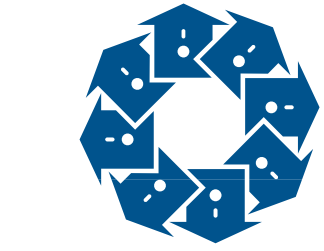
c-treeSQL™

ISQL and Tools Reference Guide

For use with c-treeSQL Server

This manual provides reference material for the ISQL interactive SQL utility and other administrative tools provided in the c-treeSQL environment. It also includes a tutorial describing how to use the ISQL utility.



FairCom®

Copyright © 1992-2004 FairCom Corporation All rights reserved.

Portions © 1987-2004 Dharma Systems, Inc. All rights reserved.

Eleventh Edition, First printing: September 2003

Information in this document is subject to change without notice.

No part of this publication may be stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior written permission of FairCom Corporation. Printed in the United States of America.

FairCom welcomes your comments on this document and the software it describes. Send comments to:

Documentation Comments

FairCom Corporation

2100 Forum Blvd., Suite C

Columbia, MO 65203

Phone: 573-445-6833

Fax: 573-445-9698

Electronic Mail: support@faircom.com

Web Page: http://www.faircom.com

c-tree, c-tree Plus, r-tree, the circular disk logo, and FairCom are registered trademarks of the FairCom Corporation.

c-treeSQL, c-treeSQL ODBC, c-treeSQL ODBC SDK, c-treeVCL/CLX, c-tree ODBC Driver, c-tree Crystal Reports Driver, and c-treeDBX are trademarks of FairCom Corporation.

The following are third-party trademarks:

DBstore is a trademark of Dharma Systems, Inc.

Microsoft, MS-DOS, Windows, Windows NT, and Windows XP are registered trademarks of Microsoft Corporation.

Java, Java Development Kit, Solaris, SPARC, SunOS, and SunSoft are trademarks of Sun Microsystems, Inc.

Macintosh and MacOS are trademarks licensed to Apple Computer Co.

IBM and AIX are registered trademarks of International Business Machines Corp.

HP-UX is a registered trademark of Hewlett-Packard Company.

All other trademarks, trade names, company names, product names, and registered trademarks are the property of their respective holders

ISQL-V8.14-041015

Table of Contents

Documentation Overview

Purpose of This Manual v
Audience v
Structure v
Syntax Diagram Conventions v
Related Documentation vi

1 Introduction

1.1 Overview 1-1

2 Quick Tour

2.1 Introductory Tutorial 2-1
2.1.1 Init 2-1
2.1.2 Define 2-2
2.1.3 Manage 2-2
2.1.4 Done 2-3
2.1.5 Complete Introductory Tutorial Code 2-3
2.2 Relational Model and Indexing Tutorial 2-4
2.2.1 Init 2-4
2.2.2 Define 2-4
2.2.3 Manage 2-6
2.2.4 Done 2-7
2.2.5 Complete Relational Model and Indexing Tutorial Source Code 2-8
2.3 Locking Tutorial 2-9
2.3.1 Init 2-9
2.3.2 Define 2-10
2.3.3 Manage 2-10
2.3.4 Done 2-11
2.3.5 Complete Locking Tutorial Source Code 2-12
2.4 Transaction Processing Tutorial 2-13
2.4.1 Init 2-13
2.4.2 Define 2-13
2.4.3 Manage 2-15
2.4.4 Done 2-16
2.4.5 Complete Transaction Processing Tutorial Source Code 2-16

3 ISQL Statements

3.1 Overview 3-1
3.2 Starting Interactive SQL 3-1
3.3 Statement History Support 3-2
3.4 Formatting Output of ISQL Queries 3-3
3.4.1 Formatting Column Display with the COLUMN Statement 3-6
3.4.2 Summarizing Data with DISPLAY, COMPUTE, and BREAK Statements 3-7
3.4.3 Adding Beginning and Concluding Titles with the TITLE Statement 3-9

3.5 The HELP and TABLE Statements 3-11
3.6 Transaction Support 3-11
3.7 ISQL Reference 3-12
3.7.1 @ (Execute) 3-12
3.7.2 BREAK 3-13
3.7.3 CLEAR 3-15
3.7.4 COLUMN 3-16
3.7.5 COMPUTE 3-21
3.7.6 DEFINE 3-23
3.7.7 DISPLAY 3-23
3.7.8 EDIT 3-25
3.7.9 EXIT or QUIT 3-26
3.7.10 GET 3-27
3.7.11 HELP 3-29
3.7.12 HISTORY 3-29
3.7.13 HOST or SH or ! 3-31
3.7.14 LIST 3-32
3.7.15 QUIT or EXIT 3-33
3.7.16 RUN 3-33
3.7.17 SAVE 3-34
3.7.18 SET 3-34
3.7.19 SHOW 3-37
3.7.20 SPOOL 3-38
3.7.21 START 3-39
3.7.22 TABLE 3-40
3.7.23 TITLE 3-42

4 Data Load Utility: dbload

4.1 Introduction 4-1
4.2 Prerequisites for dbload 4-2
4.3 dbload Command Line Syntax 4-2
4.4 Data File Formats 4-3
4.4.1 Variable Length Records 4-4
4.4.2 Fixed Length Records 4-4
4.5 The Commands File 4-4
4.5.1 The DEFINE RECORD Statement 4-5
4.5.2 The FOR EACH Statement 4-6
4.6 Examples 4-7
4.7 dbload Errors 4-8
4.7.1 Compilation Errors 4-8
4.7.2 Fatal Errors 4-9

5 Data Unload Utility: dbdump

5.1 Introduction 5-1
5.2 Prerequisites for dbdump 5-1
5.3 dbdump Command Line Syntax 5-2

5.4 Data File Formats 5-2
5.5 The Commands File 5-2
5.5.1 The DEFINE RECORD Statement 5-3
5.5.2 The FOR RECORD Statement 5-4
5.6 Examples 5-4
6 Schema Export Utility: dbschema
6.1 Introduction 6-1
6.2 Examples 6-2
A Tutorial Source Code
A.1 Introductory Tutorial A-1
A.2 Relational Model and Indexing Tutorial A-1
A.3 Locking Tutorial A-3
A.4 Transaction Processing Tutorial A-3
Index . . . . . Index-i

List of Figures

Figure 4-1: dbload Execution Process 4-2
Figure 5-1: dbdump Execution Process 5-1

List of Tables

Table 3-1: ISQL Statements for Statement History Support 3-2
Table 3-2: ISQL Statements for Query Formatting 3-4
Table 3-3: Numeric Format Strings for the COLUMN Statement 3-18
Table 3-4: Date-Time Format Strings for the COLUMN Statement 3-18

List of Examples

Example 3-1: Unformatted Query Display from ISQL 3-5
Example 3-2: Controlling Display Width of Character Columns 3-6
Example 3-3: Customizing Format of Numeric Column Displays 3-7
Example 3-4: Specifying Column Breaks and Values with DISPLAY 3-8
Example 3-5: Calculating Statistics on Column Breaks with COMPUTE 3-9
Example 3-6: Specifying a Query Header and Footer with TITLE 3-10
Example 3-7: Sample ISQL script 3-12
Example 4-1: Sample dbload commands files 4-7

Documentation Overview

PURPOSE OF THIS MANUAL

This manual provides reference material for the ISQL interactive SQL utility as well as the *dbload*, *dbdump*, and *dbschema* administrative tools provided in the c-treeSQL environment. It also includes a tutorial describing how to use the ISQL utility.

AUDIENCE

The reader of this manual should be familiar with general SQL concepts.

STRUCTURE

The manual contains the following chapters:

Chapter 1	Describes the use of Interactive SQL (ISQL) for performing ad-hoc queries and for report generation.
Chapter 2	Includes tutorials for getting started with ISQL.
Chapter 3	Includes reference information for ISQL statements.
Chapter 4	Describes the data load utility, *dbload*, which is used to load data from files into existing tables of a database.
Chapter 5	Describes the data unload utility, *dbdump*.
Chapter 6	Describes the data definition export utility, *dbschema*.
Appendix A	Includes full source code for tutorials.

SYNTAX DIAGRAM CONVENTIONS

Syntax diagrams appear in `Courier` type and use the following conventions:

UPPERCASE	Uppercase type denotes reserved words. You must include reserved words in statements, but they can be upper or lower case.

lowercase	Lowercase type denotes either user-supplied elements or names of other syntax diagrams. User-supplied elements include names of tables, host-language variables, expressions, and literals. Syntax diagrams can refer to each other by name. If a diagram is named, the name appears in lowercase type above and to the left of the diagram, followed by a double-colon (for example, privilege ::). The name of that diagram appears in lowercase in diagrams that refer to it.
{ }	Braces denote a choice among mandatory elements. They enclose a set of options, separated by vertical bars (|). You must choose at least one of the options.
[ ]	Brackets denote an optional element or a choice among optional elements.
|	Vertical bars separate a set of options.
...	A horizontal ellipsis denotes that the preceding element can optionally be repeated any number of times.
( ) , ;	Parentheses and other punctuation marks are required elements. Enter them as shown in syntax diagrams.

RELATED DOCUMENTATION

Refer to the following documents for more information:

c-treeSQL Reference Manual	Describes the syntax and semantics of statements and language elements for the c-treeSQL interface.
c-treeSQL Embedded SQL User's Guide	Describes how to develop host language programs containing embedded SQL statements that access c-treeSQL environments.
c-treeSQL ODBC Driver Guide	Describes c-treeSQL support for the ODBC interface and how to configure the c-treeSQL ODBC Driver.
c-treeSQL JDBC Driver Guide	Describes c-treeSQL support for the JDBC interface, configuring the c-treeSQL JDBC Driver, and how applications connect to databases through the driver.
c-treeSQL Guide to Java Stored Procedures and Triggers	Describes how to write and use Java stored procedures and triggers—Java routines which contain SQL statements and are stored in a database. ODBC, JDBC, and SQL applications call stored procedures, while triggers are invoked automatically by database updates.
c-tree Plus Quick Start and Product Overview Guide	Describes the installation process, how to get started, and recommendations for c-tree Plus ISAM/Low-Level, c-treeDB, c-treeVCL/CLX, c-tree Server, and c-treeSQL Server.

Chapter 1

Introduction

1.1 OVERVIEW

Interactive SQL (often referred to throughout this manual as ISQL) is a utility supplied with c-treeSQL that lets you issue SQL statements directly from a terminal and see results displayed at the terminal. You can use interactive SQL to:

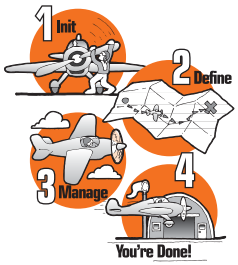
- Learn how SQL statements work
- Test and prototype SQL statements to be embedded in programs
- Modify an existing database with data definition statements
- Perform ad-hoc queries and generate formatted reports with special ISQL formatting statements

With few exceptions, you can issue any SQL statement in interactive SQL that can be embedded in a program, including CREATE, SELECT, and GRANT statements. Interactive SQL includes an online help facility with syntax and descriptions of the supported statements.

Chapter 2

Quick Tour

2.1 INTRODUCTORY TUTORIAL



iSQL_Tutorial1.sql

This introductory tutorial will rapidly take you through the basic use of the powerful interactive SQL (iSQL) database interface. iSQL is a full featured command line client side query tool useful for submitting ad hoc SQL statements to a Server. Likewise the tool provides ample output formatting capabilities.

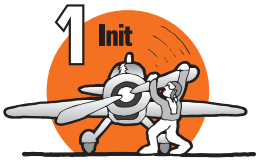
By no means does this introduction cover the full scope, detail, or flexibility that iSQL offers. It does however provide a quick glimpse of the benefits a tool such as this provides in a development environment.

This tutorial operates on the assumption that the database named 'myDatabase', already exists. Please refer to Section 3.6 “Introduction to the c-treeSQL ISQL Utility” in *c-tree Plus Quick Start and Product Overview Guide* for details on how to set up the environment for the tutorial.

This example, like all others in this set of documentation, will take the creation and use of a database and fit it into a simple four step flow of initialization, definition, management, and completion. (Init, define, manage, and you're done!)

Now let's break into the four areas.

2.1.1 Init



The initialize step is as simple as launching the iSQL tool. The syntax for this is as follows:

```
isql [-u user_name]  [-a password] [connect_string]
```

At the command line prompt type:

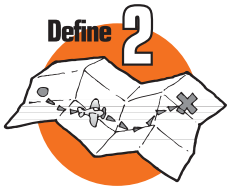
```
isql -u ADMIN -a ADMIN myDatabase
```

iSQL responds with the following prompt:

```
ISQL>
```

At this point, any valid SQL statement terminated with a semi-colon may be submitted.

2.1.2 Define



In this case define consists of the CREATE TABLE statement. This is done in a single iSQL statement in which specific fields are defined. Upon successful creation of the table, the changes made to the database by this transaction are made permanent by executing the COMMIT WORK statement. The following SQL syntax provides the functionality for the define phase:

- CREATE TABLE — Create a table.
- COMMIT WORK — Make changes permanent.

Below is the interactive SQL for DEFINE:

```
ISQL> CREATE TABLE CUSTMAST (
        cm_custnum    VARCHAR(5),
        cm_custzip    VARCHAR(10),
        cm_coststate  VARCHAR(3),
        cm_custrating VARCHAR(2),
        cm_custname   VARCHAR(48),
        cm_custaddrs  VARCHAR(48),
        cm_custcity   VARCHAR(48)  );

ISQL> COMMIT WORK CUSTMAST;
```

2.1.3 Manage

This step provides data management functionality for the application. We will simply add records to a table and then get and display those records. Then a simple record deletion is performed and the records are displayed again. The following SQL statements provide the functionality to manipulate the records in our table.

- INSERT INTO - This will add a record by inserting it into the table
- SELECT - Fetch records according to select criteria
- DELETE FROM - Delete records from a table.
- COMMIT WORK - Make changes permanent.

Below is the interactive SQL for MANAGE:

Add Records

```
ISQL> INSERT INTO CUSTMAST
VALUES ('1000', '92867', 'CA', '1', 'Bryan Williams', '2999
   Regency', 'Orange');

ISQL> INSERT INTO CUSTMAST
```

```
VALUES ('1001', '61434', 'CT', '1', 'Michael Jordan', '13 Main', 'Harford');

ISQL> INSERT INTO CUSTMAST
VALUES ('1002', '73677', 'GA', '1', 'Joshua Brown', '4356
   Cambridge', 'Atlanta');

ISQL> INSERT INTO CUSTMAST
VALUES ('1003', '10034', 'MO', '1', 'Keyon Dooling', '19771 Park
   Avenue', 'Columbia');

ISQL> COMMIT WORK;
```

Display Records

```
ISQL> SELECT * FROM CUSTMAST;
```

Delete Records

```
ISQL> DELETE FROM CUSTMAST;

ISQL> COMMIT WORK;
```

2.1.4 Done

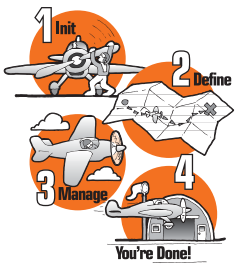
When a client application has completed operations with the server, it must release resources by disconnecting from the database. iSQL is an application that provides an interactive interface for SQL. It may not be explicit but a connection is made with the server when the isql tool is launched. Likewise, a disconnect occurs when the isql tool is exited.

Below is the interactive SQL for DONE:

```
ISQL> quit
```

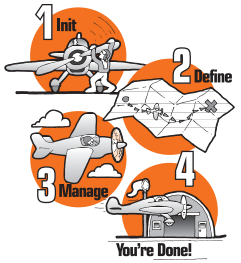
This will return the process back to a regular command line prompt.

2.1.5 Complete Introductory Tutorial Code



Complete source code for the introductory tutorial can be found in Appendix A "Tutorial Source Code".

2.2 RELATIONAL MODEL AND INDEXING TUTORIAL



iSQL_Tutorial2.sql

This intermediate tutorial will advance the concepts introduced in the first tutorial by expanding the number of tables and building a relational model. This tutorial will walk you through defining an index for each table, demonstrating the power of indexes in a relational model using a few simple API calls.

This tutorial operates on the assumption that the database named 'myDatabase', already exists. Please refer to Section 3.6 “Introduction to the c-treeSQL ISQL Utility” in *c-tree Plus Quick Start and Product Overview Guide* for details on how to set up the environment for the tutorial.

This example, like all others in this set of documentation, will take the creation and use of a database and fit it into a simple four step flow of initialization, definition, management, and completion. (Init, define, manage, and you're done!)

Now let's break into the four areas.

2.2.1 Init

The initialize step is as simple as launching the iSQL tool. The syntax for this is as follows:

```
isql [-u user_name]  [-a password] [connect_string]
```

At the command line prompt type:

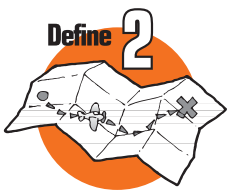
```
isql -u ADMIN -a ADMIN myDatabase
```

iSQL responds with the following prompt:

```
ISQL>
```

At this point, any valid SQL statement terminated with a semi-colon may be submitted.

2.2.2 Define



In this case define consists of the CREATE TABLE statement. This is done in a single iSQL statement in which specific fields are defined. Upon successful creation of the table, the changes made to the database by this transaction are made permanent by executing the COMMIT WORK statement.

Relational Database

This process of defining tables and indices is in actuality creating a relational database. For the sake of simplicity, we will not be enforcing constraints use in this tutorial. In this example there are 4 tables being defined as depicted in the drawing below. The fields that make up the index are shown in bold italics.

