InterBase
QLI Guide

Disclaimer

Borland International, Inc. (henceforth, Borland) reserves the right to make changes in specifications and other information contained in this publication without prior notice. The reader should, in all cases, consult Borland to determine whether or not any such changes have been made.

The terms and conditions governing the licensing of InterBase software consist solely of those set forth in the written contracts between Borland and its customers. No representation or other affirmation of fact contained in this publication including, but not limited to, statements regarding capacity, response-time performance, suitability for use, or performance of products described herein shall be deemed to be a warranty by Borland for any purpose, or give rise to any liability by Borland whatsoever.

In no event shall Borland be liable for any incidental, indirect, special, or consequential damages whatsoever (including but not limited to lost profits) arising out of or relating to this publication or the information contained in it, even if Borland has been advised, knew, or should have known of the possibility of such damages.

The software programs described in this document are confidential information and proprietary products of Borland.

Restricted Rights Legend. Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subdivision (b) (3) (ii) of the Rights in Technical Data and Computer Software clause at 52.227-7013.

Corporate Headquarters: Borland International Inc., 100 Borland Way, P. O. Box 660001, Scotts Valley, CA 95067-0001, (408) 438-5300. Offices in: Australia, Belgium, Canada, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Latin America, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Taiwan, and United Kingdom.

Software Version: V3.0

Current Printing: October 1993
Documentation Version: v3.0.1

Reprint note

This documentation is a reprint of InterBase V3.0 documentation. It contains most of the information from *InterBase Previous Versions Documentation Corrections* and *InterBase Version* 3.2 *Documentation Corrections* and a new index. For information on features added since InterBase Version V3.0, consult the appropriate release notes.

Table Of Contents

Preface

Qli stands for **q**uery **l**anguage **i**nterpreter. It is InterBase's interactive application environment and report writer. It supports subsets of SQL and GDML data manipulation languages.

Who Should Read this Book

Anyone who wants to retrieve, store or modify data interactively using the Query Language Interpreter should read this manual. **Qli** is also useful for quickly prototyping an application without programming. You can define most of a database's metadata interactively using either interactive GDML or SQL.

This book assumes that you have read the *Getting Started with InterBase* provided with your documentation set.

Using this Book

This manual contains the following chapters:

Chapter 1	Provides a general introduction to **qli**, including how to invoke **qli** and how to access the sample database.
Chapter 2	Describes the relational data model and introduces some of the major concepts used to query a database.
Chapter 3	Describes creating record selection expressions using the GDML variant of **qli.**
Chapter 4	Describes creating record selection expressions using the SQL variant of **qli**.
Chapter 5	Describes how to store and modify data in the database.
Chapter 6	Describes how to create metadata definitions using GDML and SQL.
Chapter 7	Describes how to define and use procedures.
Chapter 8	Introduces the **qli** report writer. You can use the report writer to format query output.
Chapter 9	Introduces the forms manipulation capabilities of **qli**. Forms can be used in **qli** to display and accept data.
Chapter 10	Describes **qli** transactions, used to control what is written to and read from the database.
Chapter 11	Discusses how to convert **qli** statements into GDML and SQL statements for inclusion in programs.
Appendix A	Includes a chart pointing out differences between support for GDML and SQL in **qli**.
Appendix B	Includes the data definitions for atlas.gdb, the sample database included with InterBase.

Text Conventions

This book uses the following text conventions.

boldface

Indicates a command, option, statement, or utility. For example:

- Use the **commit** command to save your changes.
- Use the **sort** option to specify record return order.
- The **case_menu** statement displays a menu in the FORMS window.
- Use **gdef** to extract a data definition.

italic

Indicates chapter and manual titles; identifies filenames and pathnames. Also used for emphasis, or to introduce new terms. For example:

- See the introduction to SQL in the *Programmer's* Guide.
- /usr/interbase/lock_header
- Subscripts in RSE references *must* be closed by parentheses and separated by commas.
- C permits only *zero-based* array subscript references.

`fixed width font`

Indicates user-supplied values and example code:

- $run sys$system:iscinstall
- add field population_1950 long

UPPER CASE

Indicates relation names and field names:

- Secure the RDB$SECURITY_CLASSES system relation.
- Define a missing value of *X* for the LATITUDE_COMPASS field.

Syntax Conventions

This book uses the following syntax conventions.

{braces}	Indicates an alternative item: • `option::=` `{`**`vertical`**`|`**`horizontal`**`|`**`transparent`**`}`
[brackets]	Indicates an optional item: • `dbfield-expression[`**`not`**`]`**`missing`**
`fixed width`	Indicates user-supplied values and example code: • `$run sys$system:iscinstall` • `add field population_1950 long`

commalist	Indicates that preceding word can be repeated to create an expression of one or more words, with each word pair separated by one comma and one or more spaces.
	For example,
	`field_def-commalist`
	resolves to:
	`field_def[,field_def[,field_def]...]`

italics	Indicates syntax variable: • **`create_blob`** *`blob-variable`* **`in`** *`dbfield-expression`*
|	Separates items in a list of choices.
⇓	Indicates that parts of a program or statement have been omitted.

InterBase Documentation

The InterBase Version 3.0 documentation set contains the following books:

Getting Started with InterBase (INT0032WW2179A) provides an overview of InterBase components and interfaces.

Database Operations (INT0032WW2178D) describes how to use InterBase utilities to maintain databases.

Data Definition Guide (INT0032WW2178F) describes how to create and modify InterBase databases.

DDL Reference (INT0032WW2178E) describes the function and syntax for each of the data definition language clauses and statements. It also lists the standard error messages for **gdef**.

DSQL Programmer's Guide (INT0032WW2179C) describes how to program with DSQL, a capability for accepting or generating SQL statements at runtime.

Forms Guide (INT0032WW2178A) describes how to create forms using the InterBase forms editor, **fred,** and how to use forms in **qli** and GDML applications.

Programmer's Guide (INT0032WW2178I) describes how to program with GDML, a relational data manipulation language, and SQL, an industry standard language.

Programmer's Reference (INT0032WW2178H) describes the function and syntax for each of the GDML and InterBase supported SQL clauses and statements. It also lists the standard error messages for **gpre**.

Qli Guide (INT0032WW2178C) describes the use of **qli**, the InterBase query language interpreter that allows you to read to and write from the database using interactive GDML or SQL statements.

Qli Reference (INT0032WW2178B) describes the function and syntax for each of the data definition, GDML, and SQL clauses and statements that you can use in **qli.**

Sample Programs (INT0032WW2178G) contains sample programs that show the use of InterBase features.

Master Index (INT0032WW2179B) contains index entries for the entire InterBase Version 3.0 documentation set.

In addition, platform-specific installation instructions are available for all supported platforms.

Chapter 1
Introduction

This chapter provides an introduction to the capabilities of **qli** and characteristics of its data manipulation language.

Overview

Qli is an interactive interface to databases managed by InterBase. You can use **qli** to:

- Display, store, and update data and prototype these operations for inclusion in application programs.
- Define databases, relations, fields, indexes, views, and procedures.
- Display or generate reports of data.
- Display, store, and update data using forms.

Qli has the following capabilities:

- Subsets of the GDML and SQL data manipulation languages.

  You can use either language, or a combination of the two, for full read and write access to InterBase databases. Because **qli** supports a major subset of each language and programming constructs, such as loops and if—then—else, you can prototype most applications interactively. Later, you can easily convert your **qli** statements into SQL or GDML statements for use in programs.

  You can also mix both GDML and SQL statements during your **qli** session and in your programs. For reasons of clarity, none of the examples in this manual mix statements and clauses from the two languages.

- A subset of **gdef**'s data definition capabilities.

  You can define a database, populate it with relations, define indexes for the relation, and then modify fields, relations, and indexes.

