIX/SQL programs, error and help files, embedded ULTRIX SQL libraries, and other miscellaneous files.	II_SYSTEM
ULTRIX/SQL files	Error, help, header, symbol table, user, and other miscellaneous files.	II_CONFIG
Log file	File to store information about all ULTRIX/SQL transactions that are performed in an installation.	II_LOG_FILE
Data files	File containing the database's tables and system catalogs. Each database has its own system catalogs, which store detailed information about it. Also contains the **iidbdb**, the ULTRIX/SQL master database	II_DATABASE (points to the default directory for data files)
Checkpoint files	Static copies of a database.	II_CHECKPOINT (points to the default directory for checkpoints)

ULTRIX/SQL File or Directory	Description	Environment Variable
Journal files	Dynamic records of the changes made to a database since the last checkpoint. These records are kept only for databases with journaling enabled.	II_JOURNAL (points to the default directory for journals)

The lockstat Utility B

B.1 Overview

This appendix tells you how to use the **lockstat** utility and describes its output.

B.2 Using the lockstat Utility

The **lockstat** utility allows you to examine the state of the ULTRIX/SQL Lock Database. It provides a summary listing and a "snapshot" of the installation's locking activity.

To invoke the **lockstat** utility, issue the following command at the operating system prompt:

```
lockstat
```

The following figure shows an example of the output from the **lockstat** utility. The example is followed by an explanation of each part of the output.

```
=======8-AUG-1989 14:02:17.17 Locking System Summary=======
  Create lock list    42        Release lock list        23
  Request lock       157        Re-request lock           4
  Convert lock        68        Release lock            103
  Escalate             0        Lock wait                 1
  Convert wait         0        Convert Deadlock          0
  Deadlock Search      1        Deadlock                  0
  Cancel               0
--------------------Locks by lock list---------------------
Id: 0001001E Tran_id: 0000009287AB931A R_llb: 00000000 R_cnt: 0
  Wait: 00000000 Locks: (0,0/128) Status: NONPROTECT,EWAIT,ESET
Id: 0001001F Tran_id: 0000000000000010 R_llb: 00000000 R_cnt: 0
  Wait: 00000000 Locks: (0,0/128) Status: NONPROTECT,NOINTERRUPT
Id: 00010022 Tran_id: 000000000000000E R_llb: 00000000 R_cnt: 0
  Wait: 00000000 Locks: (8,0/128) Status: NONPROTECT
    Id: 00030066 Rsb: 0001002C Gr: IS  Req: IS  State: GR PHYS(1)
      KEY(BM_DATABASE,DB=00000001)
    Id: 0003006E Rsb: 00010029 Gr: IS Req: IS State: GR PHYS(1)
      KEY(SV_PAGE,DB=00000001,TABLE=[1,0],PAGE=13)
    Id: 0003000D Rsb: 00010027 Gr: IS  Req: IS  State: GR PHYS(1)
      KEY(SV_PAGE,DB=00000001,TABLE=[1,0],PAGE=4)
```

```
     Id: 0002005C Rsb: 0001001D Gr: IS  Req: IS  State: GR PHYS(1)
       KEY(SV_PAGE,DB=00000001,TABLE=[3,0],PAGE=4)

     Id: 00040062 Rsb: 0002001F Gr: IS  Req: IS  State: GR PHYS(1)
       KEY(BM_TABLE,DB=00000001,TABLE=[44,0])

     Id: 00010057 Rsb: 00010058 Gr: IS  Req: IS  State: GR PHYS(1)
       KEY(SV_PAGE,DB=00000001,TABLE=[44,0],PAGE=10)
Id: 00010027 Tran_id: 000000000000000B R_llb: 00000000 R_cnt: 0
  Wait: 00000000 Locks: (0,0/128) Status: NONPROTECT,NOINTERRUPT
Id: 00010028 Tran_id: 000000000000000A R_llb: 00000000 R_cnt: 0
   Wait: 00000000 Locks: (8,0/128) Status: NONPROTECT
     Id: 0001002B Rsb: 0001002C Gr: IS  Req: IS  State: GR PHYS(1)
        KEY(BM_DATABASE,DB=00000001)

     Id: 00010028 Rsb: 00010029 Gr: IS  Req: IS  State: GR PHYS(1)
        KEY(SV_PAGE,DB=00000001,TABLE=[1,0],PAGE=13)

     Id: 00010026 Rsb: 00010027 Gr: IS  Req: IS  State: GR PHYS(1)
        KEY(SV_PAGE,DB=00000001,TABLE=[1,0],PAGE=4)

     Id: 00010020 Rsb: 00010021 Gr: IS  Req: IS  State: GR PHYS(1)
        KEY(SV_PAGE,DB=00000001,TABLE=[1,0],PAGE=20)

     Id: 0001001C Rsb: 0001001D Gr: IS  Req: IS  State: GR PHYS(1)
        KEY(SV_PAGE,DB=00000001,TABLE=[3,0],PAGE=4)

     Id: 0002001E Rsb: 0002001F Gr: IS  Req: IS  State: GR PHYS(1)
        KEY(BM_TABLE,DB=00000001,TABLE=[44,0])

     Id: 00010018 Rsb: 00010019 Gr: IS  Req: IS  State: GR PHYS(1)
        KEY(SV_PAGE,DB=00000001,TABLE=[44,0],PAGE=3)
Id: 00010029 Tran_id: 0000000000000009 R_llb: 00000000 R_cnt: 0
   Wait: 00000000 Locks: (0,0/128) Status: NONPROTECT,NOINTERRUPT
---------------------------Locks by resource-----------------------------
Id: 00010001 Gr: IS  Conv: IS  Value: 00000000000
   KEY(SV_PAGE,DB=00000001,TABLE=[1,0],PAGE=15)
     Id: 00010070 Llb: 0001002C Gr: IS  Req: IS State: GR PHYS(1)
Id: 00010003 Gr: IS  Conv: IS  Value: 00000000000
   KEY(SV_PAGE,DB=00000001,TABLE=[1,2],PAGE=2)
     Id: 00010002 Llb: 0001002C Gr: IS  Req: IS  State: GR PHYS(1)
Id: 00010019 Gr: IS  Conv: IS  Value: 00000000000
   KEY(SV_PAGE,DB=00000001,TABLE=[44,0],PAGE=3)
     Id: 00010010 Llb: 0001002C Gr: IS  Req: IS  State: GR PHYS(1)
     Id: 00010018 Llb: 00010028 Gr: IS  Req: IS  State: GR PHYS(1)
Id: 0003006D Gr: IS  Conv: IS  Value: 00000000000
   KEY(BM_TABLE,DB=00000001,TABLE=[151,0])
     Id: 0003006C Llb: 0001002C Gr: IS  Req: IS  State: GR PHYS(1)
==============================================================================
```