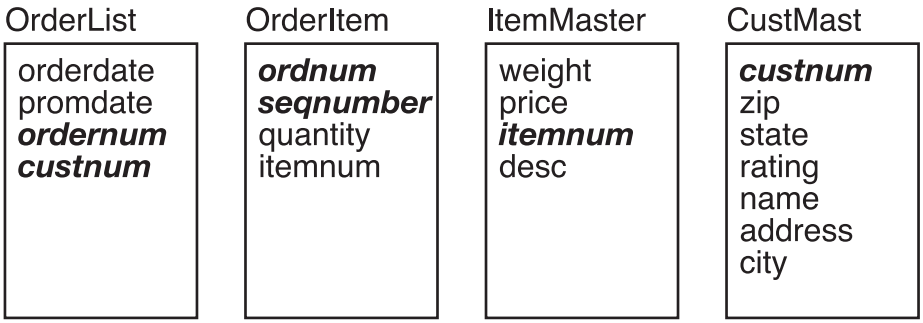
OrderList - A table of records consisting of a list of orders.

OrderItem - A table of records consisting of specific items associated with an order.

ItemMaster - A table of records consisting of information about items.

CustomerMaster - A table of records consisting of specific info related to each customer.

Each order (ordernum) in the ***orderlist*** table will contain 1 or more items (itemnum) in the ***orderitem*** table. Each item will have a corresponding definition (weight, price, description) in the ***itemmast*** table. An order is related to a specific customer (custnum) in the ***custmast*** table which contains information about each customer.



The following SQL syntax provides the functionality for the define phase:

- CREATE TABLE - Create a table.
- COMMIT WORK - Make changes permanent.

Below is the interactive SQL for DEFINE:

```
ISQL> CREATE TABLE orderlist (
   ol_orderdate DATE,
   ol_promdate  DATE,
   ol_ordernum  VARCHAR(7),
   ol_custnum   VARCHAR(4));

ISQL> CREATE INDEX custorder ON orderlist (ol_ordernum, ol_custnum);

ISQL> CREATE TABLE orderitems (
   oi_ordernum  VARCHAR(7),
   oi_seqnumber SMALLINT,
   oi_quantity  SMALLINT,
   oi_itemnum   VARCHAR(6));

ISQL> CREATE INDEX orderitem ON orderitems (oi_ordernum, oi_seqnumber);

ISQL> CREATE TABLE itemmast (
   im_weight  INTEGER,
   im_price   MONEY,
   im_itemnum VARCHAR(6),
   im_desc    VARCHAR(48));
```

```
ISQL> CREATE INDEX itemnum ON itemmast (im_itemnum);

ISQL> CREATE TABLE custmast (
   cm_custnum VARCHAR(5),
   cm_zip     VARCHAR(10),
   cm_state   VARCHAR(3),
   cm_rating  VARCHAR(2),
   cm_name    VARCHAR(48),
   cm_address VARCHAR(48),
   cm_city    VARCHAR(48));

ISQL> CREATE INDEX custnum ON custmast (cm_custnum);

ISQL> COMMIT WORK;
```

2.2.3 Manage

This step provides data management functionality for the application. In this example we will simply add records to the tables, process records in the form of a query, and finally display the results of the query.

This process involves fetching from 'OrderList' table, and for each record fetch related records from the 'OrderItems' table based on the index ordernum. The result is a recordset that amounts to a list of items associated with an order. The output of the manage step is a list of items that comprise an order showing name, quantity and price.

The following SQL syntax provides the functionality for the manage phase:

- INSERT INTO - This will add a record by inserting it into the table
- SELECT - Fetch records according to select criteria
- DELETE FROM - Delete records from a table.
- COMMIT WORK - Make changes permanent.

The following SQL statements populate all the tables with data.

```
INSERT INTO orderlist VALUES ('9/1/2002', '9/5/2002', '1', '1001');
INSERT INTO orderlist VALUES ('9/2/2002', '9/6/2002', '2', '1002');

INSERT INTO orderitems VALUES ('1',   1,  2,   '1');
INSERT INTO orderitems VALUES ('1',   2,  1,   '2');
INSERT INTO orderitems VALUES ('1',   3,  1,   '3');
INSERT INTO orderitems VALUES ('2',   1,  3,   '3');

INSERT INTO itemmast VALUES (10, 19.95,  '1', 'Hammer');
INSERT INTO itemmast VALUES (3,   9.99,  '2', 'Wrench');
INSERT INTO itemmast VALUES (4,  16.59,  '3', 'Saw');
INSERT INTO itemmast VALUES (1,   3.98,  '4', 'Plyers');
```

```
INSERT INTO custmast VALUES ('1000', '92867', 'CA', '1',
   'Bryan Williams', '2999 Regency', 'Orange');
INSERT INTO custmast VALUES ('1001', '61434', 'CT', '1',
   'Michael Jordan', '13 Main', 'Harford');
INSERT INTO custmast VALUES ('1002', '73677', 'GA', '1',
   'Joshua Brown', '4356 Cambridge', 'Atlanta');
INSERT INTO custmast VALUES ('1003', '10034', 'MO', '1',
   'Keyon Dooling', '19771 Park Avenue', 'Columbia');
COMMIT WORK;
```

The following SQL statement performs the query and displays a very small report. This query uses a join of 3 tables to list quantity and price for each item of an order. A basic introduction to the report generation capabilities provided by the iSQL Utility is shown in the form of the Column, Format and Heading syntax.

```
COLUMN cm_name FORMAT "A15"  heading "NAME"
COLUMN oi_quantity FORMAT "A10" heading "QTY"
COLUMN im_price FORMAT "$A10" heading "PRICE"
SELECT custmast.cm_name, orderitems.oi_quantity, itemmast.im_price
   FROM custmast, orderitems, itemmast, orderlist
   WHERE orderlist.ol_custnum = custmast.cm_custnum AND
   orderlist.ol_ordernum = orderitems.oi_ordernum AND
   orderitems.oi_itemnum = itemmast.im_itemnum
   ORDER BY orderlist.ol_custnum;
```

The report will appear as follows:

```
Name             QTY    PRICE
----             ---    -----
Michael Jordan    2     $19.95
Michael Jordan    1     $9.99
Michael Jordan    1     $16.59
Joshua Brown      3     $16.59
```

2.2.4 Done

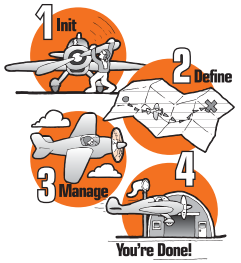
When a client application has completed operations with the server, it must release resources by disconnecting from the database. iSQL is an application that provides an interface for interactive SQL. It may not be explicit but a connection is made with the server when the isql tool is launched. Likewise, a disconnect occurs when the isql tool is exited.

Below is the interactive SQL for DONE:

```
ISQL> quit
```

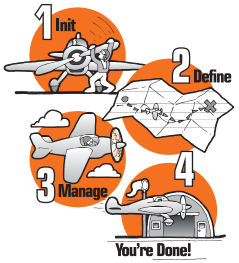
This will return the process back to a regular command line prompt.

2.2.5 Complete Relational Model and Indexing Tutorial Source Code



Complete source code for the relational model and indexing tutorial can be found in Appendix A "Tutorial Source Code".

2.3 LOCKING TUTORIAL



iSQL_Tutorial3.sql

This tutorial will introduce the concept of locking. The functionality for this tutorial focuses on adding records, then updating a single record to the customer master table. From the iSQL utility the script will be parsed and passed to the server. The script completes without "committing" the update. This leaves the updated record locked. The user will launch another instance of the Interactive SQL utility in another window which will block, waiting on the lock held by the first instance. Typing "COMMIT WORK;" from the first instance of the iSQL utility will complete the update transaction and remove the lock. This will allow the second instance to continue execution. Launching two processes provides a visual demonstration of the effects of locking and a basis for experimentation on your own.

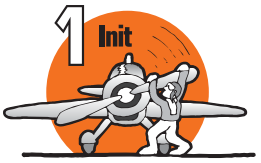
It is important to note that locking is inherent at the SQL level. This means that when one process has a record selected for update, it is locked even though there is no explicit SQL syntax.

This tutorial operates on the assumption that the database named 'myDatabase', already exists. Please refer to Section 3.6 “Introduction to the c-treeSQL ISQL Utility” in *c-tree Plus Quick Start and Product Overview Guide* for details on how to set up the environment for the tutorial.

This example, like all others in this set of documentation, will take the creation and use of a database and fit it into a simple four step flow of initialization, definition, management, and completion. (Init, define, manage, and you're done!)

Now let's break into the four areas.

2.3.1 Init



The initialize step is as simple as launching the iSQL tool. The syntax for this is as follows:

```
isql [-u user_name]  [-a password] [connect_string]
```

At the command line prompt type:

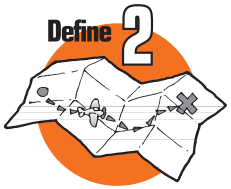
```
isql -u ADMIN -a ADMIN myDatabase
```

iSQL responds with the following prompt:

```
ISQL>
```

At this point, any valid SQL statement terminated with a semi-colon may be submitted.

2.3.2 Define



In this case define consists of the CREATE TABLE statement. This is done in a single iSQL statement in which specific fields are defined. Upon successful creation of the table, the changes made to the database by this transaction are made permanent by executing the COMMIT WORK statement. The following SQL syntax provides the functionality for the define phase:

- CREATE TABLE - Create a table.
- COMMIT WORK - Make changes permanent.

Below is the interactive SQL for DEFINE:

```
ISQL> CREATE TABLE CUSTMAST (
   cm_custnum    VARCHAR(5),
   cm_custzip    VARCHAR(10),
   cm_coststate  VARCHAR(3),
   cm_custrating VARCHAR(2),
   cm_custname   VARCHAR(48),
   cm_custaddrs  VARCHAR(48),
   cm_custcity   VARCHAR(48)  );

ISQL> COMMIT WORK CUSTMAST;
```

2.3.3 Manage

This step provides data management functionality for the application. In this example we will operate on the customer master table by first deleting all records, then adding, updating, and finally displaying the contents of the table. Deleting all records will ensure a clean starting point. Adding records will populate the table allowing subsequent record manipulation.

- INSERT INTO - This will add a record by inserting it into the table
- SELECT - Fetch records according to select criteria
- DELETE FROM - Delete records from a table.
- COMMIT WORK - Make changes permanent.

In order to run this tutorial you should execute:

```
@ISQL> @iSQL_Tutorial3.sql
```

in the first ISQL instance. When the script is finished, start the same tutorial in a second instance of ISQL.

The first execution will finish without committing the update. When the second process is launched, it will encounter a lock on a record it is trying to delete and will block.

The first process has the record associated with customer number 1003 locked. Meanwhile the second process has attempted to delete the contents of the customer master table and has been blocked when attempting to lock the table. At this point both processes are effectively blocked. The first process is holding the lock waiting on a keystroke and the second is waiting to grab the lock.

Typing "COMMIT WORK;" from the utility in the first process will unlock the record and allow execution to resume in the second process.

Below is the interactive SQL for MANAGE:

Delete Records

```
ISQL> DELETE FROM CUSTMAST;
```

Add Records

```
ISQL> INSERT INTO CUSTMAST
VALUES ('1000', '92867', 'CA', '1', 'Bryan Williams', '2999
   Regency', 'Orange');

ISQL> INSERT INTO CUSTMAST
VALUES ('1001', '61434', 'CT', '1', 'Michael Jordan', '13 Main', 'Harford');

ISQL> INSERT INTO CUSTMAST
VALUES ('1002', '73677', 'GA', '1', 'Joshua Brown', '4356 Cam
   bridge', 'Atlanta');

ISQL> INSERT INTO CUSTMAST
VALUES ('1003', '10034', 'MO', '1', 'Keyon Dooling', '19771 Park
   Avenue', 'Columbia');

ISQL> COMMIT WORK;
```

Update Record

```
ISQL> UPDATE custmast SET cm_name = 'KEYON DOOLING'  where cm_custnum = '1003';
```

2.3.4 Done

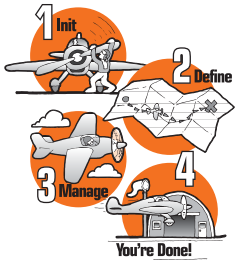
When a client application has completed operations with the server, it must release resources by disconnecting from the database. iSQL is an application that provides an interface for interactive SQL. It may not be explicit but a connection is made with the server when the isql tool is launched. Likewise, a disconnect occurs when the isql tool is quit.

Below is the interactive SQL for DONE:

```
ISQL> quit
```

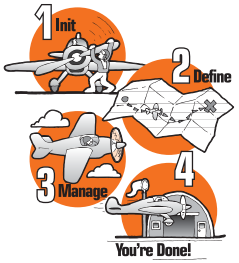
This will return the process back to a regular command line prompt.

2.3.5 Complete Locking Tutorial Source Code



Complete source code for the locking tutorial can be found in Appendix A "Tutorial Source Code".

2.4 TRANSACTION PROCESSING TUTORIAL



iSQL_Tutorial4.sql

This tutorial will introduce the concept of transaction processing, based on the relational model of the previous tutorial. Records will be added to tables orderlist and orderitems as a single transaction.

Transaction processing in iSQL is handled by two statements:

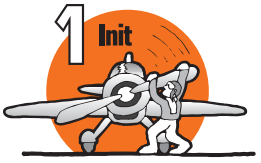
1. COMMIT WORK that makes the changes done during the transaction permanent and starts a new transaction.
2. ROLLBACK WORK that undoes all the changes done during the transaction, reverts the database to the status before the start of the transaction, and start a new transaction.

Notice that iSQL automatically starts a transaction when launched.

This example, like all others in this set of documentation, will take the creation and use of a database and fit it into a simple four step flow of initialization, definition, management, and completion. (Init, define, manage, and you're done!)

Now let's break into the four areas.

2.4.1 Init



The initialize step is as simple as launching the iSQL tool. The syntax for this is as follows:

```
isql [-u user_name]  [-a password] [connect_string]
```

At the command line prompt type:

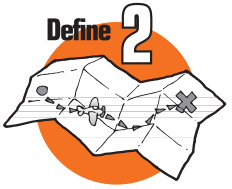
```
isql -u ADMIN -a ADMIN myDatabase
```

iSQL responds with the following prompt:

```
ISQL>
```

At this point, any valid SQL statement terminated with a semi-colon may be submitted.

2.4.2 Define



In this case define consists of the CREATE TABLE statement. This is done in a single iSQL statement in which specific fields are defined. Upon successful creation of the tables, the changes made to the database by this transaction are made permanent by executing the COMMIT WORK statement.

Transaction

These tables consist of a Customer Master table and an Item Master table that support primarily static information regarding a company's product line and customer demographics. The orderlist and orderitems tables consist of dynamic information pertinent to day to day sales. This dynamic data makes its way into the database as a transaction. If any part of the data is invalid then the transaction is rolled back and none of the data will enter the database. In this tutorial a transaction is comprised of the order and the items that make up an order.

The following SQL syntax provides the functionality for the define phase:

- CREATE TABLE - Create a table.
- COMMIT WORK - Make changes permanent.

Below is the interactive SQL for DEFINE:

```
ISQL> CREATE TABLE orderlist (
   ol_orderdate DATE,
   ol_promdate  DATE,
   ol_ordnum    VARCHAR(7),
   ol_custnum   VARCHAR(4));

ISQL> CREATE TABLE orderitems (
   oi_ordernum  VARCHAR(7),
   oi_seqnumber SMALLINT,
   oi_quantity  SMALLINT,
   oi_itemnum   VARCHAR(6));

ISQL> CREATE TABLE itemmast (
   im_weight   INTEGER,
   im_price    MONEY,
   im_itemnum  VARCHAR(6),
   im_desc     VARCHAR(48));

ISQL> CREATE TABLE custmast (
   cm_custnum  VARCHAR(5),
   cm_zip      VARCHAR(10),
   cm_state    VARCHAR(3),
   cm_rating   VARCHAR(2),
   cm_name     VARCHAR(48),
   cm_address  VARCHAR(48),
   cm_city     VARCHAR(48));

ISQL> COMMIT WORK;
```

2.4.3 Manage

This step provides data management functionality for the application. In this example we will add records to the itemmast and custmast tables, intended as static data. Then, sales orders will be processed as a transaction and "commited" or "rolled back" depending on the validity of the data. The final step will be to display the result of the transaction processing by dumping the contents of the orderlist and orderitems tables.

- INSERT INTO - This will add a record by inserting it into the table
- SELECT - Fetch records according to select criteria
- DELETE FROM - Delete records from a table.
- COMMIT WORK - Make changes permanent.

The following excerpt from iSQL_Tutorial4 shows the SQL syntax that would be used to implement the entry of an order and related items as a transaction. The items and the order are inserted, then a select statement is used to verify the existence of the item and customer number. iSQL is a tool that requires the user to interact. This script automatically implements the appropriate commit or rollback statement based on the results of the verification.

```
INSERT INTO orderitems VALUES ('1', 1,  2, '1');
INSERT INTO orderitems VALUES ('1', 2,  1, '2');
INSERT INTO orderlist VALUES ('9/1/2002', '9/5/2002', '1', '1001');
SELECT orderitems.oi_itemnum, itemmast.im_itemnum
   FROM orderitems, itemmast
   WHERE orderitems.oi_itemnum = itemmast.im_itemnum;
SELECT orderlist.ol_custnum, custmast.cm_custnum
   FROM orderlist, custmast
   WHERE orderlist.ol_custnum = custmast.cm_custnum;
COMMIT WORK;

INSERT INTO orderitems VALUES ('2', 1,  1, '3');
INSERT INTO orderitems VALUES ('2', 2,  3, '4');
INSERT INTO orderlist VALUES ('9/2/2002', '9/6/2002', '2', '9999');
SELECT orderitems.oi_itemnum, itemmast.im_itemnum
   FROM orderitems, itemmast
   WHERE orderitems.oi_itemnum = itemmast.im_itemnum;
SELECT orderlist.ol_custnum, custmast.cm_custnum
   FROM orderlist, custmast
   WHERE orderlist.ol_custnum = custmast.cm_custnum;
ROLLBACK WORK;

INSERT INTO orderitems VALUES ('3', 1,  2, '3');
INSERT INTO orderitems VALUES ('3', 2,  3, '99');
INSERT INTO orderlist VALUES ('9/22/2002', '9/26/2002', '3', '1002');
SELECT orderitems.oi_itemnum, itemmast.im_itemnum
   FROM orderitems, itemmast
   WHERE orderitems.oi_itemnum = itemmast.im_itemnum;
```

```
SELECT orderlist.ol_custnum, custmast.cm_custnum
   FROM orderlist, custmast
   WHERE orderlist.ol_custnum = custmast.cm_custnum;
ROLLBACK WORK;
```

2.4.4 Done

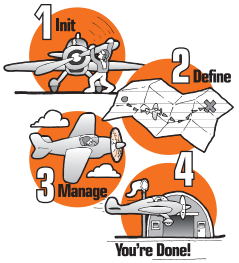
When a client application has completed operations with the server, it must release resources by disconnecting from the database. iSQL is an application that provides an interface for interactive SQL. It may not be explicit but a connection is made with the server when the isql tool is launched. Likewise, a disconnect occurs when the isql tool is exited.