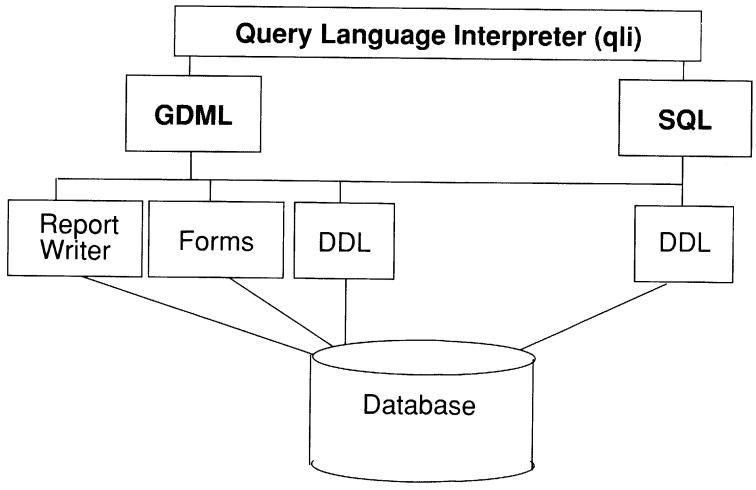
- A report writer. **Qli**'s report writer provides comprehensive formatting options, control breaks, and summaries of control groups.

- Forms. **Qli** supports the automatic and selective use of stored forms for **print**, **store**, and **modify** statements. For those same statements, if you set the **form** option, **qli** automatically generates a form if none exists for the specified relations.

The Qli Environment

The following figure illustrates the **qli** environment.

Figure 1-1. The Qli Environment



The Sample Database

An InterBase database consists primarily of an arbitrary number of relations, each containing an equally arbitrary number of fields.

Interactive examples are used extensively in this manual to illustrate **qli** concepts. The sample database used in the InterBase documentation is based on a North American atlas and gazetteer. The database consists of relations that represent:

- U.S. states (STATES) and Canadian provinces and territories (PROVINCES).
- A sampling of North American cities (CITIES).
- Tourism offices for each of the states and provinces (TOURISM).
- Ski areas (SKI_AREAS).
- State populations (POPULATIONS).
- A selection of North American rivers (RIVERS) and some of the states through which they meander (RIVER_STATES).
- Mayors for certain cities (MAYORS),as of 1985. This relation includes the name, party affiliation, date of next election or date of original appointment for mayors and city managers in approximately one hundred cities.
- The population center for the United States every ten years since 1790 (POPULATION_CENTER).
- Information about some cross country skiing areas in Massachusetts, Maine, and Vermont (CROSS_COUNTRY). This relation contains facts about trails, availability of various amenities, and a comment field that describes the ski area.
- Baseball teams and their stadiums, from both the American and National Leagues (BASEBALL_TEAMS).

The sample database also includes views. Views are subsets of relations or combinations of relations. The *views* include:

- Population density for states (POPULATION_DENSITY), a view that divides the area of a state by its population for each of the last four censuses.
- Geographical data for cities (GEO_CITIES).

Accessing the Sample Database

The atlas.gdb database included with your software is located in the *examples* directory. The exact location of the sample database file depends on the operating system you are using:

- VMS systems. Copy the sample database from *interbase$ivp* to a file in your directory:

```
$ copy  interbase$ivp:atlas.gdb  atlas.gdb
```

- UNIX systems. Copy the sample database from */usr/interbase/examples/atlas.gdb* to a file in your directory:

```
% cp  /usr/interbase/examples/atlas.gdb  atlas.gdb
```

- APOLLO systems. Copy the sample database from */interbase/examples/atlas.gdb* to a file in your directory:

```
% cpf  /interbase/examples/atlas.gdb  atlas.gdb
```

Starting Qli and Readying a Database

Copy the sample database to a local directory as described in the previous section. Once you have copied the database, invoke **qli** and open atlas.gdb for access with the **ready** command:

```
% qli
Welcome to QLI
Query Language Interpreter
QLI> ready atlas.gdb
QLI>
```

If the atlas.gdb database is not in your working directory, use a full pathname when readying a database. For example,:

```
QLI> ready /usr/interbase/examples/atlas.gdb
QLI>
```

Qli Prompts

Qli has two prompts, QLI> and CON>. The QLI> prompt indicates the system is ready for a new command. The CON> prompt indicates a command has not been completed.

The following sections discuss some things you should know about **qli** before you begin using it.

Correcting Errors

If you mistype a command, type the word "edit" and **qli** makes your previously typed command available with the operating system editor. **Qli** uses the default editor on your system. Use the editor's commands to correct the error as you would any other text, exit from the editor, and **qli** executes the corrected command. If you do not want to execute any changes, quit from the editor.

See the section titled *Error Handling* for a description of how **qli** reports errors.

Case Sensitivity

Qli is sensitive to the case of quoted strings in search conditions. For example, the quoted strings "MA" and "ma" represent different values in the following queries:

```
QLI> print cities with state = 'MA'
QLI> print cities with state = 'ma'
```

Given the data in the sample database, the first query returns records, while the second does not. The quotes are important; if you leave them off, **qli** capitalizes the string. For example, an unquoted ma is equivalent to a quoted "MA". The following queries select the same records:

```
QLI> print cities with state = ma
QLI> print cities with state = 'MA'
```

However, if the unquoted string contains blank spaces, non-alphanumeric characters, or a hyphen, **qli** returns an error:

```
QLI> print states with state_name = New York
** QLI error: expected end of statement, encountered "York" **
QLI>
```

The error message tells you **qli** expected the statement to end after the word New and could not process what followed the blank space.

Qli is not sensitive to whether you spell its keywords, relation names, or field names with uppercase or lowercase characters. However, it is sensitive to the case of file specifications if the operating system is case-sensitive:

- If you invoke **qli** on an operating system whose command interpreter or shell is case-sensitive, **qli** distinguishes between uppercase and lowercase characters in file specifications. For example, if the database file is named atlas.gdb, you should type atlas.gdb when you ready it. However, if its name is ATLAS.GDB, type ATLAS.GDB. UNIX operating systems have case-sensitive shells.
- If you invoke **qli** from an operating system whose command interpreter is not case-sensitive, **qli** does *not* distinguish between uppercase and lowercase characters in file specifications. VMS and Apollo have command line interpreters that are not sensitive to case. Apollo Domain/OS is case-sensitive.

Commands and Statements

Qli distinguishes between *commands* and *statements*:

- A command deals with the **qli** environment or its transactions. **Qli** environmental commands are **edit**, **exit**, **abort**, **finish**, **quit**, **ready**, **set**, **shell**, **spawn**, and **show**. Transaction control commands are **commit**, **prepare**, and **rollback**.
- A statement reads, writes, or reports on records. **Qli** statements are *assignment*, **begin/end**, **delete**, **erase**, **for**, **list**, **modify**, **print**, **report**, *restructure*, **select**, **store**, and **update**. Procedures containing statements are also considered to be statements. You can also repeat statements using the **repeat** statement.
- **Qli** also provides the **declare variable** statement for local and global variables, the **copy procedure**, **define procedure**, **edit procedure**, **rename proce-**

dure, and **delete procedure** commands for maintaining procedures, and the **define**, **create**, **modify**, **delete**, and **drop** commands for metadata objects.

This distinction between commands and statements is important in some **qli** contexts, such as in blocks structured with a **begin-end** statement. You cannot include a command in a begin-end block, or in any iterative statement (for example, **for** and **repeat**).

Line Continuation

Qli handles line continuation differently depending upon where you break the line, and whether or not you use the **set semicolon** command.

If you use the **set semicolon** command, **qli** assumes command input is not complete until you type a semicolon. For example:

```
QLI> for states with
CON> state containing
CON> 'M' print state_name, area
CON>
CON>
CON> ;
```

If you finish the command and hit return without typing a semicolon, **qli** returns its continuation prompt.