B.2.1 Interpreting the Locking System Summary

The first portion of the output is a summary listing of locking activity for this installation. All values are cumulative from the time **iistartup** was run for this iteration of the system. The meaning of each entry is described in the following table.

Table B-1: Explanation of Locking System Summary

Entry	Explanation
Create lock list	Number of times a lock list was created for server, session, or transaction.
Release lock list	Number of times a release of a lock list occurred for a server, session, or transaction.
Request lock	Number of new lock requests that the ULTRIX/SQL locking system processed.
Re-request lock	Number of times an implicit lock conversion request was issued on a resource that the lock list already had locked. Implicit lock conversion requests can occur when a request is made on a page for update that was previously requested for read.
Convert lock	Number of times an explicit lock conversion request is made to change a lock mode on a physical lock from one mode to another. These types of requests occur as a result of a physical lock being converted during an existing transaction to lower or higher modes.
Release lock	Number of times a specific ULTRIX/SQL logical lock is released, as opposed to a full, partial, or physical lock release.
Escalate	Number of times a partial release occurred to allow lock escalation from page to table level.
Lock wait	Number of times a new lock request had to wait to be granted.
Convert wait	Number of times an existing lock waited for conversion to a different lock mode.
Convert Deadlock	Number of times a request for conversion turned into a deadlock.
Deadlock Search	Number of times a deadlock search was initiated.
Deadlock	Number of times that deadlock existed.
Cancel	Number of times a lock request was cancelled due to a timeout or interrupt.

B.2.2 Interpreting the "Locks by lock list" Portion

The second portion of the **lockstat** utility output shows the lock information sorted by lock list. Lock lists either represent transaction units or can span transactions (related lock lists) and contain items such as database locks. The first line item reports the lock list identifier. Any locks associated with the specified lock list are listed following the lock list description and indented to set them off.

"Locks by lock list" values are described in the following table.

Table B-2: Explanation of "Locks by lock list" Values

"Locks by lock list" Values	Explanation	
Id	Internal lock list identifier (lock list block).	
Tran_id	Transaction identifier associated with this lock list. This value correlates to a transaction identifier in the logstat utility output.	
R_llb	Related lock list identifier, if not a transaction lock list.	
R_cnt	Number of related lock list identifiers that this lock list must assure are released before this lock list can be released.	
Wait	Internal resource block identifier of the lock that is currently blocked.	
Locks	Made up of three values: total number of locks currently on the list, number of logical locks currently on the list, and total number of locks allowed to be on this list.	
Status	The state of the lock list at the present time. The possible values are:	
	WAIT	Waiting for a lock
	NONPROTECT	Can be released without going through a recovery (system lock lists)
	ORPHAN	Lock list remaining without a transaction
	EWAIT	Waiting on a system event
	RECOVER	Lock list taken over by the recovery process
	MASTER	Lock list owned by the recovery process
	ESET	Lock list set on the wait queue for an event
	EDONE	Indicates that the event that lock list is waiting on is done
	NOINTERRUPT	List of non-interruptable lock requests

The values indented under individual lock lists are lock block values. These are described in the following table.

Table B-3: Explanation of Lock Block Values (Locks by lock list)

Lock Block Values	Explanation
Id	Internal lock block identifier.
Rsb	Internal resource block identifier.
Gr	Granted lock mode.
Req	Requested lock mode.
State	Current state of lock (GR = granted, WT = waiting).
KEY	Information used to identify the resource being locked. When checking contention on data pages, the key will contain **PAGE**, the database ID that can be traced back to **logstat** output, the table **reltid** and **reltidx**, and the page number.

B.2.3 Interpreting the "Locks by resource" Portion

The third portion of **lockstat** output groups the individual locks by resource block and shows any contention that can lead to query performance problems.

The "Locks by resource" values are described in the following table.

Table B-4: Explanation of "Locks by resource" Block Values

"Locks by resource" Values	Explanation
Id	Internal resource block identifier
Gr	Granted mode of the resource
Conv	Conversion mode requested on the resource
Value	Lock value associated with the resource
KEY	Byte string identifying the resource

The indented portions of the resource blocks show the individual lock blocks that are contending for the resource. These lock blocks are described in the following table.

Table B-5: Explanation of Lock Block Values (Locks by resource)

Lock Block Values	Explanation
Id	Internal lock block identifier
Llb	Lock list identifier that this lock resides on
Gr	Granted mode of the lock
Req	Requested lock mode
State	Current state of the lock (GR = granted, WT = waiting)

Index

A

B

C

D

E

H

I

J

L

M

N

O

P

R

S

T

U

V

W

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 Database Administrator's Guide
AA-PBZ8A-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 Database Administrator's Guide
AA-PBZ8A-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