Below is the interactive SQL for DONE:

```
ISQL> quit
```

This will return the process back to a regular command line prompt.

2.4.5 Complete Transaction Processing Tutorial Source Code



Complete source code for the transaction processing tutorial can be found in Appendix A "Tutorial Source Code".

Chapter 3

ISQL Statements

3.1 OVERVIEW

This chapter describes only those statements that are specific to ISQL. See the *c-treeSQL Reference Guide* for detailed reference information on standard SQL statements that can be issued in other environments.

3.2 STARTING INTERACTIVE SQL

Start ISQL by issuing the isql command at the shell prompt. c-treeSQL invokes ISQL and displays the ISQL prompt:

```
$ isql sampledb
       c-treeSQL Interactive Interpreter

ISQL>
```

Issue c-treeSQL statements at the ISQL> prompt and terminate them with a semicolon. You can continue statements on multiple lines. ISQL automatically prompts for continuation lines until you terminate the statement with a semicolon.

To execute host operating system commands from the ISQL prompt, type HOST followed by the operating system command. After completion of the HOST statement, the ISQL> prompt returns. To execute SQL scripts from ISQL, type @ followed by the name of the file containing SQL statements.

To exit from interactive SQL, type EXIT or QUIT.

You can supply optional switches and arguments to the isql command.

Syntax

```
isql [-s script_file] [-u user_name] [-a password] [connect_string]
```

Arguments

-s script_file
The name of an SQL script file that c-treeSQL executes when it invokes ISQL.

Note: For Windows platforms, if the file name has a space, such as:
test script.sql

The file name must be enclosed in doubles quotes, such as:
isql -s "test script.sql" testdb

-u user_name
The user name c-treeSQL uses to connect to the database specified in the *connect_string*. c-treeSQL verifies the user name against a corresponding password before it connects to the database. If omitted, the default value depends on the environment. (On UNIX, the value of the DH_USER environment variable specifies the default user name. If DH_USER is not set, the value of the USER environment variable specifies the default user name.)

-a password
The password c-treeSQL uses to connect to the database specified in the *connect_string*. c-treeSQL verifies the password against a corresponding user name before it connects to the database. If omitted, the default value depends on the environment. (On UNIX, the value of the DH_PASSWD environment variable specifies the default password.)

connect_string
A string that specifies which database to connect to. The *connect_string* can be a simple database name or a complete connect string. For example, to connect to a local database named myDatabase, you would use the following syntax:

```
isql -u ADMIN -a ADMIN myDatabase
```

To connect to a remote database named c-treeSQL, you would use the 6597@remotehost:database syntax as follows:

```
isql -u ADMIN -a ADMIN  6597@hotdog.faircom.com:ctreeSQL
```

See the CONNECT statement in the *c-treeSQL Reference Manual* for details on how to specify a complete connect string. If omitted, the default value depends on the environment. (On UNIX, the value of the DB_NAME environment variable specifies the default connect string.)

3.3 STATEMENT HISTORY SUPPORT

ISQL provides statements to simplify the process of executing statements you already typed. ISQL implements a history mechanism similar to the one found in the *csh* (C-shell) supported by UNIX.

The following table summarizes the ISQL statements that support retrieving, modifying, and rerunning previously entered statements.

Table 3-1: ISQL Statements for Statement History Support

Statement	Summary
HISTORY	Displays a fixed number of statements (specified by the SET HISTORY statement) which have been entered before this statement, along with a statement number for each statement. Other statements take the statement number as an argument. See Section 3.7.12 "HISTORY" on page 3-29 for details.

Table 3-1: ISQL Statements for Statement History Support

Statement	Summary
RUN [stmt_num]	Displays and executes the current statement or specified statement in the history buffer. See Section 3.7.16 "RUN" on page 3-33 details.
LIST [stmt_num]	Displays the current statement or specified statement in the history buffer, and makes that statement the current statement by copying it to the end of the history list. See Section 3.7.14 "LIST" on page 3-32 for details.
EDIT [stmt_num]	Edits the current statement or specified statement in the history buffer, and makes the edited statement the current statement by copying it to the end of the history list. The environment variable EDITOR can be set to the editor of choice. See Section 3.7.8 "EDIT" on page 3-25 for details.
SAVE filename	Saves the current statement in the history buffer to the specified file, which can then be retrieved through the GET or START statements. See Section 3.7.17 "SAVE" on page 3-34 for details.
GET filename	Fetches the contents of the specified file, from the beginning of the file to the first semicolon, and appends it to the history buffer. The statement fetched by the GET can then be executed by using the RUN statement. See Section 3.7.10 "GET" on page 3-27 for details.
START filename [argument …]	Fetches and executes a statement stored in the specified file. Unlike the GET statement, START executes the statement and accepts arguments that it substitutes for parameter references in the statement stored in the file. START also appends the statement to the history buffer. See Section 3.7.21 "START" on page 3-39 for details.

3.4 FORMATTING OUTPUT OF ISQL QUERIES

Formatting of database query results makes the output of a database query more presentable and understandable. The formatted output of an ISQL database query can be either displayed on the screen, written to a file, or spooled to a printer to produce a hardcopy of the report.

ISQL includes several statements that provide simple formatting of SQL queries. The following table summarizes the ISQL query-formatting statements.

Table 3-2: ISQL Statements for Query Formatting

Statement	Summary
DISPLAY	Displays text, variable values, and/or column values after the specified set of rows (called a break specification). See Section 3.7.7 "DISPLAY" on page 3-23 for details.
COMPUTE	Performs aggregate-function computations on column values for the specified set of rows, and assigns the results to a variable. DISPLAY statements can then refer to the variable to display its value. See Section 3.7.5 "COMPUTE" on page 3-21 for details.
BREAK	Specifies at what point ISQL processes associated DISPLAY and COMPUTE statements. BREAK statements can specify that processing occurs after a change in a column's value, after each row, after each page, or at the end of a query. DISPLAY and COMPUTE statements have no effect until you issue a BREAK statement with the same break specification. See Section 3.7.2 "BREAK" on page 3-13 for details.
DEFINE	Defines a variable and assigns a text value to it. When DISPLAY statements refer to the variable, ISQL prints the value. See Section 3.7.6 "DEFINE" on page 3-23 for details.
COLUMN	Controls how ISQL displays a column's values (the FORMAT clause) and/or specifies alternative column-heading text (the HEADING clause). See Section 3.7.4 "COLUMN" on page 3-16 for details.
TITLE	Specifies text and its positioning that ISQL displays before or after it processes a query. See Section 3.7.23 "TITLE" on page 3-42 for details.
CLEAR	Removes settings made by the previous DISPLAY, COMPUTE, COLUMN, BREAK, DEFINE, or TITLE statements. See Section 3.7.3 "CLEAR" on page 3-15 for details.
SET LINESIZE SET PAGESIZE SET REPORT SET ECHO	Specifies various attributes that affect how ISQL displays queries and results.

The rest of this section provides an extended example that illustrates how to use the statements together to improve formatting.

All the examples use the same ISQL query. The query retrieves data on outstanding customer orders. The query joins two tables, *customers* and *orders*. The examples for the TABLE statement on Section 3.7.13 "HOST or SH or !" on page 3-31 show the columns and data types for these sample tables.

The following example shows the query and an excerpt of the results as ISQL displays them without the benefit of any query-formatting statements:

Example 3-1: Unformatted Query Display from ISQL

```
ISQL> select c.customer_name, c.customer_city, o.order_id, o.order_value
   from customers c, orders o
   where o.customer_id = c.customer_id
   order by c.customer_name;

CUSTOMER_NAME                                      CUSTOMER_CITY
-------------                                      -------------
                                 ORDER_ID  ORDER_VALUE
                                 --------  -----------
Aerospace Enterprises Inc.                                Scottsdale
                                       13      3000000
Aerospace Enterprises Inc.                                Scottsdale
                                       14      1500000
Chemical Construction Inc.                                     Joplin
                                       11      3000000
Chemical Construction Inc.                                     Joplin
                                       12      7500000
Luxury Cars Inc.                                        North Ridgeville
                                       21      6000000
Luxury Cars Inc.                                        North Ridgeville
                                       20      5000000
```

Although this query retrieves the correct data, the formatting is inadequate:

- The display for each record wraps across two lines, primarily because of the column definitions for the text columns *customer_name* and *customer_city*. ISQL displays the full column width (50 characters for each column) even though the contents don't use the entire width.
- It's not clear that the values in the *order_value* column represent money amounts.

The next section shows how to use the COLUMN statement to address these formatting issues.

In addition, you can use DISPLAY, COMPUTE, and BREAK statements to present order summaries for each customer. Section 3.4.2 "Summarizing Data with DISPLAY, COMPUTE, and BREAK Statements" on page 3-7 shows how to do this. Finally, you can add text that ISQL displays at the beginning and end of query results with the TITLE statement, as described in Section 3.4.3 "Adding Beginning and Concluding Titles with the TITLE Statement" on page 3-9.

All of these statements are independent of the actual query. You do not need to change the query in any way to control how ISQL formats the results.

3.4.1 Formatting Column Display with the COLUMN Statement

You can specify the width of the display for character columns with the COLUMN statement's "An" format string. Specify the format string in the FORMAT clause of the COLUMN statement. You need to issue separate COLUMN statements for each column whose width you want to control in this manner.

The following example shows COLUMN statements that limit the width of the *customer_name* and *customer_city* columns, and re-issues the original query to show how they affect the results.

Example 3-2: Controlling Display Width of Character Columns

```
ISQL> COLUMN CUSTOMER_NAME FORMAT "A19"
ISQL> COLUMN CUSTOMER_CITY FORMAT "A19"
ISQL> select c.customer_name, c.customer_city, o.order_id, o.order_value
   from customers c, orders o
   where o.customer_id = c.customer_id
   order by c.customer_name;

CUSTOMER_NAME       CUSTOMER_CITY           ORDER_ID  ORDER_VALUE
-------------       -------------           --------  -----------
Aerospace Enterpris Scottsdale                    13      3000000
Aerospace Enterpris Scottsdale                    14      1500000
Chemical Constructi Joplin                        11      3000000
Chemical Constructi Joplin                        12      7500000
Luxury Cars Inc.    North Ridgeville              21      6000000
Luxury Cars Inc.    North Ridgeville              20      5000000
```

Note that ISQL truncates display at the specified width. This means you should specify a value in the FORMAT clause that accommodates the widest column value that the query will display.

To improve the formatting of the *order_value* column, use the COLUMN statement's numeric format strings. Issue another COLUMN statement, this one for *order-_value*, and specify a format string using the "$", "9", and "," format-string characters:

- The format-string character 9 indicates the width of the largest number. Specify enough 9 format-string characters to accommodate the largest value in the column.
- The format-string character $ directs ISQL to precede column values with a dollar sign.
- The comma (,) format-string character inserts a comma at the specified position in the display.

For the *order_value* column, the format string "$99,999,999.99" displays values in a format that clearly indicates that the values represent money. (For a complete list of the valid numeric format characters, see Table 3-3: Numeric Format Strings for the COLUMN Statement on page 3-18.)

The following example shows the complete COLUMN statement that formats the *order_value* column. As shown by issuing the COLUMN statement without any arguments, this example retains the formatting from the COLUMN statements in the previous example.

Example 3-3: Customizing Format of Numeric Column Displays

```
ISQL> column order_value format "$99,999,999.99"
ISQL> column; -- Show all the COLUMN statements now in effect:
column CUSTOMER_NAME format "A19"  heading  "CUSTOMER_NAME"
column CUSTOMER_CITY format "A19"  heading  "CUSTOMER_CITY"
column ORDER_VALUE format "$99,999,999.99" heading  "ORDER_VALUE"
ISQL> select c.customer_name, c.customer_city, o.order_id, o.order_value
   from customers c, orders o
   where o.customer_id = c.customer_id
   order by c.customer_name;
CUSTOMER_NAME       CUSTOMER_CITY    ORDER_ID   ORDER_VALUE
-------------       -------------    --------   -----------
Aerospace Enterpris  Scottsdale            13   $3,000,000.00
Aerospace Enterpris  Scottsdale            14   $1,500,000.00
Chemical Constructi  Joplin                11   $3,000,000.00
Chemical Constructi  Joplin                12   $7,500,000.00
Luxury Cars Inc.     North Ridgeville      21   $6,000,000.00
Luxury Cars Inc.     North Ridgeville      20   $5,000,000.00
.
.
.
```

3.4.2 Summarizing Data with DISPLAY, COMPUTE, and BREAK Statements

Now that the query displays the rows it returns in a more acceptable format, you can use DISPLAY, COMPUTE, and BREAK statements to present order summaries for each customer.

All three statements rely on a break specification to indicate to ISQL when it should perform associated processing. There are four types of breaks you can specify:

- Column breaks are processed whenever the column associated with the break changes in value
- Row breaks are processed after display of each row returned by the query
- Page breaks are processed at the end of each page (as defined by the SET PAGESIZE statement)
- Report breaks are processed after display of all the rows returned by the query

While DISPLAY and COMPUTE statements specify what actions ISQL takes for a particular type of break, the BREAK statement itself controls which type of break is currently in effect. A consequence of this behavior is that DISPLAY and COMPUTE statements don't take effect until you issue the BREAK statement with the corresponding break specification.

Also, keep in mind that there can be only one type of break in effect at a time. This means you can format a particular query for a single type of break.

In our example, we are interested in a column break, since we want to display an order summary for each customer. In particular, we want to display the name of the customer along with the number and total value of orders for that customer. And, we want this summary displayed

whenever the value in the *customer_name* column changes. In other words, we need to specify a column break on the *customer_name* column.

Approach this task in two steps. First, devise a DISPLAY statement to display the customer name and confirm that it is displaying correctly. Then, issue COMPUTE statements to calculate the statistics for each customer (namely, the count and sum of orders), and add DISPLAY statement to include those statistics. All of the DISPLAY, COMPUTE and BREAK statements have to specify the same break to get the desired results.

The following example shows the DISPLAY and BREAK statements that display the customer name. The COL clause in the DISPLAY statement indents the display slightly to emphasize the change in presentation.

The following example uses the column formatting from previous examples. Notice that the column formatting also affects DISPLAY statements that specify the same column.

Example 3-4: Specifying Column Breaks and Values with DISPLAY

```
ISQL> display col 5 "Summary of activity for", customer_name on customer_name;
ISQL> break on customer_name
ISQL> select c.customer_name, c.customer_city, o.order_id, o.order_value
   from customers c, orders o
   where o.customer_id = c.customer_id
   order by c.customer_name;
CUSTOMER_NAME       CUSTOMER_CITY           ORDER_ID   ORDER_VALUE
-------------       -------------           --------   -----------
Aerospace Enterpris Scottsdale                    13  $3,000,000.00
Aerospace Enterpris Scottsdale                    14  $1,500,000.00
     Summary of activity for Aerospace Enterpris
Chemical Constructi Joplin                        11  $3,000,000.00
Chemical Constructi Joplin                        12  $7,500,000.00
     Summary of activity for Chemical Constructi
Luxury Cars Inc.    North Ridgeville              21  $6,000,000.00
Luxury Cars Inc.    North Ridgeville              20  $5,000,000.00
     Summary of activity for Luxury Cars Inc.
.
.
.
```

Next, issue two COMPUTE statements to calculate the desired summary values.

COMPUTE statements specify an SQL aggregate function (AVG, MIN, MAX, SUM, or COUNT), a column name, a variable name, and a break specification. ISQL applies the aggregate function to all values of the column for the set of rows that corresponds to the break specification. It stores the result in the variable, which subsequent DISPLAY statements can use to display the result.

For this example, you need two separate compute statements. One calculates the number of orders (COUNT OF the *order_id* column) and the other calculates the total cost of orders (SUM OF the *order_value* column). Both specify the same break, namely, *customer_name*. The following example shows the COMPUTE statements, which store the resulting value in the variables *num_orders* and *tot_value*.

The following example also issues two more DISPLAY statements to display the variable values. As before, the DISPLAY statements must specify the *customer_name* break. They also indent their display farther to indicate the relationship with the previously issued DISPLAY.

As before, this example uses the COLUMN and DISPLAY statements from previous examples. ISQL processes DISPLAY statements in the order they were issued. Use the DISPLAY statement, without any arguments, to show the current set of DISPLAY statements in effect. Also, in the query results, notice that the column formatting specified for the *order_value* column carries over to the *tot_value* variable, which is based on *order_value*.