To turn off the semicolon option, use the **set no semicolon** command. Be sure that you type the semicolon at the end of the command:

```
QLI> set no semicolon;
```

Unless you use the **set semicolon** command, you must break lines in the middle of a clause or at a comma, or use a hyphen. Otherwise, **qli** processes any line that appears to be complete. For example, **qli** recognizes that the following lines are not complete and returns the CON> continuation prompt:

```
QLI> for states with
CON> state containing
CON> 'M' print state_name,
CON>


QLI> for states sorted
CON>


QLI> print state, capital,
CON>
```

The following statement causes **qli** to terminate the command and print the result:

```
QLI> print state, capital, statehood of states
     ⇓
QLI>
```

What this means is that you can type a carriage return into the middle of a query, but **qli** attempts to execute the query fragment. This can invoke an error or cause a query to execute before you intended. For example, if you type the following statement and then press the Return key, **qli** has no idea where to find the fields:

```
QLI> print state, capital
** QLI error: "STATE" is undefined or used out of context **
QLI>
```

In this case, **qli** understands what you want to do only if you supply the name of the relation:

```
QLI> print state, capital of states
     ⇓
QLI>
```

If you are not using the **set semicolon** option, you can indicate to **qli** that a line is incomplete by putting a hyphen at the end of a line. For example, if you want to add a condition to the previous query, you must indicate that the first line is not the entire query, or **qli** processes the **print** statement before you have completed your request. Adding a hyphen after states causes **qli** to return the continuation prompt in anticipation of further input:

```
QLI> print state, capital of states -
CON>
```

Qli Command Options

Qli accepts input interactively from your keyboard, scripts, or command files. Most of this manual discusses input from your keyboard. The following sections discuss alternative input methods.

Startup Files

If you have a set of commands you ordinarily use at the start of every session, you can put them in a file **qli** always uses during its startup. The startup file can include any commands or statements, including **set**, **ready**, and so on. For example, you can ready the same database each time you use **qli**, select various **set** options, and use a **spawn** or **shell** command to execute a "check the date" command outside **qli**.

Place these commands in a startup file. The name of the file **qli** looks for varies by operating system:

- On VMS systems, the startup file has the logical name QLI_STARTUP.
- On UNIX systems, the startup file is *~/.qli_startup*.
- On Apollo systems, the startup file is *~/user_data/qli_startup*.

If you want to use an alternate startup file occasionally, invoke **qli** with the **i** option. The **i** option tells **qli** not to use the standard startup file. The **i** option with no argument suppresses execution of the default command file without executing an alternative. If you follow the **i** option with a filename, **qli** executes the file before any others. If you do not follow the **i** option with a filename, **qli** cancels execution of the standard initialization file. For example:

- On VMS systems:

```
$ qli/i alternate_startup_file
Welcome to QLI
Query Language Interpreter
QLI>
```

- On all other systems:

```
% qli -i alternate_startup_file
Welcome to QLI
Query Language Interpreter
QLI>
```

Setting a Buffer Size

If you are using a very large database (for example, a database with more than 50,000 records) and you have the required memory, you may want to change the **qli** buffer size to enhance performance. The default buffer size depends on the type of server you are using. All servers, with the exception of the central server, use a default buffer size of 75 pages. The default buffer size for the central server is 500 pages. You can increase the buffer size as much as your memory resources allow. To alter the buffer size, use the **b** option with the **qli** command, and specify a new buffer size.

For example, to double the buffer size to 150 pages, enter the following at the system prompt:

- On VMS systems:

```
$ qli/b 150
```

- On all other systems:

```
⇒ qli -b 150
```

For more information on specific servers, refer to the installation guide for your platform.

Scripts and Command Files

If you have other groups of statements or commands that you use frequently, but cannot store as procedures because they **ready** or **finish** the database in which they are stored, or because they are not specific to any one database, you can store them as external scripts or command files. Chapter 7 discusses such command files in the section titled *Command Procedures*.

Non-Interactive Qli

When you have a very short query for which you do not want to start a complete interactive session, you can pass a statement on the **qli** command line. You must put quotation marks around the statement. **Qli** executes its startup command file, if you have one, then the statement from the command line, and exits without entering its interactive mode. For example:

- On VMS systems:

```
$ qli "ready atlas.gdb; print count of states"
 COUNT
=======
     51
$
```

- On all other systems:

```
% qli  "ready atlas.gdb; print count of states"
 COUNT
=======
     51
%
```

You can create a more complicated query or even an entire application that runs without entering **qli**'s normal interactive mode by invoking a script from the command line. For example, consider a script consisting of the following commands:

```
print "Hello world!"
exit
```

To execute this script, place the above commands in a file (in this case *script.com*), use the **n** option to suppress printing the startup banner, and the **a** option to specify the script file:

- On VMS systems:

```
$ qli/n/a script.com
Hello world!
$
```

- On all other systems:

```
% qli -n -a script.com
Hello world!
%
```

The **a** option causes **qli** to invoke the named script or command file after executing the startup file. If the script contains an **exit** command, as the sample script did, **qli** exits without entering interactive mode. Otherwise, it enters interactive mode at the end of the script. You might use this to run an application that prohibits an end-user from accessing **qli** interactively.

Error Reporting

You may encounter errors with any **qli** command or statement. **Qli** reports errors in the following way:

```
**  QLI error <error text>  **
   <additional error text>
```

If you encounter an error and cannot determine the cause or meaning of the error, see the manual page for that statement or command in the *Qli Reference*. Most manual pages in the *Qli Reference* include a section titled *Troubleshooting* that lists possible errors and suggests why the error occurred and how to correct the problem. The error may come from any of three sources:

- A **qli** error. These are generally errors that **qli** encounters when parsing a command, such as an unrecognized word or invalid syntax. The most common are:
  - *Unterminated quoted string.*

    You did not close quotes.
  - Expected <string-1>, encountered <string-2>.

    You mistyped something that **qli** expected would be a keyword. Check the syntax of the command or statement, and the spelling of database names.
  - *<string> is undefined or used out of context.*

    You mistyped something that **qli** expected to be a field or relation name. Check the syntax of the command or statement, and the spelling of database names.
- *No databases are ready.*

  You tried to manipulate data without first readying a database.
- A database error. Database errors can be any one of many problems, such as conversion errors, arithmetic exceptions, and validation errors. You may encounter the following errors:
  - *I/O error during <name> operation for file <database-file>*
  - *Operating system directive failed*
  - *Communication error with journal <journal-name>*

    If you encounter one of these messages, check any secondary messages that are displayed.
- A bugcheck, core dump, or internal error. Bugchecks and core dumps reflect a software problem you should report. If you encounter a bugcheck, do a traceback, if possible, and save the query output and a copy of the database. Contact your system manager and ask to have the problem reported to Interbase Software Corporation.

Examples in this Manual

Many of the chapters in this manual include examples for both the SQL and GDML language variants of **qli**. When both languages are being discussed, an example from each is provided and the output follows. For example, the following queries print the CAPITAL and STATE_NAME fields from each record in the STATES relation:

GDML	`QLI> print capital, state_name of` `CON> states sorted by state_name`
SQL	`QLI> select capital, state_name from` `CON> states order by state_name`

```
                                STATE
      CAPITAL                   NAME
========================= ========================
Montgomery                Alabama
Juneau                    Alaska
Phoenix                   Arizona
Little Rock               Arkansas
Sacramento                California
<display continues for all fifty states>
QLI>
```

In some cases, the SQL and GDML variants are different enough to warrant a more detailed discussion. In those cases, this guide discusses them separately.

Note

When you try examples, the output may not appear exactly as shown in this guide. This is because InterBase sorts record streams arbitrarily unless a sort order is specified.

Changing the Qli Display

As you try examples, you may find that the results are wrapping even when your window is sized generously, creating a confusing display. If this occurs, you can adjust the column width that controls the display. The default column size is 256 character units for Apollo screens, and 80 for all others. To change the column size, use the **set columns** command and specify a new width in character units. For example:

```
QLI> set columns 100
QLI>
```

Using Online Help

Qli provides two online help facilities:

- The help database which consists of statement definitions and topic discussions stored in a database.
- The **show** statement displays information about a database and its entities.

Accessing help

To invoke help, type **help** followed by the name of a command, statement, or a topic for which you want information. For example, suppose you want information on the **commit** command. The **help** facility displays the following:

```
QLI> help commit

COMMIT

The COMMIT command writes to the database changes made during a
transaction.
------------------------------------------------------------
COMMIT [database-handle-comma-list]
------------------------------------------------------------
The following qli script readies a database, stores a record,
thus starting a transaction, and then commits the transaction:

QLI> ready atlas.gdb
QLI> store ski_areas
     ⇓
QLI> commit
QLI>
```

To see a list of all **help** topics, type **help** with no topic specifier, as follows:

```
QLI> help
```

Qli displays a list of all available **help** topics.

Using the Show Command

To use the **show** command, type **show** followed by the name of a database entity. **Show** provides information about the following structures:

- All
- Databases

- Fields
- Forms
- Functions
- Global fields
- Indices (and Indexes)
- Matching_language
- Procedures
- Ready
- Relations
- Security classes
- System relations
- Tables
- Variables
- Version

For example, if you want to see what forms are defined in the atlas.gdb database, type:

```
QLI> show forms
Forms in database QLI_O
  CITY_POPULATIONS
  CITY_POP_LINE
  BASEBALL_TOWNS
  FOO_CITIES
  STATES
  STATE_CITY
```

You can also request information on a specific object rather than a class of objects in the database. For example, to see the definition of the STATES relation, type:

```
QLI> show relation states
STATES
  STATE          varying text, length 4
  STATE_NAME     varying text, length 25
  AREA           long binary
  STATEHOOD      date
  CAPITAL        varying text, length 25
```

Ending a Qli Session

All **qli** operations take place within the context of a *transaction*. A transaction is the basic unit of data retrieval and manipulation. All or none of the statements within the scope of the transaction are executed. If changes made during a transaction cannot be written in their entirety to a database, the database is automatically restored to its state before the transaction started.

A new transaction in **qli** is started automatically when you ready a database, or end a transaction. In **qli**, you have two options for ending the current transaction. Read both of the bulleted items before choosing one:

- Permanently commit your changes to the database. If you stored or modified any records and want to save your changes, end the transaction with the **commit** command:

```
QLI> commit
QLI>
```

- Restore the database to its pre-transaction state. If you made changes to the database, but do not want them to be permanent, end the transaction with the **rollback** command:

```
QLI> rollback
QLI>
```

To close a database, use the **finish** command, as follows:

```
QLI> finish
QLI>
```

The **finish** command with no arguments closes all open databases. If you have more than one database readied, and you want to close only one, specify the database handle along with the **finish** command, as follows:

```
QLI> finish QLI_1
```

Use the **show databases** command to list open databases and their database handles.

To exit from **qli**, use the **exit** command:

```
QLI> exit
⇒
```

The **exit** command automatically commits changes to the database. Be sure to roll back changes you do not want entered before exiting **qli**.

You can also terminate a **qli** session using the **quit** command. If you have made changes in the current transaction and have not committed them or rolled them back, the **quit** command prompts you to specify whether you want the current transaction to

be rolled back before terminating the **qli** session. Respond **y** to rollback the changes, or **n** to commit the changes.

```
QLI> quit
Do you want to rollback your updates? n
⇒
```

Chapter 10 discusses **qli** transactions.

For More Information

For information on transactions, see Chapter 10.

Refer to the *Qli Reference* for syntax for the following commands:

- **ready**
- **show**
- **help**
- **commit**
- **rollback**
- **finish**
- **exit**
- **quit**

Chapter 2
Accessing Data in Qli

This chapter introduces the relational data model and describes how to access relational data.

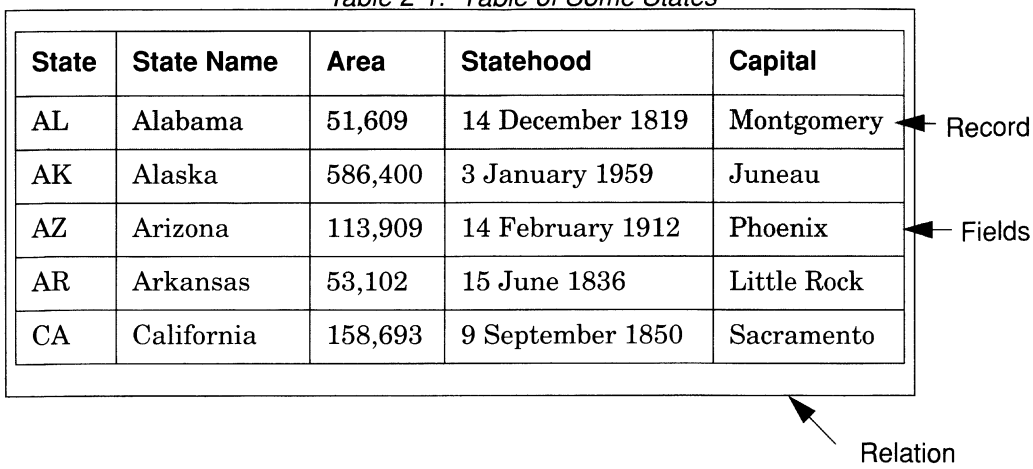
Overview

InterBase stores data using the *relational data model*. This model represents data as two-dimensional tables. A relational table is roughly equivalent to a data file in traditional data processing. Each row in a table represents a record, and each column in a row is a field.

Table 2-1 is a table with five rows, in which each row represents a state in the United States and each column contains different information about each state.

Table 2-1. Table of Some States

State	State Name	Area	Statehood	Capital
AL	Alabama	51,609	14 December 1819	Montgomery
AK	Alaska	586,400	3 January 1959	Juneau
AZ	Arizona	113,909	14 February 1912	Phoenix
AR	Arkansas	53,102	15 June 1836	Little Rock
CA	California	158,693	9 September 1850	Sacramento



Tables are called relations because each row in a table relates a group of field values to each other. For example, the states table or relation relates facts about Alaska; specifically, it became a state in 1959, its capital is Juneau, and its area is 586,400 miles.

Database concepts can be expressed using a variety of industry-accepted terms. From this point on, this manual uses the InterBase terms in the following table.

Table 2-2. Relational Terminology

InterBase	Academic	Traditional
Relation	Relation	File, table
Record	Tuple	Record, row
Field	Attribute	Field or column

Retrieving Data

This section describes retrieving and manipulating the values stored in a database.

Specifying Fields and Values

Field values are the core of relational data retrieval and manipulation because:

- All record selection is performed by comparing a value with the value of a field or fields in a record. For example, suppose you want to find the capital of Arkansas. You might ask for the value of the CAPITAL field for the STATES record with the values "Arkansas" or "AR" for STATE_NAME or STATE, respectively.
- Only the field values you request are returned by InterBase. In the query above, the value "Little Rock" is returned without the AREA, STATEHOOD, STATE_-NAME, and other field values.
- Records are created by storing values in the fields of a relation. For example, if New York City had become a separate state on 4 July 1989, you would store a record that includes "NA", "New Amsterdam", "301", "4 July 1989", and "Manhattan" as the values for the fields STATE, STATE_NAME, AREA, STATEHOOD, and CAPITAL, respectively.