Example 3-5: Calculating Statistics on Column Breaks with COMPUTE

```
ISQL> compute count of order_id in num_orders on customer_name
ISQL> compute sum of order_value in tot_value on customer_name
ISQL> display col 10 "Total number of orders:", num_orders on customer_name;
ISQL> display col 10 "Total value of orders:", tot_value on customer_name;
ISQL> display  -- See all the DISPLAY statements currently active:
display  col 5  "Summary of activity for" ,customer_name  on customer_name
display  col 10  "Total number of orders:" ,num_orders  on customer_name
display  col 10  "Total value of orders:" ,tot_value  on customer_name
ISQL> select c.customer_name, c.customer_city, o.order_id, o.order_value
   from customers c, orders o
   where o.customer_id = c.customer_id
   order by c.customer_name;
CUSTOMER_NAME       CUSTOMER_CITY           ORDER_ID    ORDER_VALUE
-------------       -------------           --------    -----------
Aerospace Enterpris Scottsdale                    13   $3,000,000.00
Aerospace Enterpris Scottsdale                    14   $1,500,000.00
     Summary of activity for Aerospace Enterpris
          Total number of orders:        2
          Total value of orders:   $4,500,000.00
Chemical Constructi Joplin                        11   $3,000,000.00
Chemical Constructi Joplin                        12   $7,500,000.00
     Summary of activity for Chemical Constructi
          Total number of orders:        2
          Total value of orders:  $10,500,000.00
Luxury Cars Inc.   North Ridgeville               21   $6,000,000.00
Luxury Cars Inc.   North Ridgeville               20   $5,000,000.00
     Summary of activity for Luxury Cars Inc.
          Total number of orders:        2
          Total value of orders:  $11,000,000.00
.
.
.
```

3.4.3 Adding Beginning and Concluding Titles with the TITLE Statement

You can add some finishing touches to the query display with the TITLE statement.

The TITLE statement lets you specify text that ISQL displays before (TITLE TOP) or after (TITLE BOTTOM) the query results.

The title can also be horizontally positioned by specifying the keywords LEFT, CENTER, or RIGHT; or by specifying the actual column number corresponding to the required positioning of the title. Use the SKIP clause to skip lines after a top title or before a bottom title.

The following example uses two TITLE statements to display a query header and footer.

Example 3-6: Specifying a Query Header and Footer with TITLE

```
ISQL> TITLE TOP LEFT "Orders Summary" RIGHT "September 29, 1998" SKIP 2;
ISQL> SHOW LINESIZE  -- RIGHT alignment of TITLE is relative to this value:
LINESIZE ..................... : 78
ISQL> TITLE BOTTOM CENTER "End of Orders Summary Report" SKIP 2;
ISQL> select c.customer_name, c.customer_city, o.order_id, o.order_value
   from customers c, orders o
   where o.customer_id = c.customer_id
   order by c.customer_name;
Orders Summary                                        September 29, 1998


CUSTOMER_NAME       CUSTOMER_CITY           ORDER_ID    ORDER_VALUE
-------------       -------------           --------    -----------
Aerospace Enterpris Scottsdale                    13   $3,000,000.00
Aerospace Enterpris Scottsdale                    14   $1,500,000.00
     Summary of activity for Aerospace Enterpris
          Total number of orders:        2
          Total value of orders:   $4,500,000.00
Chemical Constructi Joplin                        11   $3,000,000.00
Chemical Constructi Joplin                        12   $7,500,000.00
     Summary of activity for Chemical Constructi
          Total number of orders:        2
          Total value of orders:  $10,500,000.00
Luxury Cars Inc.    North Ridgeville              21   $6,000,000.00
Luxury Cars Inc.    North Ridgeville              20   $5,000,000.00
     Summary of activity for Luxury Cars Inc.
          Total number of orders:        2
          Total value of orders:  $11,000,000.00
.
.
.
Tower Construction  Munising                       8   $2,000,000.00
Tower Construction  Munising                      10   $6,000,000.00
Tower Construction  Munising                       9   $8,000,000.00
     Summary of activity for Tower Construction
          Total number of orders:        3
          Total value of orders:  $16,000,000.00
```

```
                              End of Orders Summary Report
23 records selected
ISQL>
```

3.5 THE HELP AND TABLE STATEMENTS

ISQL supports an on-line help facility that can be invoked by using the HELP statement. Typing HELP at the ISQL prompt will display a help file which will list the options accepted by the HELP statement. The various forms of the HELP statement are listed below:

- HELP - Displays the options that can be specified for HELP.
- HELP COMMANDS - Displays all the statements that ISQL accepts.
- HELP *command_name* - Displays help file corresponding to the specified statement.

TABLE is an ISQL statement that displays all the tables present in the database including any system tables. TABLE can be used also to display the description of a single table by explicitly giving the table name. Both forms of the TABLE statement are shown below:

```
TABLE;
TABLE table_name;
```

3.6 TRANSACTION SUPPORT

A transaction is started with the execution of the first SQL statement. A transaction is committed using the COMMIT WORK statement and rolled back using the ROLLBACK WORK statement.

If the AUTOCOMMIT option is set to ON, then ISQL treats each SQL statement as a single transaction. This prevents the user from holding locks on the database for an extended period of time. This is very critical when the user is querying an on-line database in which a transaction processing application is executing in real time.

A set of SQL statements can be executed as part of a transaction and committed using the COMMIT WORK statement. This is shown below:

```
<SQL statement>

<SQL statement>

<SQL statement>

COMMIT WORK ;
```

Instead, a transaction can also be rolled back using the ROLLBACK WORK statement as shown:

```
<SQL statement>

<SQL statement>
```

```
<SQL statement>

ROLLBACK WORK ;
```

An SQL statement starting immediately after a COMMIT WORK or ROLLBACK WORK statement starts a new transaction.

3.7 ISQL REFERENCE

This section provides reference material for statements specific to ISQL.

This section does not include descriptions of standard SQL statements or statements specific to embedded SQL. For details on the syntax and semantics of those other SQL statements, see the *c-treeSQL Reference Manual*.

3.7.1 @ (Execute)

Syntax

```
@filename
```

Description

Executes the SQL statements stored in the specified SQL script file. The statements specified in the file are not added to the history buffer.

Arguments

filename
The name of the script file.

Notes

The GET, START, and @ (execute) statements are similar in that they all read SQL script files. Both GET and START read an SQL script file and append the first statement in it to the history buffer. However, the START statement also executes the script statement and accepts arguments that it substitutes for parameter references in the script statement. The @ (execute) statement, on the other hand, executes all the statements in an SQL script file but does not add any of the statements to the history buffer. The @ statement does not support argument substitution.

Example

The following example shows a simple ISQL script file.

Example 3-7: Sample ISQL script

```
connect to demodb;
set echo on ;
create table stores (item_no integer, item_name char(20));
```

```
insert into stores values (1001,chassis);
insert into stores values (1002,chips);
select * from stores where item_no > 1001;
set echo off ;
```

To execute the above statements stored in a file named *cmdfile*, enter:

```
ISQL> @cmdfile
```

3.7.2 BREAK

Syntax

```
BREAK [ ON break_spec [ SKIP n ] ] ;
break_spec::
       { column_name [ , … ] | ROW | PAGE | REPORT }
```

Description

The BREAK statement specifies at what point ISQL processes associated DISPLAY and COMPUTE statements. DISPLAY and COMPUTE statements have no effect until you issue a BREAK statement with the same break specification.

A break can be specified on any of the following events:

- Change in the value of a column
- Selection of each row
- End of a page
- End of a report

Only one BREAK statement can be in effect at a time. When a new BREAK statement is entered, it replaces the previous BREAK statement. The BREAK statement can specify one or more columns on which the break can occur.

The BREAK statement without any clauses displays the currently-set break, if any.

Arguments

break_spec
The events that cause an SQL query to be interrupted and the execution of the associated COMPUTE and DISPLAY statements. *break_spec* can be any of the following values:

column_name	Causes a break when the value of the column specified by *column_name* changes.
ROW	Causes a break on every row selected by a SELECT statement.

PAGE	Causes a break at the end of each page. The end of a page is specified in the SET PAGESIZE statement. See Section 3.7.18 "SET" on page 3-34 for details on the SET statement.
REPORT	Causes a break at the end of a report or query.

SKIP n

The optional SKIP clause can be used to skip the specified number of lines when the specified break occurs and before processing of any associated DISPLAY statements.

Examples

The following examples illustrate how various break settings and corresponding DISPLAY statements affect the display of the same query.

```
ISQL> break
no break specified
ISQL> select customer_name from customers;  -- Default display
CUSTOMER_NAME
-------------
Sports Cars Inc.
Mighty Bulldozer Inc.
Ship Shapers Inc.
Tower Construction Inc.
Chemical Construction Inc.
Aerospace Enterprises Inc.
Medical Enterprises Inc.
Rail Builders Inc.
Luxury Cars Inc.
Office Furniture Inc.
10 records selected
ISQL> -- Set DISPLAY values for different breaks:
ISQL> display "Break on change in value of customer_name!" on customer_name;
ISQL> display "Break on every row!" on row;
ISQL> display "Break on page (page size set to 2 lines)" on page;
ISQL> display "Break on end of report!" on report;
ISQL> set pagesize 2
ISQL> break on customer_name
ISQL> select customer_name from customers;
CUSTOMER_NAME
-------------
Sports Cars Inc.
Break on change in value of customer_name!
Mighty Bulldozer Inc.
Break on change in value of customer_name!
Ship Shapers Inc.
Break on change in value of customer_name!
.
.
.
ISQL> break on row
```

```
ISQL> select customer_name from customers;
CUSTOMER_NAME
-------------
Sports Cars Inc.
Break on every row!
Mighty Bulldozer Inc.
Break on every row!
Ship Shapers Inc.
Break on every row!
.
.
.
ISQL> break on page
ISQL> select customer_name from customers;
CUSTOMER_NAME
-------------
Break on page (page size set to 2 lines)
CUSTOMER_NAME
-------------
Sports Cars Inc.
Break on page (page size set to 2 lines)
CUSTOMER_NAME
-------------
Mighty Bulldozer Inc.
Break on page (page size set to 2 lines)
.
.
.
ISQL> break on report
ISQL> select customer_name from customers;
CUSTOMER_NAME
-------------
Sports Cars Inc.
Mighty Bulldozer Inc.
Ship Shapers Inc.
Tower Construction Inc.
Chemical Construction Inc.
Aerospace Enterprises Inc.
Medical Enterprises Inc.
Rail Builders Inc.
Luxury Cars Inc.
Office Furniture Inc.
Break on end of report!
10 records selected
ISQL>
```

3.7.3 CLEAR

Syntax

```
CLEAR option ;
option::
        HISTORY
```

```
|  BREAK
|  COLUMN
|  COMPUTE
|  DISPLAY
|  TITLE
```

Description

The CLEAR statement removes settings made by the ISQL statement corresponding to option.

Argument

option
Which ISQL statement's settings to clear:

- CLEAR HISTORY - Clears the ISQL statement history buffer.
- CLEAR BREAK - Clears the break set by the BREAK statement.
- CLEAR COLUMN - Clears formatting options set by any COLUMN statements in effect.
- CLEAR COMPUTE - Clears all the options set by the COMPUTE statement.
- CLEAR DISPLAY - Clears the displays set by the DISPLAY statement.
- CLEAR TITLE - Clears the titles set by the TITLE statement.

Examples

The following example illustrates clearing the DISPLAY and BREAK settings.

```
ISQL> DISPLAY  -- See the DISPLAY settings currently in effect:
display  "Break on change in value of customeer_name!"  on customer_name
display  "Break on every row!"  on row
display  "Break on page (page size set to 2 lines)"  on page
display  "Break on end of report!"  on report
ISQL> CLEAR DISPLAY
ISQL> DISPLAY
No display specified.
ISQL> BREAK
break on report skip 0
ISQL> CLEAR BREAK
ISQL> BREAK
no break specified
ISQL>
```

3.7.4 COLUMN

Syntax

```
COLUMN [ column_name
[ FORMAT " format_string " ] |  [ HEADING " heading_text " ] ] ;
```

Description

The COLUMN statement controls how ISQL displays a column's values (the FORMAT clause) and specifies alternative column-heading text (the HEADING clause).

The COLUMN statement without any arguments displays the current column specifications.

Arguments

column_name
The name of the column affected by the COLUMN statement. If the COLUMN statement includes *column_name* but omits both the FORMAT and HEADING clauses, ISQL clears any formatting and headings in effect for that column. The formatting specified for *column_name* also applies to DISPLAY statements that specify the same column.

FORMAT " format_string "
Specifies a quoted string that formats the display of column values. Valid values for format strings depend on the data type of the column.

Character	The only valid format string for character data types is of the form "An", where n specifies the width of the column display. The A character must be upper case.
Numeric	Table 3-3: Numeric Format Strings for the COLUMN Statement on page 3-18 shows valid format strings for numeric data types.
Date-time	Table 3-4: Date-Time Format Strings for the COLUMN Statement on page 3-18 shows valid format strings for date-time data types. The format strings consist of keywords that SQL interprets and replaces with formatted values. Any other character in the format string are displayed as literals. The format strings are case sensitive. For instance, SQL replaces 'DAY' with all uppercase letters, but follows the case of 'Day'. Note that the SQL scalar function TO_CHAR offers comparable functionality and is not limited to SQL statements issued within ISQL. See the *c-treeSQL Reference Manual* for details on TO_CHAR.

COLUMN format strings also affect display values in DISPLAY statements that specify the same column or a COMPUTE value based on the column.

HEADING " heading_text "
Specifies an alternative heading for the column display. The default is the column name.

(a) Format String Details

Table 3-3: Numeric Format Strings for the COLUMN Statement

Character	Example	Description
9	99999	Number of 9's specifies width. If the column value is too large to display in the specified format, ISQL displays # characters in place of the value.
0	09999	Display leading zeroes.
$	$9999	Prefix the display with '$'.
B	B9999	Display blanks if the value is zero.
,	99,999	Display a comma at position specified by the comma.
.	99,999.99	Display a decimal point at the specified position.
MI	99999MI	Display '-' after a negative value.
PR	99999PR	Display negative values between '<' and '>'.

Table 3-4: Date-Time Format Strings for the COLUMN Statement

Character	Description
CC	The century as a 2-digit number.
YYYY	The year as a 4-digit number.
YYY	The last 3 digits of the year.
YY	The last 2 digits of the year.
Y	The last digit of the year.
Y,YYY	The year as a 4-digit number with a comma after the first digit.
Q	The quarter of the year as 1-digit number (with values 1, 2, 3, or 4).
MM	The month value as 2-digit number (in the range 01-12).
MONTH	The name of the month as a string of 9 characters ('JANUARY' to 'DECEMBER ').
MON	The first 3 characters of the name of the month (in the range 'JAN' to 'DEC').
WW	The week of year as a 2-digit number (in the range 01-52).

Table 3-4: Date-Time Format Strings for the COLUMN Statement

Character	Description
W	The week of month as a 1-digit number (in the range 1-5).
DDD	The day of year as a 3-digit number (in the range 001-365).
DD	The day of month as a 2-digit number (in the range 01-31).
D	The day of week as a 1-digit number (in the range 1-7, 1 for Sunday and 7 for Saturday).
DAY	The day of week as a 9 character string (in the range 'SUNDAY' to 'SATURDAY '.
DY	The day of week as a 3 character string (in the range 'SUN' to 'SAT').
J	The Julian day (number of days since DEC 31, 1899) as an 8 digit number.
TH	When added to a format keyword that results in a number, this format keyword ('TH') is replaced by the string 'ST', 'ND', 'RD' or 'TH' depending on the last digit of the number.
AMPM	The string 'AM' or 'PM' depending on whether time corresponds to forenoon or afternoon.
A.M.P.M.	The string 'A.M.' or 'P.M.' depending on whether time corresponds to forenoon or afternoon.
HH12	The hour value as a 2-digit number (in the range 00 to 11).
HHHH24	The hour value as a 2-digit number (in the range 00 to 23).
MI	The minute value as a 2-digit number (in the range 00 to 59).
SS	The seconds value as a 2-digit number (in the range 00 to 59).
SSSSS	The seconds from midnight as a 5-digit number (in the range 00000 to 86399).
MLS	The milliseconds value as a 3-digit number (in the range 000 to 999).

Examples

The following examples are based on a table, *orders*, with columns defined as follows:

```
ISQL> table orders
COLNAME                         NULL ?       TYPE        LENGTH
-------                         ------       ----        ------
order_id                        NOT NULL     INT              4
customer_id                                  INT              4
steel_type                                   CHAR            20
```

```
order_info                                      CHAR          200
order_weight                                    INT             4
order_value                                     INT             4
order_state                                     CHAR           20
```

ISQL displays the order_info column, at 200 characters, with lots of blank space preceding the values:

```
ISQL> select order_info from orders where order_value < 1000000
ORDER_INFO
----------




 Solid Rods 5 in. diameter


1 record selected
```

You can improve formatting by using the character format string to limit the width of the display:

```
ISQL> column ORDER_INFO format "A28"  heading  "Details"
ISQL> select order_info from orders where order_value < 1000000;
ORDER_INFO
----------
 Solid Rods 5 in. diameter
1 record selected
ISQL> -- Illustrate some options with numeric format strings.
ISQL> -- No column formatting:
ISQL> select order_value from orders where order_value < 1000000;
                       ORDER_VALUE
                       -----------
                            110000
1 record selected
ISQL> -- Format to display as money, and use different heading:
ISQL>  column order_value format "$999,999,999.99" heading "Amount"
ISQL> select order_value from orders where order_value < 1000000;
             AMOUNT
             ------
             $110,000.00
1 record selected
```

The following examples use the single-value system table, *syscalctable*, and the *sysdate* scalar function, to illustrate some date-time formatting. The *sysdate* function returns today's date.

```
ISQL> select sysdate from syscalctable;  -- No formatting
SYSDATE
-------
05/07/1998
ISQL> column sysdate format "Day"
ISQL> select sysdate from syscalctable
SYSDATE
-------
```

```
    Thursday
1 record selected
ISQL> column sysdate format "Month"
ISQL> select sysdate from syscalctable
SYSDATE
-------
      May
1 record selected
ISQL> column sysdate format "DDth"
ISQL> select sysdate from syscalctable
SYSDATE
-------
           7th
1 record selected
```

Note: If the select-list of a query includes column titles, they override formatting specified in COLUMN statements for those columns. The following example illustrates this behavior.

```
ISQL> select fld from syscalctable; -- No formatting
         FLD
         ---
         100
1 record selected
ISQL> column fld heading "column title"  -- Specify heading in COLUMN statement
ISQL> select fld from syscalctable;
COLUMN TITLE
------------
         100
1 record selected
ISQL> select fld "new title" from syscalctable;  -- Specify title in select list
   NEW TITLE
   ---------
         100
1 record selected
```

3.7.5 COMPUTE

Syntax

```
COMPUTE
     [  { AVG | MAX | MIN | SUM | COUNT }
        OF column_name
        IN variable_name
        ON break_spec  ]  ;
break_spec::
     { column_name | ROW | PAGE | REPORT }
```

Description

Performs aggregate function computations on column values for the specified set of rows, and assigns the results to a variable. DISPLAY statements can then refer to the variable to display its value.