Value Expressions

When you access a database, you read and write fields, and refer to fields by their names. These data values and field names are called *value* or *scalar expressions*. **Qli**'s value expressions let you:

- Reference database fields
- Represent a string of ASCII characters
- Represent a decimal number
- Perform arithmetic operations
- Perform statistical calculations
- Form a record stream and evaluate an expression

The following sections describe using value expressions to form the most common means of data retrieval, the record selection expression.

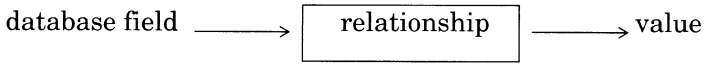
Writing a Record Selection Expression

Usually, you know something about the records you are retrieving. In order to query a database, you often know at least one of the following for a record:

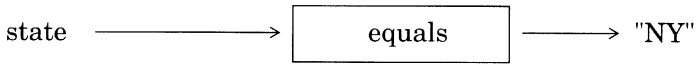
- The contents of a field or fields in that record. For example, you know the name of the state whose capital you want to display.
- A range of values for a field in that record. For example, you may be interested in cities with populations between 150,000 and 200,000.
- That a field value in the record is missing. For example, you suspect that at least one record stored in the STATES relation does not have a value for the CAPITAL city field.

Record Selection Format

In each of these cases, you construct a database query that InterBase evaluates. Many queries have the general form shown below, in which a database field is said to be in some relationship to a supplied value.



Suppose, for example, you want to find all cities in a given state. The query that specifies the state would have the general form shown here:



Record Selection Expressions

The **qli** form of this query is:

GDML	`QLI> print cities with state = "NY"`
SQL	`QLI> select * from cities where state = "NY"`

The records selected by this query are those that represent Albany, New York, and Buffalo.

The *state = "NY"* portion of this query is called a *record selection expression* (RSE). A record selection expression consists of one or more selection criteria. A selection or

search criterion is called a *Boolean expression*, *conditional expression*, or *predicate*. These terms all mean the same thing.

When a Boolean expression is included in a record selection expression, InterBase evaluates the search criteria and comes up with a value of "true," "false," or "missing" for each qualifying record. A statement that evaluates to true, false, or missing is called a *Boolean test*.

The set of records that satisfy a query is called a *record stream* in GDML or *result table* in SQL; the two terms are synonymous. A record stream can result from any of the following record sources:

- A single relation, such as the STATES and CITIES relations
- A view that includes a subset of one or more relations
- A join of two or more relations

The following sections discuss record selection in each of its forms.

Selecting Everything in a Relation

The simplest record stream consists of all fields from all records in a relation. For example, you can type the following command and **qli** displays records for all fifty states:

GDML	`QLI> print states`
SQL	`QLI> select * from states`

To list specific fields, specify field names. The following record selection expression displays only the capital and state names for the STATES relation, sorted according to state names:

GDML	`QLI> print capital, state_name of` `CON> states sorted by state_name`
SQL	`QLI> select capital, state_name from` `CON> states order by state name`

```
                                STATE
        CAPITAL                 NAME
========================= =========================
Montgomery                Alabama
Juneau                    Alaska
Phoenix                   Arizona
```

```
Little Rock                Arkansas
Sacramento                 California
<display continues for all fifty states with SQL>
QLI>
```

The result of a query in which only selected fields are displayed is often called a *vertical subset* of a relation. Figure 2-1 shows the vertical subset resulting from the last query.

Figure 2-1. Vertical Subset of the STATES Relation

State	State Name	Area	Statehood	Capital
AL	Alabama	51,609	14 December 1819	Montgomery
AK	Alaska	586,400	3 January 1959	Juneau
AZ	Arizona	113,909	14 February 1912	Phoenix
AR	Arkansas	53,102	15 June 1836	Little Rock
CA	California	158,693	9 September 1850	Sacramento

Selecting Desired Records from a Relation

Queries that select records from a relation are said to form a *horizontal subset* of records. Figure 2-2 shows an example of a horizontal subset of the STATES relation..

Figure 2-2. A Horizontal Subset of the STATES Relation

State	State Name	Area	Statehood	Capital
AL	Alabama	51,609	14 December 1819	Montgomery
AK	Alaska	586,400	3 January 1959	Juneau
AZ	Arizona	113,909	14 February 1912	Phoenix
AR	Arkansas	53,102	15 June 1836	Little Rock
CA	California	158,693	9 September 1850	Sacramento

To produce the shaded results in Figure 2-2, enter the following query:

GDML	`QLI> print states with state = "AL"`
SQL	`QLI> select * from states where state = "AL"`

You can form a horizontal subset by:

- Matching values exactly
- Matching patterns completely
- Matching patterns partially
- Retrieving all records within an inclusive range
- Choosing only those records with the value for a field missing (or null)
- Combining any

For example, the following queries demonstrate a partial match hat uses the **greater than** operator (**>**) to select records from the RIVERS relation that are longer than 500 miles:

GDML	`QLI> print rivers with length > 500`
SQL	`QLI> select * from rivers where length > 500`

Selecting Unique Values

You can also reduce a record stream to the unique values for a field (or fields). For example, the RIVER_STATES relation contains two fields, a river name and a state. Because many rivers may flow through a state, there are potentially many RIVER_STATES records for a given state:

The following query displays a state name only once if it is a state with a river; without the **distinct** option a state name would recur as many times for each river in the state.

GDML	`QLI> print distinct state of river_states`
SQL	`QLI> select distinct state from river_states`

```
STATE
=====
 AK
 AZ
 CA
 CO
 CT
 IL
 ⇓
QLI>
```

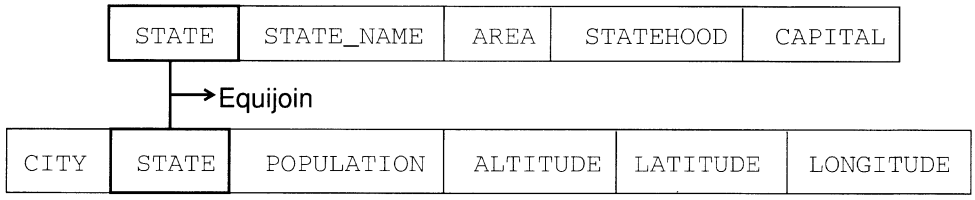
Establishing Relationships Among Relations

The relational *join* operation takes two or more relations and combines them to dynamically form a larger relation. You can join tables based on the equality of common fields (*equijoin*), inequalities (*non-equijoin*), or without regard to relationship (a *cross product*).

Defining an Equijoin

The equijoin is the most common type of join operation. It establishes a relationship between relations by equating the values of fields in each relation. For example, the sample database contains STATES and CITIES, relations that describe U.S. states and North American cities, respectively. To find something about the state in which a city is located, match the state code in the STATES relation with the same field in the CITIES relation. Figure 2-3 illustrates the concept of an equijoin.

Figure 2-3. An Equijoin on the STATES Field



The equijoin is expressed in SQL by naming the relations involved, then setting the join fields equal to each other. The query matches all CITIES and STATES on the basis of identical values for the STATE field. The STATE field is the *join field* or *join term* for this query. The join field is a field in one relation that has values matching values in the join field of another relation. In the following SQL example, the join fields are indicated in bold typeface:

```
QLI> select city, state_name, statehood -
CON>      from cities, states where -
CON>      cities.state = states.state
```

GDML uses a kind of shorthand for expressing the same relationship. Instead of explicitly setting fields equal to each other, GDML provides a **cross** clause that you use to specify the relations being joined, and an **over** clause that you use to specify the join field or fields. The form for the **cross** clause is:

```
<relation1> cross <relation2> over <join field>
```

For example, the SQL example above can be expressed as follows using GDML:

```
QLI> print city, state_name, statehood of
CON> cities cross states over state
```

Both queries return the same output:

```
                    STATE
     CITY                    NAME                 STATEHOOD
==================== ==================== ===========
Juneau               Alaska               3-Jan-1959
Birmingham           Alabama              14-Dec-1819
Montgomery           Alabama              14-Dec-1819
Little Rock          Arkansas             15-Jun-1836
Phoenix              Arizona              14-Feb-1912
  ⇓
QLI>
```

Using Context Variables in a Join Operation

As you add relations to queries, the queries can become difficult to follow. In order to identify a join field (that is, whether you are referring to the STATE field from the STATES relation or the STATE field from the CITIES relation), you can use the relation name itself, or a *context variable* that identifies the source of the record. The SQL equivalent of a context variable is an *alias*. When you associate a context variable or alias with a record stream, you can use the variable to specify the source of a field you reference. Context variables are especially useful when field names from different relations overlap, or you want to join a relation to itself.