COMPUTE statements have no effect until you issue a BREAK statement with the same *break_spec*.

Issuing the COMPUTE statement without any arguments displays the currently-set COMPUTE specifications, if any.

Arguments

AVG | MAX | MIN | SUM | COUNT
The function to apply to values of *column_name*. The functions AVG, MAX, MIN, and SUM can be used only when the column is numeric. The function COUNT can be used for any column type.

column_name
The column whose value is to be computed. The column specified in *column_name* must also be included in the select list of the query. If *column_name* is not also included in the select list, it has no effect.

variable_name
Specifies the name of the variable where the computed value is stored. ISQL issues an implicit DEFINE statement for *variable_name* and assigns the variable a value of zero. During query processing, the value of *variable_name* changes as ISQL encounters the specified breaks.

break_spec
Specifies the set of rows after which ISQL processes the COMPUTE statement. A COMPUTE statement has no effect until you issue a corresponding BREAK statement. See the description of the BREAK statement in Section 3.7.2 "BREAK" on page 3-13 for details.

Examples

The following example computes the number of items ordered by each customer.

```
ISQL> break on customer_name
ISQL> display col 5 "Number of orders placed by", customer_name, "=", n_ord on
customer_name
ISQL> compute count of order_id in n_ord on customer_name;
ISQL> select c.customer_name, o.order_id from customers c, orders o
where o.customer_id = c.customer_id;
CUSTOMER_NAME                                          ORDER_ID
-------------                                          --------
Sports Cars Inc.                                            1
Sports Cars Inc.                                            2
     Number of orders placed by Sports Cars Inc.
       =        2
Mighty Bulldozer Inc.                                       3
Mighty Bulldozer Inc.                                       4
     Number of orders placed by Mighty Bulldozer Inc.
       =        2
.
.
.
```

3.7.6 DEFINE

Syntax

```
DEFINE [ variable_name = value ] ;
```

Description

The DEFINE statement defines a variable and assigns an ASCII string value to it. When you refer to the defined variable in DISPLAY statements, ISQL prints the value.

The DEFINE statement is useful if you have scripts with many DISPLAY statements. You can change a single DEFINE statement to change the value in all of the DISPLAY statements that refer to the variable.

Issuing the DEFINE statement without any arguments displays any currently-defined variables, including those defined through the COMPUTE statement.

Arguments

variable_name
Specifies the name by which the variable can be referred to.

value
The ASCII string that is assigned to the variable. Enclose value in quotes if it contains any non-numeric values.

Example

The following example defines a variable called *nestate* and assigns the value NH to it.

```
ISQL> DEFINE nestate = "NH" ;
```

3.7.7 DISPLAY

Syntax

```
DISPLAY { [ col_position ] display_value } [ , … ] ON break_spec ;
col_position::
     { COL column_number | @ column_name }
display_value::
     { "text string" | variable | column_name }
break_spec::
     { column_name | ROW | PAGE | REPORT }
```

Description

The DISPLAY statement displays the specified text, variable value, and/or column value after the set of rows specified by *break_spec*. DISPLAY statements have no effect until you issue a BREAK statement with the same *break_spec*.

Issuing the DISPLAY statement without any arguments displays the currently-set DISPLAY specifications, if any.

Arguments

col_position
An optional argument that specifies the horizontal positioning of the associated display value. There are two forms for the argument:

COL column_number	Directly specifies the column position of the display value as an integer(1 specifies column 1, 2 specifies column 2, and so on.).
@column_name	Names a column in the select list of the SQL query. ISQL aligns the display value with the specified column.

If the DISPLAY statement omits *col_position,* ISQL positions the display value at column 1.

display_value
The value to display when the associated break occurs:

"text string"	If the display value is a text string, ISQL simply displays the text string.
variable	If the display value is a variable, ISQL displays the value of the variable when the associated break occurs. The variable argument refers to a variable named in a COMPUTE or DEFINE statement that executes before the query. If variable is undefined, ISQL ignores it.
column_name	If the display value is a column name, ISQL displays the value of the column when the associated break occurs. The column specified in *column_name* must also be included in the select list of the query. If *column_name* is not also included in the select list, it has no effect. If a COLUMN statement specifies a format for the same column, the formatting also affects the DISPLAY statement.

break_spec
Specifies the set of rows after which ISQL processes the DISPLAY statement. A DISPLAY statement has no effect until you issue a corresponding BREAK statement. See the description of the BREAK statement in Section 3.7.2 "BREAK" on page 3-13 for details of break specifications.

Examples

The following set of examples compute the number of orders placed by each customer and displays the message Number of orders placed by, followed by the customer name and the count of orders.

```
ISQL> break on customer_name
```

```
ISQL> display col 5 "Number of orders placed by", customer_name, "=", n_ord on
customer_name
ISQL> compute count of order_id in n_ord on customer_name;
ISQL> select c.customer_name, o.order_id from customers c, orders o
where o.customer_id = c.customer_id;
CUSTOMER_NAME                                          ORDER_ID
-------------                                          --------
Sports Cars Inc.                                            1
Sports Cars Inc.                                            2
     Number of orders placed by Sports Cars Inc.
       =       2
Mighty Bulldozer Inc.                                       3
Mighty Bulldozer Inc.                                       4
     Number of orders placed by Mighty Bulldozer Inc.
       =       2
Ship Shapers Inc.                                           5
Ship Shapers Inc.                                           6
Ship Shapers Inc.                                           7
     Number of orders placed by Ship Shapers Inc.
       =       3
Tower Construction Inc.                                     8
Tower Construction Inc.                                     9
Tower Construction Inc.                                    10
     Number of orders placed by Tower Construction Inc.
       =       3
```

If the select-list of a query includes column titles, they override DISPLAY statements that include variable or *column_name* display values for those columns:

```
ISQL> display col 5 "test display. Sum of fld is", tmp on fld;
ISQL> compute sum of fld in tmp on fld;
ISQL> break on fld
ISQL>  select fld from syscalctable;  -- This works:
         FLD
         ---
         100
     test display. Sum of fld is          100
1 record selected
ISQL> select fld "column title" from syscalctable;  -- DISPLAY is disabled:
COLUMN TITLE
------------
         100
1 record selected
```

3.7.8 EDIT

Syntax

```
E[DIT] [stmt_num];
```

Description

The EDIT statement invokes a text editor to edit the specified statement from the statement history buffer. If the statement number is not specified, the last statement in the history buffer is

edited. When you exit the editor, ISQL writes the buffer contents as the last statement in the history buffer.

By default, ISQL invokes the vi editor on UNIX and the MS-DOS editor on NT. You can change the default by setting the EDITOR environment variable:

- On UNIX, set the environment variable at the operating system command level:
  `setenv EDITOR /usr/local/bin/gmacs`
- On NT, set the environment variable in the initialization file DHSQL.INI in the windows directory:
  `EDITOR = c:\msoffice\winword.exe`

Examples

The following example uses the ! (shell) command to show the currently-set value of the EDITOR environment variable in the UNIX environment (it shows that it is set to invoke the GNU emacs editor). Then, the example uses the EDIT command to read in the fifth statement in the history buffer into an editing buffer.

```
ISQL> ! printenv EDITOR
/usr/local/bin/gmacs
ISQL> EDIT 5;
The following example edits the last statement in the history buffer:
ISQL> select * from systable;  -- bad table name!
              *
error(-20005): Table/View/Synonym not found
ISQL> EDIT  -- invoke an editor to correct the error
.
.
.
ISQL> list  –- corrected statement is now the current statement:
select * from systables
ISQL> run  -- run the corrected statement
.
.
.
```

3.7.9 EXIT or QUIT

Syntax

```
EXIT
```

Description

The EXIT statement terminates the ISQL session.

Related Statements

QUIT and EXIT are synonymous. There is no difference in their effect.

3.7.10 GET

Syntax

```
G[ET] filename;
```

Description

The GET statement reads the first SQL statement stored in the specified script file.

Arguments

filename
The name of the script file. ISQL reads the file until it encounters a semicolon (;) statement terminator. It appends the statement to the history buffer as the most-recent statement.

Notes

- Execute the statement read by GET using the RUN statement.
- The GET, START, and @ (execute) statements are similar in that they all read SQL script files. Both GET and START read an SQL script file and append the first statement in it to the history buffer. However, the START statement also executes the script statement and accepts arguments that it substitutes for parameter references in the script statement. The @ (execute) statement, on the other hand, executes all the statements in an SQL script file but does not add any of the statements to the history buffer. The @ statement does not support argument substitution.

Example

Once you refine a query to return the results you need, you can store it in an SQL script file. For example, the file query.sql contains a complex query that joins several tables in a sample database.

Use the GET and RUN statements to read and execute the first statement in query.sql:

```
ISQL> GET query.sql
SELECT customers.customer_name,
         orders.order_info,
         orders.order_state,
         lot_staging.lot_location,
         lot_staging.start_date
    FROM customers,
         orders,
         lots,
         lot_staging
   WHERE ( customers.customer_id = orders.customer_id ) and
         ( lots.lot_id = lot_staging.lot_id ) and
```

```
         ( orders.order_id = lots.order_id ) and
         ( ( customers.customer_name = 'Ship Shapers Inc.' ) AND
         ( lot_staging.start_date is not NULL ) AND
         ( lot_staging.end_date is NULL ) )
ISQL> RUN
SELECT customers.customer_name,
         orders.order_info,
         orders.order_state,
         lot_staging.lot_location,
         lot_staging.start_date
    FROM customers,
         orders,
     lots,
         lot_staging
   WHERE ( customers.customer_id = orders.customer_id ) and
         ( lots.lot_id = lot_staging.lot_id ) and
         ( orders.order_id = lots.order_id ) and
         ( ( customers.customer_name = 'Ship Shapers Inc.' ) AND
         ( lot_staging.start_date is not NULL ) AND
         ( lot_staging.end_date is NULL ) )
```

```
CUSTOMER_NAME                                    ORDER_INFO
-------------                                    ----------


                ORDER_STATE          LOT_LOCATION          START_DATE
                -----------          ------------          ----------

Ship Shapers Inc.                                I Beams Size 10

                Processing            Hot Rolling          12/26/1994

1 record selected
```

3.7.11 HELP

Syntax

```
HE[LP] {COMMANDS|CLAUSES};

HE[LP] ;
```

Description

The HELP statement displays the help information for the specified statement or clause.

Notes

- HELP COMMANDS displays a list of statements for which help text is available.
- HELP CLAUSES display a list of clauses for which help text is available.
- HELP statement with no clauses display the help text for the HELP statement.

Example

The following HELP statement will give a brief description of the SELECT statement.

```
ISQL> HELP SELECT;
```

3.7.12 HISTORY

Syntax

```
HI[STORY];
```

Description

The HISTORY statement lists the statements in the statement history buffer, along with an identifying number.

Notes

- ISQL maintains a list of statements typed by the user in the statement history buffer. The SET HISTORY statement sets the size of the history buffer.
- The statements LIST, EDIT, HISTORY, and RUN are not added to the history buffer.
- Use HISTORY to obtain the statement number for a particular statement in the history buffer that you want to execute. Then, use the RUN statement with the statement number as an argument to execute that statement. Or, use LIST statement with the statement number as an argument to make the statement the current statement, which can then be executed using RUN without an argument.

Example

The following example illustrates usage of the HISTORY statement.

```
ISQL> HISTORY  -- Display statements in the history buffer
      1  start start_ex.sql Ship
      2  SELECT customer_name FROM customers
         WHERE customer_name LIKE 'Ship%'
      3  select tbl from systables where tbltype = 'T'
ISQL> RUN 2  -- Run the query corresponding to statement 2
SELECT customer_name FROM customers
WHERE customer_name LIKE 'Ship%'
CUSTOMER_NAME
-------------
Ship Shapers Inc.
1 record selected
ISQL> HI  -- In addition to executing, statement 2 is now the current statement
      1  start start_ex.sql Ship
      2  SELECT customer_name FROM customers
         WHERE customer_name LIKE 'Ship%'
      3  select tbl from systables where tbltype = 'T'
      4  SELECT customer_name FROM customers
         WHERE customer_name LIKE 'Ship%'
ISQL> LIST 3 – Display statement 3 and copy it to the end of the history list
select tbl from systables where tbltype = 'T'
ISQL> history  -- Statement 3 is now also the current statement
      1  start start_ex.sql Ship
      2  SELECT customer_name FROM customers
         WHERE customer_name LIKE 'Ship%'
      3  select tbl from systables where tbltype = 'T'
      4  SELECT customer_name FROM customers
         WHERE customer_name LIKE 'Ship%'
      5  select tbl from systables where tbltype = 'T'
```

3.7.13 HOST or SH or !

Syntax

```
{ HOST | SH | ! } [host_command];
```

Description

The HOST statement executes a host operating system command without terminating the current ISQL session.

Arguments

HOST | SH | !
Synonyms for directing ISQL to execute an operating system command.

host_command
The operating system command to execute. If *host_command* is not specified, ISQL spawns a subshell from which you can issue multiple operating system commands. Use the exit command to return to the ISQL> prompt.

Example

Consider a file in the local directory named query.sql. It contains a complex query that joins several tables in a sample database. From within ISQL You can display the contents of the file with the ISQL ! (shell) statement:

```
ISQL> -- Check the syntax for the UNIX 'more' command:
ISQL> host more
Usage: more [-dfln] [+linenum | +/pattern] name1 name2 ...
ISQL> -- Use 'more' to display the query.sql script file:
ISQL> ! more query.sql
SELECT customers.customer_name,
         orders.order_info,
         orders.order_state,
         lot_staging.lot_location,
         lot_staging.start_date
    FROM customers,
         orders,
         lots,
         lot_staging
    WHERE( customers.customer_id = orders.customer_id ) and
         ( lots.lot_id = lot_staging.lot_id ) and
         ( orders.order_id = lots.order_id ) and
         ( ( customers.customer_name = 'Ship Shapers Inc.' ) AND
         ( lot_staging.start_date is not NULL ) AND
         ( lot_staging.end_date is NULL ) )    ;
ISQL> -- Spawn a subshell process to issue multiple OS commands:
ISQL> sh
.
.
.
```

3.7.14 LIST

Syntax

```
L[IST] [ stmt_num ];
```

Description

The LIST statement displays the statement with the specified statement number from the statement history buffer and makes it the current statement by adding it to the end of the history list.

If LIST omits *stmt_num*, it displays the last statement in the history buffer.

Example

The following example uses the LIST statement to display the 5th statement in the history buffer (select *customer_name* from customers) and copy it to the end of the history list. It then executes the now-current statement using the RUN statement:

```
ISQL> history
      1  title
      2  title top "fred" skip 5
      3  title
      4  help title
      5  select customer_name from customers
      6  display "Display on page break!"
      7  display "Test page break display" on page
      8  select customer_name from customers
      9  select customer_name from customers
     10  clear title
ISQL> list 5
select customer_name from customers
ISQL> run
select customer_name from customers
CUSTOMER_NAME
-------------
Sports Cars Inc.
Mighty Bulldozer Inc.
Ship Shapers Inc.
Tower Construction Inc.
Chemical Construction Inc.
Aerospace Enterprises Inc.
Medical Enterprises Inc.
Rail Builders Inc.
Luxury Cars Inc.
Office Furniture Inc.
10 records selected
ISQL>
```

3.7.15 QUIT or EXIT

Syntax

```
Q[UIT]
```

Description

The QUIT statement terminates the current ISQL session.

Related Statements

QUIT and EXIT are synonymous. There is no difference in their effect.

3.7.16 RUN

Syntax

```
R[UN] [stmt_num];
```

Description

The RUN statement executes the statement with the specified statement number from the statement history buffer and makes it the current statement by adding it to the end of the history list.

If RUN omits *stmt_num*, it runs the current statement.

Example

The following example runs the fifth statement in the history buffer.

```
ISQL> HISTORY
      1  title
      2  title top "TEST TITLE" skip 5
      3  title
      4  help title
      5  select customer_name from customers
      6  display "Display on page break!"
      7  display "Test page break display" on page
ISQL> RUN 5
select customer_name from customers
CUSTOMER_NAME
-------------
Sports Cars Inc.
Mighty Bulldozer Inc.
Ship Shapers Inc.
Tower Construction Inc.
Chemical Construction Inc.
Aerospace Enterprises Inc.
Medical Enterprises Inc.
Rail Builders Inc.
Luxury Cars Inc.
```

```
Office Furniture Inc.
10 records selected
ISQL>
```

3.7.17 SAVE

Syntax

```
S[AVE] filename;
```

Description

The SAVE statement saves the last statement in the history buffer in filename. The GET and START statements can then be used to read and execute the statement from a file.

If filename does not exist, ISQL creates it. If filename does exist, ISQL overwrites it with the contents of the last statement in the history buffer.

Example

```
ISQL> ! more test.SQL
test.sql: No such file or directory
ISQL> select customer_name, customer_city from customers;
CUSTOMER_NAME                          CUSTOMER_CITY
-------------                          -------------

Sports Cars Inc.                       Sewickley
Mighty Bulldozer Inc.                  Baldwin Park
Ship Shapers Inc.                      South Miami
Tower Construction Inc.                Munising
Chemical Construction Inc.             Joplin
Aerospace Enterprises Inc.             Scottsdale
Medical Enterprises Inc.               Denver
Rail Builders Inc.                     Claymont
.
.
.
ISQL> save test.sql
ISQL> ! ls -al test.sql
-rw-r--r--  1 ADMIN         51 May  1 18:21 test.sql
ISQL> ! more test.sql
select customer_name, customer_city from customers
ISQL>
```

3.7.18 SET

Syntax

```
SET set_option ;
set_option ::
          HISTORY number_statements
     |    PAGESIZE number_lines
     |    LINESIZE number_characters
```

```
|    COMMAND LINES  number_lines
|    REPORT { ON | OFF }
|    ECHO { ON | OFF }
|    PAUSE { ON | OFF }
|    TIME { ON | OFF }
|    DISPLAY COST { ON | OFF }
|    AUTOCOMMIT { ON | OFF }
|    TRANSACTION ISOLATION LEVEL isolation_level
|    CONNECTION { database_name | DEFAULT }
```

Description

The SET statement changes various characteristics of an interactive SQL session.