Performing joins in SQL requires that you use aliases when the field names of the join term are the same, as in the example above. When you are using GDML, it is good practice to use context variables in joins; they are essential in more complex joins.

Context variables are assigned using the form:

GDML	`context-variable in relation-name`
SQL	`relation-name alias`

The context variable then identifies the source for the field, using the form:

context-variable.field

For example, the following example uses context variables to identify source relations in a query crossing the CITIES relation and the STATES relation:

GDML	QLI> print c.city, s.state_name, c.latitude, CON> c.longitude of c in cities cross s in CON> states over state
SQL	QLI> select c.city, s.state_name, c.latitude, CON> c.longitude from cities c, states s where CON> c.state = s.state

```
                              STATE
      CITY                    NAME                 STATEHOOD
==================== ==================== ===========
Juneau               Alaska               3-Jan-1959
Birmingham           Alabama              14-Dec-1819
Montgomery           Alabama              14-Dec-1819
Little Rock          Arkansas             15-Jun-1836
Phoenix              Arizona              14-Feb-1912
<display continues for remainder of records>
QLI>
```

When you find yourself in an increasingly complex query and have not used context variables, you can qualify fields with their relation names to avoid having to re-do the query. However, it often makes more sense to type some nonsensical word so that **qli** returns an error and then its prompt. At that point, type "edit" at the QLI> prompt. **Qli** then places your incomplete statement in a buffer for your default text editor. You can then add context variables and aliases and otherwise improve the statement in an edit buffer. When you exit from the editor, **qli** executes the improved statement.

Joining More than Two Relations

Joining more than two relations is the same as joining two relations. However, as you increase the number of participating relations, you have to increase the number of join terms. In the atlas database, nearly every relation contains a STATE field, so STATE is the join term in many example joins. The following query returns the names of mayors or city managers who were up for election in 1987 and beyond, displaying fields from the MAYORS, CITIES, and STATES relations:

GDML	QLI> print m.mayor_name, c.city, s.state_name of CON> m in mayors cross c in cities over city, state - CON> cross s in states over state with CON> m.init_term > '31-dec-1986'
SQL	QLI> select m.mayor_name, c.city, s.state_name from CON> mayors m, cities c, states s where - CON> m.city = c.city and c.state = s.state and CON> m.state = c.state and CON> m.init_term > '31-dec-1986'

The preceding query returns information from the MAYORS, CITIES, and STATES relations for which there are matching records. You can specify more restrictive selection criteria to further limit the records that **qli** returns.

The join fields for relations do not have to have the same name. The following example displays the populations of capital cities:

GDML	QLI> print city, population, state_name of CON> cities cross states over state with CON> city = capital
SQL	QLI> select city, population, state_name from CON> cities, states where cities.state = state.state - CON> and cities.city = states.capital

Defining Reflexive Joins

In addition to joining one relation with another, you can join a relation with itself in an operation that is called a *reflexive* or *self-join*. Self-joins are not common, but they can occur. Reflexive joins are frequently used to establish a hierarchy. For example, records in the RIVERS relation contain the name of a river and the body of water into which the river flows. By joining the RIVERS relation with itself, you can find out which rivers flow into other rivers. The following query displays rivers and their tributaries:

GDML	QLI> print r1.river, r2.river of CON> r1 in rivers cross r2 in rivers with CON> r1.river = r2.outflow
SQL	QLI> select r1.river, r2.river from rivers r1, CON> rivers r2 where r1.river = r2.outflow

The context variables or aliases are essential with self-joins because they designate which occurrence of the relation is the source of the referenced fields. If you do not use context variables or aliases with a self-join, **qli** makes its best guess at context, and the results of the query might not be what you expect.

Using Value Expressions in Record Selections

The following sections describe the various value expressions you can use in a record selection expression.

Database Field Expressions

As described earlier in this chapter, the most frequently encountered value expression is the database field value expression, the means by which you refer to a field in a relation.

The following statement includes several database field expressions used in the print list:

GDML	QLI> print city, state, population of cities
SQL	QLI> select city, state, population from cities

```
          CITY             STATE POPULATION
========================== ===== ===========
Albany                      NY
Albuquerque                 NM         331767
Amarillo                    TX
Annapolis                   MD
Atlanta                     GA         425022
  ⇓
QLI>
```

The following example includes a database field expression used as a join term, and several more used in the **print** statement:

GDML	QLI> print city, state_name, latitude, longitude of CON> cities cross states over state
SQL	QLI> select city, state_name, latitude, longitude - CON> from cities c, states s where c.state = s.state

```
                    STATE
   CITY              NAME              LATITUDE LONGITUDE
=========== ======================== ======== =========
Juneau              Alaska             58 18N   134 25W
Birmingham          Alabama            33 31N   086 48W
```

```
Montgomery          Alabama          32 23N    086 19W
Little Rock         Arkansas         34 45N    092 17W
Phoenix             Arizona          33 27N    112 04W
Fresno              California       36 44N    119 47W
Los Angeles         California       34 04N    118 15W
  ⇓
QLI>
```

Quoted String Expressions

Another common value expression is the quoted string, a string of ASCII printing characters, blanks, and tabs enclosed in single (') or double (") quotation marks. ASCII printing characters are:

ASCII Characters...	Printed as...
Uppercase alphabetic	A—*Z*
Lowercase alphabetic	a—z
Numerals	0—9
Special characters	! @ # $ % ^ & * ( ) _ - + = ‘ ~ [ ] { } < > ; : ’ " \ | / ? . ,

The *quoted-string-expression* is most often used either to provide the value used in a value-based retrieval, or for an assignment to a character string or date field. The following query includes a quoted string expression:

GDML	`QLI> print ski_areas with name = "Birchwood Acres"`
SQL	`QLI> select * from ski_areas where name =` `CON> "Birchwood Acres"`

```
  NAME              TYPE              CITY                 STATE
====================  ====  ==========================  =====
Birchwood Acres          N    Groton                        MA
```

Integer and Decimal String Expressions

Qli accepts a string of digits and optional decimal point and interprets it as a decimal number. This is called a *numeric-literal-expression*. It returns an error if the *numeric-literal-expression* exceeds the maximum length of the field as specified by its datatype.

The following query includes a *numeric-literal-expression* used in a value-based retrieval:

GDML	QLI> print city, state, latitude, longitude of CON> cities with altitude < 40
SQL	QLI> select city, state, latitude, longitude from CON> cities where altitude < 40

```
          CITY              STATE LATITUDE LONGITUDE
========================== ===== ======== =========
Boston                      MA    42 20N     071 05W
Brownsville                 TX    25 54N     097 30W
Miami                       FL    25 47N     080 07W
New Orleans                 LA    29 58N     090 04W
New York                    NY    40 40N     073 50W
San Diego                   CA    32 43N     117 09W
⇓
QLI>
```

In numeric-literal expressions, negative values are expressed with a leading numeric sign, as in "-71". However, **qli** does not accept leading plus signs for positive values.

Arithmetic Expressions

Qli supports addition, subtraction, multiplication, division, and concatenation of value expressions in retrievals and assignments. It evaluates the operators (+ - * / |) used in the *arithmetic-expression* in the order listed, but lets you use parentheses to change the order of evaluation.

The following query calculates and displays the altitude in meters for all cities in New York:

GDML	QLI> print city, altitude * .308 of CON> cities with state = "NY"
SQL	QLI> select city, altitude * .308 from CON> cities where state = "NY"

```
          CITY                          ALTITUDE
========================== ====================
Albany                                     18.172
Buffalo                                   187.572
New York                                    2.772
QLI>
```

The concatenation operator, | combines field values in record selection expressions. The following query concatenates two field values and a quoted string:

GDML	`QLI> print city | " is in " | state_name of` `CON> cities cross states over state`
SQL	`QLI> select c.city | ' is in ' | s.state_name from` `CON> cities c, states s where s.state = c.state`

```
Juneau is in Alaska
Birmingham is in Alabama
Montgomery is in Alabama
Little Rock is in Arkansas
Phoenix is in Arizona
⇓
QLI>
```

The following query returns the name of cities where the seating capacity of the National League baseball stadium is greater than one-tenth the population of the city:

GDML	`QLI> print city, population, seating of`
	`CON>  baseball_teams cross cities over city, state with`
	`CON>  league = "N" and seating > population/10`

SQL	`QLI> select city, population, seating from`
	`CON> baseball_teams b, cities c, where c.city =`
	`CON> b.city and c.state = b.state and league = N and`
	`CON> seating gt population/10`

```
          CITY             POPULATION    SEATING
========================== ========== ==========
Atlanta                         425022       53046
Pittsburgh                      423938       54429
St. Louis                       453085       50100
QLI>
```

Aggregate Expressions

Qli supports the following statistical operations on a record stream:

- Average value of a field (**avg**)
- Minimum value stored in a field (**min**)
- Maximum value stored in a field (**max**)
- Sum of values stored in a field (**sum**)

Table 2-3. Aggregate Expressions

GDML Operator	SQL Operator	GDML Abbreviation	What the Operator Returns
count	**count**	**count**	Number of records that satisfy the RSE
maximum	**max**	**max**	Largest non-null value for the field
minimum	**min**	**min**	Smallest non-null value for the field
total	**sum**	**total**	Sum of non-null values in the field
average	**avg**	**avg**	Average of non-null values in the field

In each of the statistical operations, **qli** calculates a value based on a value expression for every qualifying record in a stream. For example, the following query returns the average length of all records in the RIVERS relation:

GDML	`QLI> print average length of rivers`
SQL	`QLI> select avg (length) from rivers`

```
   LENGTH
===========
       696
QLI>
```

Qli can also return the number (**count**) of records that satisfy the record selection expression. The following query returns the number of STATES records:

GDML	`QLI> print count of states with state ne "DC"`
SQL	`QLI> select count (*) from states where` `CON> state ne "DC"`

```
COUNT
=======
     50
QLI>
```

You can also include statistical operations in the selection expression. For example, the following query returns the number of states for which the value of the AREA field is less than the average value of that field:

GDML	QLI> print count of states with area < avg area of CON> states
SQL	QLI> select count (state) from states where area < CON> (select avg (area) from states)

```
COUNT
=======
     35
QLI>
```

User-Defined Functions

Qli supports user-defined functions created using **gdef**. To access a user-defined function in **qli**, name the function, and enclose the input arguments in parentheses. The chapter on user-defined functions in the *Data Definition Guide* describes how to define your own functions and make them available for use. For example, the *Data Definition Guide* provides code you can enter to define a function named **upper** that takes a text string as an argument and returns a value in uppercase type. You can use the **upper** function in a GDML statement as follows:

```
QLI> print upper ("Iowa")
IOWA
```

Or:

```
QLI> print c in city of cities cross
CON> s in ski_areas over city, state with
CON> upper(c.city) = upper(s.city)
```

To see what user-defined functions are available in the active database, use the **show functions** command. To see information on a particular function, use the **show function** command and specify the function name.

Storing Data

Qli supports full read-write access of databases, whether they are on your computer or on other computers running InterBase or compatible software in the same network. You can store new records, modify field values in existing records, or erase records.

You can store data in two ways:

- Automatic prompting for values
- Direct assignment of values

Both methods are fully described in Chapter 5, *Writing Data*. The following sections serve to introduce the different methods for storing data.

Storing Data with Automatic Prompting

Automatic prompting is only available with GDML. To store a record, type the **store** statement followed by the relation name. For example, the following stores a record for the SKI_AREAS relation:

```
QLI> store ski_areas
Enter NAME: Posted Land
Enter TYPE: N
Enter CITY: Billerica
Enter STATE: MA
QLI>
```

Modifying Data

You can modify a record by typing **modify**, the field you want to modify, and an optional search condition to select the record to change. For example, the following statement modifies the CITY field of the SKI_AREAS relation so that the city for the ski area named Posted Land is changed to Wilmington:

```
QLI> modify city of ski_areas with name = 'Posted Land'
Enter CITY: Wilmington
QLI>
```

To store more than one record at a time, use the **repeat** statement qualified by the number of records you wish to store. When you store more than one record, **qli** continues to prompt you for values until you have stored the number of records specified by the **repeat** statement. For example, the following statement stores two records for the RIVER_STATES relation:

```
QLI> repeat 2 store river_states
Enter STATE: MA
Enter RIVER: Nashua
Enter STATE: MA
Enter RIVER: Charles
```

Making Direct Assignments

In addition to storing new records using automatic prompting, you can make direct assignments storing and modifying records using either GDML or SQL. The following statements store a new record in the SKI_AREAS relation.

Note

The assignment clauses are nested in a **begin-end** block. The **begin-end** block groups together statements so that they are evaluated as a single statement.

GDML	`QLI> store ski_areas using begin`
	`CON>  name = 'No Trespassing'`
	`CON>  city = 'Concord'`
	`CON>  state = 'MA'`
	`CON>  type = 'N'`
	`CON> end`

SQL	`QLI> insert into ski_areas`
	`CON>  (name, city, state, type) -`
	`CON>  values ('No Trespassing', 'Concord','MA, 'N')`

For More Information

For a discussion of GDML and SQL expressions, see Chapters 3 and 4, respectively.

For information on defining relations and views, refer to Chapter 6.

For information on transactions in **qli**, refer to Chapter 10.

Refer to the *Qli Reference* for syntax for the following statements:

- **store**
- **modify**
- **repeat**

Chapter 3
Accessing Data Using GDML

This chapter describes using interactive GDML to retrieve data from an InterBase database.

Overview

Using interactive GDML in **qli**, you can:

- Read values from and write values to the database
- Update values in the database
- Define, delete, or modify data definitions

The following sections describe ways of selecting, retrieving, sorting, and formatting data through GDML.

Writing Record Selection Expressions

The *record selection expression* or RSE clause specifies the search and delivery conditions for record retrieval. The RSE identifies the records you want to work with and forms a *record stream*, or group of records from a database. With a record selection expression you can:

- Identify the target relation
- Perform a join operation
- Specify search conditions
- Perform a project operation
- Sort the output
- Limit the number of records in a stream

Displaying Relations and Fields Using the Print Statement

The simplest RSE uses the **print** statement to display records that satisfy the query. The **print** statement requires a *record source*, which identifies the relation or relations from which a record stream is created. For example, the following query retrieves all of the records from the CITIES relation:

```
QLI> print cities
```

The standard form for writing a query using the **print** statement is:

```
print field-comma-list of record-selection-expression
```

The *field-comma-list* specifies the fields you want to select from the relation. If no fields are specified, all fields are retrieved. The following query displays the values for the CAPITAL and STATE_NAME fields in the STATES relation:

```
QLI> print capital, state_name of states
```

In the previous examples, the RSE has simply been the name of one relation. Since it is the role of the RSE to specify search and retrievals, the record selection expression can be quite complex. The following example uses the **print** statement to display the results of a query that joins the CITIES relation and the STATES relation over the STATE field. The *cross-clause*, described in detail later in this chapter, is one example of an RSE.

```
QLI> print city, state_name of cities cross states over state
                 STATE
CITY             NAME
==================== ===============
Montgomery       Alabama
```

```
Birmingham          Alabama
Juneau              Alaska
Phoenix             Arkansas
⇓
QLI>
```

Displaying Records and Fields Using the For Loop

The **print** statement, as described above, is the most common means of displaying a record stream in GDML. However, the **print** statement can be incorporated in a **for** loop to achieve the same results. Using a **for** loop to retrieve and display records is advantageous when your queries are more complex, since **for** loops can be nested and **print** statements cannot.