Arguments

HISTORY
Sets the number of statements that ISQL will store in the history buffer. The default is 1 statement and the maximum is 250 statements.

PAGESIZE number_lines
Sets the number of lines per page. The default is 24 lines. After each *number_lines* lines, ISQL executes any DISPLAY ON PAGE statements in effect and re-displays column headings. The PAGESIZE setting affects both standard output and the file opened through the SPOOL statement.

LINESIZE
Sets the number of characters per line. The default is 80 characters. The LINESIZE setting affects both standard output and the file opened through the SPOOL statement.

COMMAND LINES
Sets the number of lines to be displayed. The default is 1.

REPORT ON | OFF
SET REPORT ON copies only the results of SQL statements to the file opened by the SPOOL filename ON statement. SET REPORT OFF copies both the SQL statement and the results to the file. SET REPORT OFF is the default.

ECHO ON | OFF
SET ECHO ON displays SQL statements as well as results to standard output. SET ECHO OFF suppresses the display of SQL statements, so that only results are displayed. SET ECHO ON is the default.

PAUSE ON | OFF
SET PAUSE ON prompts the user after displaying one page of results on the screen. SET PAUSE ON is the default.

TIME ON | OFF
SET TIME ON displays the time taken for executing a database query statement. SET TIME OFF disables the display and is the default.

DISPLAY COST ON | OFF
SET DISPLAY COST ON displays the values the c-treeSQL optimizer uses to calculate the least-costly query strategy for a particular SQL statement.

The UPDATE STATISTICS statement updates the values displayed by SET DISPLAY COST ON. SET DISPLAY COST OFF suppresses the display and is the default.

AUTOCOMMIT ON | OFF
SET AUTOCOMMIT ON commits changes and starts a new transaction immediately after each SQL statement is executed. SET AUTOCOMMIT OFF is the default. SET AUTOCOMMIT OFF requires that you end transactions explicitly with a COMMIT or ROLLBACK WORK statement.

TRANSACTION ISOLATION LEVEL isolation_level
Specifies the isolation level. Isolation levels specify the degree to which one transaction can modify data or database objects being used by another concurrent transaction. The default is 3. See the SET TRANSACTION ISOLATION LEVEL statement in the *c-treeSQL Reference Manual* for more information on isolation levels.

CONNECTION { database_name | DEFAULT}
Sets the active connection to *database_name* or to the default connection. See the description of the CONNECT statement in the *c-treeSQL Reference Manual* for details on connections.

Notes

SET REPORT and SET ECHO are similar:

- SET REPORT affects the SPOOL file only, and ON suppresses statement display
- SET ECHO affects standard output only, and OFF suppresses statement display

Other statements control other characteristics of an interactive SQL session:

- The editor invoked by the EDIT statement is controlled by the value of the environment variable EDITOR.
- The file to which interactive SQL writes output is controlled by the SPOOL filename ON statement.

Examples

```
ISQL> -- Illustrate PAGESIZE
ISQL> DISPLAY "Here's a page break!" ON PAGE
ISQL> SET PAGESIZE 4
ISQL> BREAK ON PAGE;
ISQL> SELECT TBL FROM SYSTABLES;
TBL
---
sys_chk_constrs
Here's a page break!
TBL
---
```

```
sys_chkcol_usage
sys_keycol_usage
Here's a page break!
.
.
.
ISQL> SET DISPLAY COST ON
ISQL> -- Select from the one-record SYSCALCTABLE table:
ISQL> SELECT * FROM SYSCALCTABLE;

 Estimated Cost Values :
 -----------------------
 COST        : 8080
 CARDINALITY : 200
 TREE SIZE   : 3072

         FLD
         ---
         100
```

3.7.19 SHOW

Syntax

```
SHOW [ show_option | SPOOL ] ;
show_option ::
        HISTORY
     |  PAGESIZE
     |  LINESIZE
     |  COMMAND LINES
     |  REPORT
     |  ECHO
     |  PAUSE
     |  TIME
     |  DISPLAY COST
     |  AUTOCOMMIT
     |  TRANSACTION ISOLATION LEVEL
     |  CONNECTION
```

Description

The SHOW statement displays the values of the various settings controlled by corresponding SET and SPOOL statements. If the SHOW statement omits *show_option*, it displays all the ISQL settings currently in effect.

See Section 3.7.18 "SET" on page 3-34, Section 3.7.20 "SPOOL" on page 3-38, and Section 3.7.8 "EDIT" on page 3-25 for details on the settings displayed by the SHOW statement.

Example

```
ISQL> SHOW
                            ISQL  ENVIRONMENT
```

```
                                  ____________________

EDITOR ..................... : vi
HISTORY buffer size ........ : 50   PAUSE ..................... : ON
COMMAND LINES .............. : 10   TIMEing command execution.. : OFF

SPOOLing ................... : ON   LINESIZE .................. : 78
REPORTing Facility ......... : ON   PAGESIZE .................. : 72
Spool File ................. : spool_file

AUTOCOMMIT ................. : OFF  ECHO commands ............. : ON
TRANSACTION ISOLATION LEVEL. : 0 (Snapshot)

                              DATABASE CONNECTIONS
                             ______________________

DATABASE       CONNECTION NAME          IS DEFAULT ?      IS CURRENT ?
--------       ---------------          ------------      -----------
salesdb        conn_1                   No                Yes
```

3.7.20 SPOOL

Syntax

```
SPOOL filename [ON] ;
SPOOL OFF ;
SPOOL OUT ;
```

Description

The SPOOL statement writes output from interactive SQL statements to the specified file.

Arguments

filename ON
Opens the file specified by filename and writes the displayed output into that file. The filename cannot include punctuation marks such as a period (.) or comma (,).

OFF
Closes the file opened by the SPOOL ON statement.

OUT
Closes the file opened by the SPOOL ON statement and prints the file. The SPOOL OUT statement passes the file to the system utility statement pr and the output is piped to *lpr.*

Example

To record the displayed output into the file called STK, enter:

```
ISQL> SPOOL STK ON ;

ISQL> SELECT * FROM customer ;

ISQL> SPOOL OFF ;
```

3.7.21 START

Syntax

```
ST[ART] filename [ argument ] [ ... ] ;
```

Description

The START statement executes the first SQL statement stored in the specified script file.

Arguments

filename
The name of the script file. ISQL reads the file until it encounters a semicolon (;) statement terminator.

argument …
ISQL substitutes the value of argument for parameter references in the script. Parameter references in a script are of the form &n, where n is an integer. ISQL replaces all occurrences of &1 in the script with the first argument value, all occurrences of &2 with the second argument value, and so on. The value of argument must not contain spaces or other special characters.

Notes

- In addition to executing the first statement in the script file, the START statement appends the statement (after any argument substitution) to the history buffer.
- The GET, START, and @ (execute) statements are similar in that they all read SQL script files. Both GET and START read an SQL script file and append the first statement in it to the history buffer. However, the START statement also executes the script statement and accepts arguments that it substitutes for parameter references in the script statement. The @ (execute) statement, on the other hand, executes all the statements in an SQL script file but does not add any of the statements to the history buffer. The @ statement does not support argument substitution.

Example

```
ISQL> -- Nothing in history buffer:
ISQL> history
History queue is empty.
```

```
ISQL> -- Display a script file with the ! shell statement. The script's SQL
ISQL> -- statement uses the LIKE predicate to retrieve customer names
ISQL> -- beginning with the string passed as an argument in a START statement:
ISQL> ! more start_ex.sql
SELECT customer_name FROM customers
WHERE customer_name LIKE '&1%';
ISQL> -- Use the START statement to execute the SQL statement in the script
ISQL> -- start_ex.sql. Supply the value 'Ship' as a substitution argument:
ISQL> START start_ex.sql Ship
CUSTOMER_NAME
-------------
Ship Shapers Inc.
1 record selected
ISQL> -- ISQL puts the script statement, after argument substitution,
ISQL> -- in the history buffer:
ISQL> history
      1  ! more start_ex.sql
      3  START start_ex.sql Ship
      4  SELECT customer_name FROM customers
         WHERE customer_name LIKE 'Ship%'
```

3.7.22 TABLE

Syntax

```
T[ABLE] [ tablename ] ;
```

Description

The TABLE statement with no argument displays a list of all the user tables in the database that are owned by the current user.

With the *tablename* argument, the TABLE statement displays a brief description of the columns in the specified table.

Examples

You can use the TABLE statement to see the structure of system tables. Unless you are logged in as the c-treeSQL database administrator (the user *ADMIN*, by default), you need to qualify the system table name with the administrator user name, as in the following example:

```
ISQL> table ADMIN.systables
COLNAME                         NULL ?       TYPE        LENGTH
-------                         ------       ----        ------
id                              NOT NULL     INT              4
tbl                             NOT NULL     VARCHAR         32
creator                         NOT NULL     VARCHAR         32
owner                           NOT NULL     VARCHAR         32
tbltype                         NOT NULL     VARCHAR          1
tblpctfree                      NOT NULL     INT              4
segid                           NOT NULL     INT              4
has_pcnstrs                     NOT NULL     VARCHAR          1
has_fcnstrs                     NOT NULL     VARCHAR          1
```

```
has_ccnstrs                     NOT NULL     VARCHAR          1
has_ucnstrs                     NOT NULL     VARCHAR          1
tbl_status                      NOT NULL     VARCHAR          1
rssid                           NOT NULL     INT              4
```

The following example uses the *table* command to detail the structure of the tables used in examples throughout this chapter.

```
ISQL> table  - List the sample tables
TABLENAME
---------
customers
lot_staging
lots
orders
quality
samples
ISQL> table customers
COLNAME                         NULL ?       TYPE        LENGTH
-------                         ------       ----        ------
customer_id                     NOT NULL     INT              4
customer_name                                CHAR            50
customer_street                              CHAR           100
customer_city                                CHAR            50
customer_state                               CHAR            10
customer_zip                                 CHAR             5
ISQL> table orders
COLNAME                         NULL ?       TYPE        LENGTH
-------                         ------       ----        ------
order_id                        NOT NULL     INT              4
customer_id                                  INT              4
steel_type                                   CHAR            20
order_info                                   CHAR           200
order_weight                                 INT              4
order_value                                  INT              4
order_state                                  CHAR            20
ISQL> table lots
COLNAME                         NULL ?       TYPE        LENGTH
-------                         ------       ----        ------
lot_id                          NOT NULL     INT              4
order_id                        NOT NULL     INT              4
lot_units                                    INT              4
lot_info                                     CHAR           200
ISQL> table lot_staging
COLNAME                         NULL ?       TYPE        LENGTH
-------                         ------       ----        ------
lot_id                                       INT              4
lot_location                                 CHAR            20
start_date                                   DATE
end_date                                     DATE
issues                                       CHAR           200
ISQL> table quality
COLNAME                         NULL ?       TYPE        LENGTH
-------                         ------       ----        ------
```

```
lot_id                          NOT NULL     INT             4
purity                                       DOUBLE          8
p_deviation                                  DOUBLE          8
strength                                     DOUBLE          8
s_deviation                                  DOUBLE          8
comments                                     CHAR          200
ISQL> table samples
COLNAME                         NULL ?       TYPE        LENGTH
-------                         ------       ----        ------
lot_id                                       INT             4
samples                                      INT             4
comments                                     CHAR          200
ISQL>
```

3.7.23 TITLE

Syntax

```
TITLE  [
       [ TOP | BOTTOM ]
       [ [ LEFT | CENTER | RIGHT | COL n ] " text " ] [ … ]
       [ SKIP n ]
       ] ;
```

Description

The TITLE statement specifies text that ISQL displays either before or after it processes a query. TITLE with no arguments displays the titles currently set, if any.

Arguments

TOP | BOTTOM
Specifies whether the title is to be printed at the top or bottom of the page. The default is TOP.

LEFT | CENTER | RIGHT | COL n
Specifies the horizontal alignment of the title text: LEFT aligns the text to the left of the display; CENTER centers the text; RIGHT aligns the text to the right (with the right-most character in the column specified by the SET LINESIZE statement). COL n displays the text starting at the specified column (specifying COL 0 is the same as LEFT).

The default is LEFT.

" text "
The text to be displayed.

SKIP n
Skips the specified number of lines after a TOP title is printed and before a BOTTOM title is printed. By default, ISQL does not skip any lines.

Examples

The following example shows the effect of specifying a top title without a bottom title, then both a top and bottom title.

```
ISQL> TITLE "fred"
ISQL> select * from syscalctable;
fred
         FLD
         ---
         100
1 record selected
ISQL> TITLE BOTTOM "flintstone"
ISQL> select * from syscalctable;
fred
         FLD
         ---
         100
flintstone
1 record selected
The TITLE statement can specify separate positions for different text in the
same title:
ISQL> CLEAR TITLE
ISQL> TITLE TOP LEFT "Align on the left!" CENTER "Centered text" RIGHT "Right
aligned text!"
ISQL> select * from syscalctable;
Align on the left!              Centered text              Right aligned text!
         FLD
         ---
         100

1 record selected
```

Chapter 4

Data Load Utility: *dbload*

4.1 INTRODUCTION

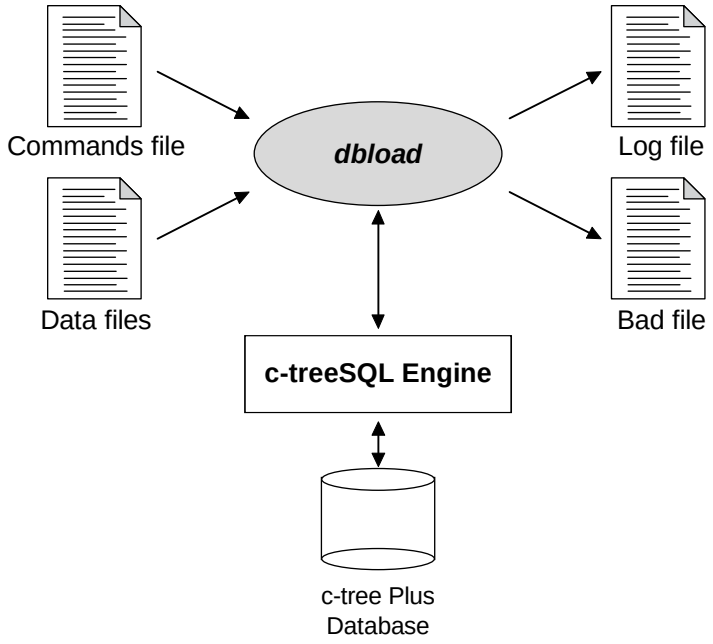
This chapter describes the c-treeSQL database load utility, *dbload*. This utility loads records from an input data file into tables of a database. The format of the data file is specified by a record description given in an input commands file to *dbload*.

Both *dbload* and *dbdump* commands files use DEFINE RECORD statements with similar syntax to specify the format of loaded or exported data records. The commands file specifies the data file, the format of data records, and the destination (or source) database columns and tables for the data.

The *dbload* utility allows loading of variable- or fixed-length records, and lets the load operation specify the set of fields and records to be stored from an input file. Data files can use multiple-character record delimiters. *dbload* also allows control of other characteristics, such as error handling and logging, in its command line. *dbload* generates a *badfile* that contains records from the input file that failed to load in the database.

The following figure shows the *dbload* execution process.

Figure 4-1: *dbload* Execution Process



4.2 PREREQUISITES FOR *DBLOAD*

Before running *dbload*, you need:

- A valid, readable commands file
- INSERT privileges on the tables named in the commands file

4.3 *DBLOAD* COMMAND LINE SYNTAX

The *dbload* command does not directly specify an input file, but instead names a commands file that in turn specifies data input files. The *dbload* command accepts the commands file name, the database name, and a list of command options.

Syntax

```
dbload -f commands_file [options] database_name
```

Options

-f commands_file
Specifies the file containing *dbload* commands.

-l logfile
Specifies the file into which the error logging is done. *stderr* is the default. *dbload* also writes statistics to the file:

- Number of records read
- Number of records skipped
- Number of records loaded
- Number of records rejected

-b badfile
The file into which the bad rows that were not loaded, are written. By default *badfile* is put in the current directory.

-c commit_frequency
Store the specified number of records before committing the transaction. The default frequency is 100 records.

-e maxerrs
The maximum number of tolerable errors. The default number is 50 errors.

-s skipcount
Skip the specified number of rows in the first data file. If multiple files are specified, the rows are skipped only in the first file. The default number is zero rows.

-m maxrows
Stop storing rows at the specified number.

-n
Parse the commands file and display errors, if any, without doing the database load. If the parsing is successful a message, *No errors in the commands file.* displays on *stdout*.

database
Name of the database.

4.4 DATA FILE FORMATS

Data files must be in one of the following record formats:

- Variable length records
- Fixed length records

For both these types of records an optional field delimiter and an optional record delimiter can be specified. The field delimiter, when specified, should be a single character. By default, comma is the field delimiter. The record delimiter can be specified in the commands file and it can be more than one character. By default, the newline character, \n, is the record delimiter.

4.4.1 Variable Length Records

For variable length records, the fields in the data file can be of varying length. Unless the keyword FIXED is used in the commands file, it is assumed that the *dbload* record processing will be for variable length records.

4.4.2 Fixed Length Records

For fixed length records, the fields in the data file must be of fixed length. The length of the record must be the same for all records and is specified in the commands file. In case of fixed length records, the field and record delimiters are ignored. That is, the POSITION specification must be such that the delimiters are ignored. For more information on the commands file refer to Section 4.5 "The Commands File" on page 4-4.

The data files that contain fixed length records can either be ASCII or binary files.

4.5 THE COMMANDS FILE

The commands file specifies instructions for *dbload* to load the records into the table specified. Thus the commands file defines what *dbload* will be performing for a particular loading process.

There is no file naming convention for the commands file. For example, the commands file name to load the *orders* table could be *orders.cmd*.

The commands file must contain the following parts:

- The DEFINE RECORD statement
- The FOR EACH statement

The syntax definition for the commands file is as shown:

```
dbload_commands:
     define_record_statement
     for_each_statement
```

The following is sample commands file showing load instructions.

```
DEFINE RECORD ord_rec AS
    ( ord_no, item_name, date, item_qty ) FIELD DELIMITER ' ' ;

FOR EACH RECORD ord_rec FROM ord_in
    INSERT INTO ADMIN.orders (order_no, product, order_date, qty)
    VALUES (ord_no, item_name, date, item_qty) ;

NEXT RECORD
```

The above commands specification instructs *dbload* to load records into the orders table. The fields in the data file, *ord_in*, appear in the order listed in the DEFINE RECORD statement.

4.5.1 The DEFINE RECORD Statement

The DEFINE RECORD statement is used to define the record that is to be loaded into the database. It describes the data found in the data file. The following are the definitions that are made known by the DEFINE RECORD statement:

- Names the record to be loaded
- Names the fields of the record to be loaded as found in the data file
- Specifies whether the records in the data file are variable length records or fixed length records
- If fixed length records, specifies the position and data type of the field

The following is the syntax definition of the DEFINE RECORD statement:

```
DEFINE RECORD record_name
[ OF FIXED LENGTH record_length
  AS (
       field_name position_specification type_specification, ...
     )
]

AS (
     field_name, ...
   )

[ FIELD DELIMITER delimiter_char ]
[ RECORD DELIMITER delimiter_string ] ;

position_specification::
     POSITION ( start_position : end_position )

type_specification::
       CHAR
     | SHORT
     | LONG
     | FLOAT
     | DOUBLE
```

The following are the variable descriptions of the DEFINE RECORD syntax:

- *record_name* is the name used to refer to the records found in the data file.
- *record_length* is the length of the fixed length record. This length should include the length of field or record delimiters, if any.
- *field_name* is the name used to refer to a field in the data file.
- *delimiter_char* is the field delimiter and is a single character. It must be specified as a literal.
- *delimiter_string* is the record delimiter and can be a single character or a string. It must be specified as a literal.

- *start_position* is the position where the field starts. It must be an unsigned integer.
- *end_position* is the position where the field ends. It must be an unsigned integer.

The first position of each record is 1 and not 0.

If date and time types are to be inserted they can be specified as characters in the data file. If it is a fixed length record then the type specification can be CHAR.

The following is an example of the DEFINE RECORD statement for fixed length records:

```
DEFINE RECORD rec_one OF FIXED LENGTH 20
AS (
     fld1  POSITION (1:4)   SHORT,
     fld2  POSITION (5:15)  CHAR,
     fld3  POSITION (16:20) CHAR
   ) ;
```

4.5.2 The FOR EACH Statement

The FOR EACH statement scans for each valid record in the data file and inserts the record into the database. The syntax for the FOR EACH statement is shown below:

```
FOR EACH RECORD record_name FROM data_file_name, ...
     INSERT INTO owner_name.target_table [ (field_name, ...) ]
     VALUES (value, ...) ;
NEXT RECORD
```

The following are the variable descriptions of the FOR EACH statement:

- *record_name* is the record name that is specified in the DEFINE RECORD statement.
- *data_file_name* is the name of the input data file name.
- *owner_name.target_table* is the target table name identified along with the owner name of the table. The *target_table* must already exist in the database and must have appropriate permissions for inserting the records.
- *field_name* is the name of the field or column in the table.
- *value* is the value that must be inserted into the table.

The *target_table* can also be a synonym on another table with the INSERT access. The list of values that are to be inserted must follow the VALUES keyword. The values that can be inserted are:

- Name of the field in the input data file
- A constant (both numeric as well as character)
- NULL

The values specified in the VALUES list must correspond one to one with that in the target table list. The list can be in any order compared to the list specified in the DEFINE RECORD

statement. The following example shows the list interchanged with respect to the list in the DEFINE RECORD statement.

```
DEFINE RECORD dept_rec AS
     ( dept_no, dept_name, location ) FIELD DELIMITER ' ' ;

FOR EACH RECORD dept_rec FROM dept_in
     INSERT INTO ADMIN.department (loc, no, name)
     VALUES (location, dept_no, dept_name) ;

NEXT RECORD
```

Here the items *no*, *name*, and *loc* are interchanged in both the table list and the values list when compared with the DEFINE RECORD list.

The keyword NEXT RECORD must be specified after the FOR EACH statement so that the insert loop is terminated.

4.6 EXAMPLES

This section gives different types of examples for *dbload*, both for variable length records as well as fixed length records. The data files can either be ASCII or binary files. If they are binary files they must be in the fixed length record format.

The following example is the commands file to load records into the *dept* table. The input data file name is *deptrecs_in* which is an ASCII file in the variable length record format.

Example 4-1: Sample *dbload* commands files

```
DEFINE RECORD dept_rec AS
     ( dept_no, dept_name, location ) FIELD DELIMITER ' ' ;

FOR EACH RECORD dept_rec FROM deptrecs_in
     INSERT INTO ADMIN.dept (no, name, loc)
     VALUES (dept_no, dept_name, location) ;

NEXT RECORD
```

The following is the commands file to load records into the *customer* table. The input data file is *cust_in* which is a binary file in the fixed length record format.

```
DEFINE RECORD cust_rec OF FIXED LENGTH 36
AS (
     cust_no POSITION (1:4) LONG,
     cust_name POSITION (5:15) CHAR,
     cust_street POSITION (16:28) CHAR,
     cust_city POSITION (29:34) CHAR,
     cust_state POSITION (35:36) CHAR
   ) ;

FOR EACH RECORD cust_rec FROM cust_in
     INSERT INTO ADMIN.customer (no, name, city, street, state)
     VALUES (cust_no, cust_name, cust_city, cust_street, 'CA') ;

NEXT RECORD
```

The following is the commands file to load records into the *orders* table. The input data file is *orders_in* which is a binary file in the fixed length record format.

```
DEFINE RECORD orders_rec OF FIXED LENGTH 30
AS (
     order_no POSITION (1:4) LONG,
     order_date POSITION (6:16) CHAR,
     product POSITION (18:25) CHAR,
     qty POSITION (27:30) LONG
   ) ;

FOR EACH RECORD orders_rec FROM orders_in
     INSERT INTO ADMIN.orders (no, date, prod, units)
     VALUES (order_no, order_date, product, qty) ;

NEXT RECORD
```

4.7 *DBLOAD* ERRORS

This section discusses the different types of errors that can occur during the execution of *dbload*.

There are three types of errors that can occur during the *dbload* execution process:

- Commands file errors
- *dbload* errors
- c-treeSQL database errors

The invalid records that are encountered during the processing of records from the data files are flagged as bad records and are written to the *badfile* that is specified in the *dbload* command option. By default, the bad records are written to the file, *badfile*, in the current directory. Any error in the input data file is messaged in the log file (if specified in the command line option) along with the statistics. The following sections discuss the compilation errors and fatal errors that could occur during the *dbload* process execution.

4.7.1 Compilation Errors

The compilation error messages are as follows:

Record name redefined.
The record name in the DEFINE RECORD statement was already defined. The record name must be unique. *dbload* creates a new definition using the same name.

Error in record definition.

Too many fields in record definition.
The number of fields used in the record definition is more than the maximum allowed. Currently, the maximum number allowed is TPE_MAX_FIELDS in the header file *sql_lib.h*.

Position not specified for fixed length record.

Position for SHORT not specified correctly.
The size of the field (start position to end position) must be equal to the size of SHORT.

Position for LONG not specified correctly.
The size of the field (start position to end position) must be equal to the size of LONG.

Position for FLOAT not specified correctly.
The size of the field (start position to end position) must be equal to the size of FLOAT.

Position for DOUBLE not specified correctly.
The size of the field (start position to end position) must be equal to the size of DOUBLE.

Field delimiter must be a single character.

Invalid record delimiter.

Record not defined.
The FOR EACH statement is used with a record name that is not defined.

Mismatch in value list.
The number of values specified in the VALUES list does not match with that specified in the DEFINE RECORD list.

Too many data files specified.
Currently, the maximum number of data files that can be specified in a FOR EACH statement is 10.

Column not found in record definition.

4.7.2 Fatal Errors

The following are a list of nonrecoverable errors.

No memory

Table not found

No columns in the table

Column not found

Too many fields

More than the maximum number of fields allowed, is specified in the table list of the FOR EACH statement.

Cannot open <bad file name>

Cannot open <data file name>

Cannot open log file <log file name>

The *dbload* execution process can also stop if the number of tolerable errors specified (-e option) on the command option is exceeded. By default the number of tolerable errors is 50.

Chapter 5

Data Unload Utility: *dbdump*

5.1 INTRODUCTION

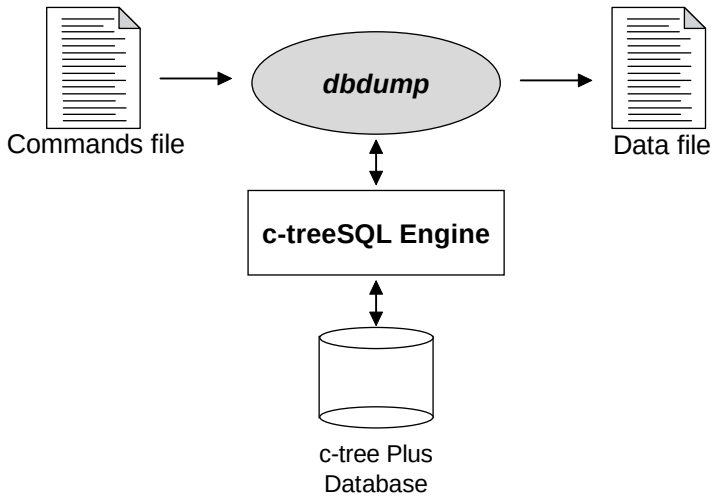
This chapter describes the c-treeSQL database dump utility, *dbdump*.

dbdump writes the data in a database to a file. The format of the exported data is specified by the record description given in an input command file to *dbdump*.

Both *dbload* and *dbdump* commands files use DEFINE RECORD statements with similar syntax to specify the format of loaded or exported data records. The commands file specifies the data file, the format of data records, and the destination (or source) database columns and tables for the data.

The following figure shows the *dbdump* execution process.

Figure 5-1: *dbdump* Execution Process



5.2 PREREQUISITES FOR *DBDUMP*

Before running *dbdump*, you need:

- A valid, readable commands file

- SELECT privileges on the tables named in the commands file

5.3 *DBDUMP* COMMAND LINE SYNTAX

The *dbdump* command accepts the commands file name, the database name and a command option.

Syntax

```
dbdump -f commands_file [-n] database_name
```

Options

-f commands_file
Specifies the file containing *dbdump* commands.

-n
Parse the commands file and display errors, if any, without exporting data. If the parsing is successful a message, *No errors in the commands file.* displays on *stdout*.

database
Name of the database.

5.4 DATA FILE FORMATS

The output data file can be defined to be having one of the following record formats:

- Variable length records
- Fixed length records

For both these types of records an optional field delimiter and an optional record delimiter can be specified. The field delimiter, when specified, should be a single character. By default, comma is the field delimiter. The record delimiter can be specified in the commands file and it can be more than one character. By default, the newline character, \n, is the record delimiter.

5.5 THE COMMANDS FILE

The commands file specifies:

- Record format for the output file
- Query which is to be used for exporting data

There is no file naming convention for the commands file. For example, the commands file name to load the *orders* table could be *orders.cmd*.

The commands file must contain the following parts:

- The DEFINE RECORD statement
- The FOR RECORD statement

The syntax definition for the commands file is as shown:

```
dbdump_commands:
     define_record_statement
     for_record_statement
```

The following is sample commands file showing dump instructions.

```
DEFINE RECORD ord_rec AS
     ( ord_no, item_name, date, item_qty ) FIELD DELIMITER ' ' ;

FOR  RECORD ord_rec dump into ord_dat
USING SELECT order_no, product, order_date, qty
FROM items;
```

5.5.1 The DEFINE RECORD Statement

The DEFINE RECORD statement is used to define the record of the output file. The following are the definitions that are made known by the DEFINE RECORD statement:

- Names the record of the output file
- Names the fields of the record
- Specifies whether the records in the data file are variable length records or fixed length records
- If fixed length records, specifies the position and data type of the field

The following is the syntax definition of the DEFINE RECORD statement:

```
DEFINE RECORD record_name
[ OF FIXED LENGTH record_length
  AS (
       field_name position_specification type_specification,
       ...
     )
]

[ FIELD DELIMITER delimiter_char ]
[ RECORD DELIMITER delimiter_string ] ;

position_specification::
     POSITION ( start_position : end_position )

type_specification::
     | CHAR
     | SHORT
     | LONG
     | FLOAT
     | DOUBLE
```

The following are the variable descriptions of the DEFINE RECORD syntax:

- *record_name* is the name used to refer to the records found in the data file.

- *record_length* is the length of the fixed length record. This length should include the length of field or record delimiters, if any.
- *field_name* is the name used to refer to a field in the data file.
- *delimiter_char* is the field delimiter and is a single character. *delimiter_char* must be specified as a literal.
- *delimiter_string* is the record delimiter and can be a single character or a string. It must be specified as a literal.
- *start_position* is the position where the field starts. It must be an unsigned integer.
- *end_position* is the position where the field ends. It must be an unsigned integer.

The first position of each record is 1 and not 0.

If *date*, *time*, and *timestamp* types are to be dumped they can be specified as characters in the commands file. If it is a fixed length record then the type specification can be CHAR.

The following is an example of the DEFINE RECORD statement for fixed length records:

```
DEFINE RECORD rec_one OF FIXED LENGTH 20
AS (
     fld1  POSITION (1:4)   SHORT,
     fld2  POSITION (5:15)  CHAR,
     fld3  POSITION (16:20) CHAR
   ) ;
```

5.5.2 The FOR RECORD Statement

The FOR RECORD statement writes each valid record into the data file after selecting the record from the database. The syntax for the FOR RECORD statement is shown below:

```
FOR  RECORD record_name DUMP INTO data_file_name
     USING select_statement ;
```

The following are the variable descriptions of the FOR RECORD statement:

- *record_name* specifies the same name used in the associated DEFINE RECORD statement.
- *data_file_name* is the name of the output data file name.
- *select_statement* is any valid SELECT statement.

5.6 EXAMPLES

This section gives different types of examples for *dbdump*, both for variable length records as well as fixed length records. The data files can either be ASCII or binary files. If they are binary files they must be in the fixed length record format.

The following is the commands file to write records from the *dept* table. The output data file name is *deptrecs_out* which is an ASCII file in the variable length record format.

```
DEFINE RECORD dept_rec AS
```

```
     ( no, name, loc ) FIELD DELIMITER ' ' ;

FOR  RECORD dept_rec DUMP INTO deptrecs_out
     USING SELECT dept_no , dept_name , location
FROM ADMIN.dept ;
```

The following is the commands file to write records from the *customer* table. The output data file is *cust_out* which is a binary file in the fixed length record format.

```
DEFINE RECORD cust_rec OF FIXED LENGTH 37
AS (
     no POSITION (1:4) LONG,
     name POSITION (5:15) CHAR,
     street POSITION (16:28) CHAR,
     city POSITION (29:34) CHAR,
     state POSITION (35:36) CHAR
   ) ;

FOR  RECORD cust_rec DUMP INTO cust_out
     USING SELECT cust_no, cust_name, cust_city, cust_street, cust_state
     FROM ADMIN.customer ;
```

The following is the commands file to dump records from the *orders* table. The output data file is *orders_out* which is a binary file in the fixed length record format.

```
DEFINE RECORD orders_rec OF FIXED LENGTH 31
AS (
     no POSITION (1:4) LONG,
     date POSITION (6:16) CHAR,
     prod POSITION (18:25) CHAR,
     units POSITION (27:30) LONG
   ) ;

FOR RECORD orders_rec DUMP INTO orders_out
    USING SELECT order_no, order_date, product, quantity
    FROM ADMIN.orders ;
```

Chapter 6

Schema Export Utility: *dbschema*

6.1 INTRODUCTION

This chapter describes the c-treeSQL utility, *dbschema*. This utility recreates specified database elements and data.

Syntax

```
dbschema [ -h ] [ -d ] [-u user_name ] [-a password ] [ -o outfile ]
         [ -p [ user_name.]procedure_name [ , ... ] ]
         [ -t [ user_name.]table_name [ , ... ] ]
         [ -T [ user_name.]trigger_name [ , ... ] ]
         [ database_name ]
```

Description

Generates SQL statements to recreate the specified database elements and data. If the *dbschema* statement omits all arguments, it displays definitions for all elements (tables, views, indexes, procedures, and triggers) for the default database on the screen.

Options

-h
Displays brief online help of *dbschema* syntax and options.

-d
In conjunction with the -t option, specifies that *dbschema* generates SQL INSERT statements for data in the tables, in addition to CREATE statements. The output of the *dbschema* command invoked with the -d option can be directed to a command file and executed in interactive SQL to duplicate and load table definitions.

-u user_name
The user name c-treeSQL uses to connect to the database. c-treeSQL verifies the user name against a corresponding password before it connects to the database. If omitted, the default value depends on the environment. (On UNIX, the value of the DH_USER environment variable specifies the default user name. If DH_USER is not set, the value of the USER environment variable specifies the default user name.)

-a password
The password c-treeSQL uses to connect to the database. c-treeSQL verifies the password

against a corresponding user name before it connects to the database. If omitted, the default value depends on the environment. (On UNIX, the value of the DH_PASSWD environment variable specifies the default password.)

-o outfile
Redirects the output to the specified file. The default is standard output.

-t [user_name.]table_name [, ...]
A comma-separated list of tables and views for which definitions should be generated. Specify a list of specific tables, or use the % to generate definitions for all tables. (Note that, in the -t option, the % character is not a true wildcard character. It substitutes for the entire *table_name* argument and cannot be used for pattern matching within in a character string. This differs from the behavior of the % in the -p and -T options.)

By default, *dbschema* generates definitions for tables owned by the current user. Use the optional *user_name* qualifier to specify a table owned by a different user.

-p [user_name.]procedure_name [, ...]
A comma-separated list of stored procedures for which definitions should be generated. The table names in the list can include the % and underscore (_) characters, which provide pattern-matching semantics:

- The % matches zero or more characters in the procedure name
- The underscore (_) matches a single character in the procedure name

By default, *dbschema* generates definitions for procedures owned by the current user. Use the optional *user_name* qualifier to specify a procedure owned by a different user.

-T [user_name.]trigger_name [, ...]
A comma-separated list of triggers for which definitions should be generated. The table names in the list can include the % and underscore (_) characters, which provide pattern-matching semantics:

- The % matches zero or more characters in the trigger name
- The underscore (_) character matches a single character in the trigger name

By default, *dbschema* generates definitions for triggers owned by the current user. Use the optional *user_name* qualifier to specify a trigger owned by a different user.

database_name
The database for which *dbschema* should generate definitions. If you omit *database_name*, *dbschema* uses the default database, if specified. (How you define the default database varies between operating systems. On UNIX, the value of the DB_NAME environment variable specifies the default database.)

6.2 EXAMPLES

The following example uses the *-t* option with a table list to generate the table definitions only for the specified table in the *rdsdb* database:

```
ADMIN@isis% dbschema -t dbp1,test_view rdsdb
DBSCHEMA

create table ADMIN.dbp1 (
        c1      integer
) pctfree 20;

create view ADMIN.test_view (
        fld
) as
select * from test_revoke1 ;
```

The following example uses the -p option with the % wildcard character to generate definitions for all stored procedures whose names begin with the characters *foo*:

```
ADMIN@isis% dbschema -p foo% rdsdb
DBSCHEMA
CREATE PROCEDURE ADMIN.foobar(in  sno  character (5),
        in  sname  character (20),
        in  sstatus  smallint,
        in  scity  character (15))
IMPORT

BEGIN
SQLIStatement stmt = new SQLIStatement("insert into s values ('foo', 'foo', 3, '
                                        foo')"); stmt.execute();

END
```

The following example uses the *-o* option to write all definitions for the *rdsdb* database to the file *schema.sql*:

```
dbschema -o schema.sql rdsdb
DBSCHEMA
/vol1/v70_rel_jsp/bin/dhserver <SQL SERVER 29100> -d rdsdb -h 394408 sqlnw_ks
calling DDMJavaCache constructor
Creating loader object
loader object created
Server 29100 done: Fri Jun 19 17:06:35 1998
ADMIN@isis%
ADMIN@isis% more schema.sql

create table ADMIN.test_revoke1 (
        fld     integer
) pctfree 20;
...
```

Appendix A

Tutorial Source Code

A.1 INTRODUCTORY TUTORIAL

```
CREATE TABLE CUSTMAST (
       cm_custnum VARCHAR(5),
       cm_zip VARCHAR(10),
       cm_state VARCHAR(3),
       cm_rating VARCHAR(2),
       cm_name VARCHAR(48),
       cm_address VARCHAR(48),
       cm_city VARCHAR(48)  );
COMMIT WORK;

INSERT INTO CUSTMAST VALUES ('1000', '92867', 'CA', '1', 'Bryan
   Williams', '2999 Regency', 'Orange');
INSERT INTO CUSTMAST VALUES ('1001', '61434', 'CT', '1', 'Michael
   Jordan', '13 Main', 'Harford');
INSERT INTO CUSTMAST VALUES ('1002', '73677', 'GA', '1', 'Joshua
   Brown', '4356 Cambridge', 'Atlanta');
INSERT INTO CUSTMAST VALUES ('1003', '10034', 'MO', '1', 'Keyon
   Dooling', '19771 Park Avenue', 'Columbia');
COMMIT WORK;

SELECT * FROM CUSTMAST;

DELETE FROM CUSTMAST;
COMMIT WORK;

SELECT * FROM CUSTMAST;
```

A.2 RELATIONAL MODEL AND INDEXING TUTORIAL

```
CREATE TABLE orderlist (
   ol_orderdate DATE,
   ol_promdate  DATE,
   ol_ordernum  VARCHAR(7),
   ol_custnum   VARCHAR(4) );

CREATE INDEX custorder ON orderlist (ol_ordernum, ol_custnum);

CREATE TABLE orderitems (
   oi_ordernum VARCHAR(7),
   oi_seqnumber SMALLINT,
   oi_quantity SMALLINT,
```

```
   oi_itemnum  VARCHAR(6) );

CREATE INDEX orderitem ON orderitems (oi_ordernum, oi_seqnumber);

CREATE TABLE itemmast (
   im_weight  INTEGER,
   im_price   MONEY,
   im_itemnum VARCHAR(6),
   im_desc    VARCHAR(48) );

CREATE INDEX itemnum ON itemmast (im_itemnum);

CREATE TABLE custmast (
   cm_custnum VARCHAR(5),
   cm_zip     VARCHAR(10),
   cm_state   VARCHAR(3),
   cm_rating  VARCHAR(2),
   cm_name    VARCHAR(48),
   cm_address VARCHAR(48),
   cm_city    VARCHAR(48));

CREATE INDEX custnum ON custmast (cm_custnum);
COMMIT WORK;

DELETE FROM ORDERLIST;
DELETE FROM ORDERITEMS;
DELETE FROM ITEMMAST;
DELETE FROM CUSTMAST;
COMMIT WORK;

INSERT INTO orderlist VALUES ('9/1/2002', '9/5/2002', '1', '1001');
INSERT INTO orderlist VALUES ('9/2/2002', '9/6/2002', '2', '1002');

INSERT INTO orderitems VALUES ('1',   1,  2,   '1');
INSERT INTO orderitems VALUES ('1',   2,  1,   '2');
INSERT INTO orderitems VALUES ('1',   3,  1,   '3');
INSERT INTO orderitems VALUES ('2',   1,  3,   '3');

INSERT INTO itemmast VALUES (10, 19.95,  '1', 'Hammer');
INSERT INTO itemmast VALUES (3,   9.99,  '2', 'Wrench');
INSERT INTO itemmast VALUES (4,  16.59,  '3', 'Saw');
INSERT INTO itemmast VALUES (1,   3.98,  '4', 'Pliers');

INSERT INTO custmast VALUES ('1000', '92867', 'CA', '1', 'Bryan Williams','2999
   Regency', 'Orange');
INSERT INTO custmast VALUES ('1001', '61434', 'CT', '1', 'Michael Jordan', '13
   Main', 'Harford');
INSERT INTO custmast VALUES ('1002', '73677', 'GA', '1', 'Joshua Brown', '4356
   Cambridge', 'Atlanta');
INSERT INTO custmast VALUES ('1003', '10034', 'MO', '1', 'Keyon Dooling','19771
    Park Avenue', 'Columbia');
COMMIT WORK;

COLUMN cm_name FORMAT "A15"  heading "NAME"
```

```
COLUMN oi_quantity FORMAT "A10" heading "QTY"
COLUMN im_price FORMAT "$99.99" heading "PRICE"
SELECT custmast.cm_name, orderitems.oi_quantity, itemmast.im_price
   FROM custmast, orderitems, itemmast, orderlist
   WHERE orderlist.ol_custnum = custmast.cm_custnum AND
   orderlist.ol_ordernum = orderitems.oi_ordernum AND
   orderitems.oi_itemnum = itemmast.im_itemnum
   ORDER BY orderlist.ol_custnum;
```

A.3 LOCKING TUTORIAL

```
CREATE TABLE CUSTMAST (
   cm_custnum VARCHAR(5),
   cm_zip     VARCHAR(10),
   cm_state   VARCHAR(3),
   cm_rating  VARCHAR(2),
   cm_name    VARCHAR(48),
   cm_address VARCHAR(48),
   cm_city    VARCHAR(48));

COMMIT WORK;

DELETE FROM CUSTMAST;
COMMIT WORK;

INSERT INTO CUSTMAST VALUES ('1000', '92867', 'CA', '1', 'Bryan
   Williams', '2999 Regency', 'Orange');
INSERT INTO CUSTMAST VALUES ('1001', '61434', 'CT', '1', 'Michael
   Jordan', '13 Main', 'Harford');
INSERT INTO CUSTMAST VALUES ('1002', '73677', 'GA', '1', 'Joshua
   Brown', '4356 Cambridge', 'Atlanta');
INSERT INTO CUSTMAST VALUES ('1003', '10034', 'MO', '1', 'Keyon
   Dooling', '19771 Park Avenue', 'Columbia');
COMMIT WORK;

UPDATE custmast SET cm_name = 'KEYON DOOLING'  where cm_custnum = '1003';
```

A.4 TRANSACTION PROCESSING TUTORIAL

```
CREATE TABLE orderlist (
   ol_orderdate DATE,
   ol_promdate  DATE,
   ol_ordernum  VARCHAR(7),
   ol_custnum   VARCHAR(4));

CREATE TABLE orderitems (
   oi_ordernum  VARCHAR(7),
   oi_seqnumber SMALLINT,
   oi_quantity  SMALLINT,
   oi_itemnum   VARCHAR(6));

CREATE TABLE itemmast (
   im_weight  INTEGER,
   im_price   MONEY,
   im_itemnum VARCHAR(6),
```

```
   im_desc    VARCHAR(48));

CREATE TABLE custmast (
   cm_custnum VARCHAR(5),
   cm_zip     VARCHAR(10),
   cm_state   VARCHAR(3),
   cm_rating  VARCHAR(2),
   cm_name    VARCHAR(48),
   cm_address VARCHAR(48),
   cm_city    VARCHAR(48));

COMMIT WORK;

DELETE FROM orderlist;
DELETE FROM orderitems;
DELETE FROM itemmast;
DELETE FROM custmast;
COMMIT WORK;

INSERT INTO itemmast VALUES (10, 19.95,  '1', 'Hammer');
INSERT INTO itemmast VALUES (3, 9.99,  '2', 'Wrench');
INSERT INTO itemmast VALUES (4, 16.59,  '3', 'Saw');
INSERT INTO itemmast VALUES (1, 3.98,  '4', 'Pliers');

INSERT INTO custmast VALUES ('1000', '92867', 'CA', '1', 'Bryan
   Williams', '2999 Regency', 'Orange');
INSERT INTO custmast VALUES ('1001', '61434', 'CT', '1', 'Michael
   Jordan', '13 Main', 'Harford');
INSERT INTO custmast VALUES ('1002', '73677', 'GA', '1', 'Joshua
   Brown', '4356 Cambridge', 'Atlanta');
INSERT INTO custmast VALUES ('1003', '10034', 'MO', '1', 'Keyon
   Dooling', '19771 Park Avenue', 'Columbia');
COMMIT WORK;

INSERT INTO orderitems VALUES ('1', 1,  2, '1');
INSERT INTO orderitems VALUES ('1', 2,  1, '2');
INSERT INTO orderlist VALUES ('9/1/2002', '9/5/2002', '1', '1001');
SELECT orderitems.oi_itemnum, itemmast.im_itemnum
   FROM orderitems, itemmast
   WHERE orderitems.oi_itemnum = itemmast.im_itemnum;
SELECT orderlist.ol_custnum, custmast.cm_custnum
   FROM orderlist, custmast
   WHERE orderlist.ol_custnum = custmast.cm_custnum;
COMMIT WORK;

INSERT INTO orderitems VALUES ('2', 1,  1, '3');
INSERT INTO orderitems VALUES ('2', 2,  3, '4');
INSERT INTO orderlist VALUES ('9/2/2002', '9/6/2002', '2', '9999');
SELECT orderitems.oi_itemnum, itemmast.im_itemnum
   FROM orderitems, itemmast
   WHERE orderitems.oi_itemnum = itemmast.im_itemnum;
SELECT orderlist.ol_custnum, custmast.cm_custnum
   FROM orderlist, custmast
   WHERE orderlist.ol_custnum = custmast.cm_custnum;
```

```
ROLLBACK WORK;

INSERT INTO orderitems VALUES ('3', 1,  2, '3');
INSERT INTO orderitems VALUES ('3', 2,  3, '99');
INSERT INTO orderlist VALUES ('9/22/2002', '9/26/2002', '3', '1002');
SELECT orderitems.oi_itemnum, itemmast.im_itemnum
   FROM orderitems, itemmast
   WHERE orderitems.oi_itemnum = itemmast.im_itemnum;
SELECT orderlist.ol_custnum, custmast.cm_custnum
   FROM orderlist, custmast
   WHERE orderlist.ol_custnum = custmast.cm_custnum;
ROLLBACK WORK;

SELECT * FROM orderlist;
SELECT * FROM orderitems;
```

Symbols

@Execute syntax 3-12

A

Adding titles 3-9

B

Beginning titles 3-9
BREAK statement 3-4
BREAK statement syntax 3-13

C

CLEAR statement 3-4
CLEAR statement syntax 3-15
Column display formatting 3-6
COLUMN statement 3-4
COLUMN statement date-time formats 3-18
COLUMN statement numeric formats 3-18
COLUMN statement syntax 3-16
Commands file 4-4, 5-2
COMPUTE statement 3-4
COMPUTE statement syntax 3-21
Concluding titles 3-9

D

Data file formats for dbdump 5-2
Data file formats for dbload 4-3
Data summaries 3-7
Date-time formats for COLUMN statement 3-18
dbdump
    commands file 5-2
    data file formats 5-2
    DEFINE RECORDS statement 5-3
    examples 5-4
    execution process diagram 5-1
    FOR RECORD statement 5-4
    overview 5-1
    prerequisites 5-1
    syntax 5-2
dbload
    commands file 4-4
    data file formats 4-3
    DEFINE RECORDS statement 4-5
    errors 4-8
    examples 4-7
    execution process diagram 4-1
    fixed length records 4-4
    FOR EACH statement 4-6
    overview 4-1
    prerequisites 4-2
    syntax 4-2
    variable length records 4-4
dbschema
    examples 6-2
    overview 6-1
    syntax 6-1
DEFINE RECORDS statement 4-5, 5-3
DEFINE statement 3-4
DEFINE statement syntax 3-23
DISPLAY statement 3-4
DISPLAY statement syntax 3-23

E

EDIT statement 3-3
EDIT statement syntax 3-25
Errors
    dbload 4-8
EXIT statement syntax 3-26

F

Fixed length records for dbload 4-4
FOR EACH statement 4-6
FOR RECORD statement 5-4
Formatting column displays 3-6

Formatting ISQL output 3-3

G

GET statement 3-3
GET statement syntax 3-27

H

HELP statement syntax 3-29
HISTORY statement 3-2
HISTORY statement syntax 3-29
HOST statement syntax 3-31

I

ISQL
- definition 1-1
- output formats 3-3
- reference 3-12
- starting 3-1
- statements for query formatting 3-4
- syntax 3-1
- usage 1-1

L

LIST statement 3-3
LIST statement syntax 3-32
Load records using dbload 4-1

N

Numeric formats for COLUMN statement .3-18

O

Output formats 3-3

P

Program source code
- sample program source code A-1

Q

Queries, unformatted 3-5
QUIT statement syntax 3-26

R

Recreate database elements and data using dbschema 6-1
References for ISQL 3-12
RUN statement 3-3
RUN statement syntax 3-33

S

Sample application A-1
SAVE statement 3-3
SAVE statement syntax 3-34
SET ECHO statement 3-4
SET LINESIZE statement 3-4
SET PAGESIZE statement 3-4
SET REPORT statement 3-4
SET statement syntax 3-34
SHOW statement syntax 3-37
Source code example A-1
SPOOL statement syntax 3-38
START statement 3-3
START statement syntax 3-39
Starting ISQL 3-1
Statement history support 3-2
- ISQL statements 3-2

Statements
- @EXECUTE syntax 3-12
- BREAK 3-4, 3-7, 3-13
- BREAK syntax 3-13
- CLEAR 3-4, 3-15
- CLEAR syntax 3-15
- COLUMN 3-4, 3-6, 3-16
- COLUMN date-time formats 3-18
- COLUMN numeric formats 3-18
- COLUMN syntax 3-16

COMPUTE 3-4, 3-7, 3-21
COMPUTE syntax 3-21
DEFINE 3-4, 3-23
DEFINE RECORD 4-5, 5-3
DEFINE syntax 3-23
DISPLAY 3-4, 3-7, 3-23
DISPLAY syntax 3-23
EDIT 3-3, 3-25
EDIT syntax 3-25
EXIT 3-26
EXIT syntax 3-26
FOR EACH 4-6
FOR RECORD 5-4
GET 3-3, 3-27
GET syntax 3-27
HELP 3-11, 3-29
HELP syntax 3-29
HISTORY 3-2, 3-29
HISTORY syntax 3-29
HOST 3-31
HOST syntax 3-31
LIST 3-3, 3-32
LIST syntax 3-32
QUIT 3-26
QUIT syntax 3-26
RUN 3-3, 3-33
RUN syntax 3-33
SAVE 3-3, 3-34
SAVE syntax 3-34
SET 3-34
SET ECHO 3-4
SET LINESZIE 3-4
SET PAGESZIE 3-4
SET REPORT 3-4
SET syntax 3-34
SHOW 3-37
SHOW syntax 3-37
SPOOL 3-38
SPOOL syntax 3-38
START 3-3, 3-39
START syntax 3-39
TABLE 3-11, 3-40
TABLE syntax 3-40
TITLE 3-4, 3-42
TITLE syntax 3-42
TITLES 3-9
Statements for query formatting 3-4
Summarizing data 3-7
Syntax for ISQL 3-1

T

TABLE statement syntax 3-40
TITLE statement 3-4
TITLE statement syntax 3-42
Titles
adding 3-9
beginning 3-9
concluding 3-9
Transaction support 3-11
Tutorial 2-1, 2-4, 2-9, 2-13

U

Unformatted queries 3-5

V

Variable length records for dbload 4-4