The **for** loop takes the following general form:

for *record-selection-expression* **print** *field-comma-list*

Using a **for** loop, you can express the last query in the following way:

```
QLI> for cities cross states over state print city, state_name
                    STATE
CITY                NAME
==================== ===============
Montgomery          Alabama
Birmingham          Alabama
Juneau              Alaska
Phoenix             Arkansas
⇓
QLI>
```

Qualifying a Relation

The record source, called the *relation-clause*, can take a variety of options to fully identify the source of the record stream. Options on the *relation-clause* are:

- Context variable.

  The context variable is a temporary variable used for name recognition. It is not stored as part of a relation definition. It can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

  Context variables are useful in complex queries involving multiple relations. In most cases, **qli** can automatically identify references when you do not use context variables. However, in some instances, for example if you are joining a relation to

itself, context variables are required to identify the intended source. In addition, context variables are required in the embedded version of GDML, so if you are using **qli** as a prototyping tool for application programs, you should use context variables.

The following queries contain examples of context variables, with the context variables identified by bold typeface:

```
QLI> for s in states
CON> print s.capital, s.state_name

QLI> for c in cities cross s in states over state
CON> print c.city, s.state_name
```

Qli is not sensitive to the case of the context variable. For example, it treats "B" and "b" as the same character. However, in programs where the host language is case-sensitive, context-variables are case-sensitive.

- *Database handle.*

  The database handle identifies the database in which a relation can be found. As with field references, **qli** can usually figure out the source of the relation. However, if you are using more than one database and the same relation name occurs in two or more of the databases, you can have problems. You can resolve the confusion with a database handle. A database handle is only associated with a database for one **qli** session. It is not stored as part of the database definition.

Assigning a Database Handle

Use the **ready** command to associate a database handle with a database. The following query fragment opens two databases, assigns a handle to each, and references identically-named relations in each database. The database handles are identified by bold typeface:

```
QLI> ready apollo:/interbase/mydemo/atlas.gdb as ap_atlas
QLI> ready atlas.gdb as atlas
QLI> print count of atlas.cities

COUNT
=====
137

QLI> print count of ap_atlas.cities

COUNT
=====
```

```
137
QLI>
```

If you have forgotten to assign database handles, you can use **qli**'s default handles. The default handle InterBase assigns is QLI_*n* where *n* is 0 for the first database readied, 1 for the second, and so on. Use the **show databases** command to display the default or assigned handles:

```
QLI> show databases
Database "apollo:/interbase/mydemo/atlas.gdb" readied as
QLI_1
    Page size is 1024 bytes. Current allocation is 140 pages.
   Database description:Database "atlas.gdb" readied as QLI_0
  Page size is 1024 bytes.  Current allocation is 140 pages.
    Database description:

QLI> print count of qli_0.cities
COUNT
=====
137
```

If you have readied more than one database, specified database handles, and you want information from a specific database, you can use the database handle associated with that database with the **show** command:

```
show <db_handle.entity>
```

For example, if you assigned the handle "emp" and want to display the relations from the "emp" database, use this command.

Specifying GDML Search Conditions

Often you want only a subset of the records in a relation. When you can describe the records you want by comparing values in the records to values you specify, InterBase selects and returns only those records you have described.

The following query returns only those records for which it is true that the STATE field contains the value "CA". The search condition is introduced with the **with** qualifier.

```
QLI> print city, latitude, longitude of cities with state = CON>
'CA'


          CITY             LATITUDE LONGITUDE
========================== ======== =========
Sacramento                 38 32N   121 30W
Fresno                     36 44N   119 47W
Los Angeles                34 04N   118 15W
San Diego                  32 43N   117 09W
San Francisco              37 45N   122 27W
QLI>
```

Combining Search Conditions

This search condition results in a value of "true," "false," or "missing" for each record in the CITIES relation. Such an expression is called a *Boolean test* and is expressed by a *Boolean expression*.

If your query has more than one search condition, such as cities with a population greater than 1,000,000 and at an altitude above 200 feet, the truth of the Boolean test depends on the combination of search conditions:

```
QLI> for cities with population > 1000000 and altitude > 200
CON> print city, state, latitude, longitude


          CITY             STATE LATITUDE LONGITUDE
========================== ===== ======== =========
Detroit                     MI    42 23N    083 05W
Chicago                     IL    41 50N    087 45W
QLI>
```

Only those CITIES records for which both conditions are true are returned. **Qli** supports logical **not**, **and**, and **or**, evaluating them in that order unless you put an expression in parentheses to change the order of evaluation.

To compare a value for equality against one of several other values, you can combine conditions with **or** statements. The following query uses an **or** operator to find cities with a population greater than 1,000,000 *or* altitude above 200 feet:

```
QLI> for cities with population > 1000000 or altitude > 200
CON> print city, state, latitude, longitude


          CITY             STATE LATITUDE LONGITUDE
========================== ===== ======== =========
Indianapolis                IN    39 45N    086 10W
Phoenix                     AZ    33 27N    112 04W
Detroit                     MI    42 23N     083 05W
Chicago                     IL    41 50N     087 45W
⇓
QLI>
```

The following statement uses the **or** operator to find cities in New England:

```
QLI> print city, latitude, longitude of cities with state =
CON> CT or state = RI or state = MA or state = NH or
CON> state = VT or state = ME
```

A simpler, shorthand notation lets you compare a value to a list of values separated by commas. The previous query can be expressed as follows:

```
QLI> print city, latitude, longitude of cities with
CON> state = CT, RI, MA, NH, VT, ME
```

Both queries produce the following result:

```
          CITY             LATITUDE LONGITUDE
========================== ======== =========
Hartford                   41 46N     072 40W
Augusta                    44 17N     069 50W
Boston                     42 20N     071 05W
Concord                    43 13N     071 34W
Providence                 41 50N     071 25W
Montpelier                 44 16N     072 34W
Burlington                 44 28N     073 14W
Portland                   43 41N     070 18W
Presque Isle               46 42N     068 01W
⇓
QLI>
```

Using GDML Operators

In addition to the standard equality, inequality, and range operators described above and in Chapter 2, **qli** supports the following GDML operators for specifying search conditions:

- **containing** (or **ct**) to test for the presence of a string in a field. This test is not sensitive to the case of the characters in the string.

```
QLI> print ski_areas with name containing 'acres'

        NAME          TYPE           CITY             STATE
==================== ==== ========================= =====
Birchwood Acres       N   Groton                     MA
QLI>
```

- **starting with** (or **st**) to test for the presence of a string at the beginning of a field value. This test is case-sensitive. If a string is not enclosed in quotes, InterBase upcases the string before seeking a match.

```
QLI> print state_name, statehood, capital of
CON> states with state_name starting with 'New'

           STATE
           NAME             STATEHOOD           CAPITAL
========================== =========== ====================
New Hampshire              21-Jun-1788 Concord
New Jersey                 18-Dec-1787 Trenton
New Mexico                  6-Jan-1912 Santa Fe
New York                   26-Jul-1788 Albany
QLI>
```

  Generally, **starting with** performs better than **containing** because it allows indexed lookups. However, if you are unsure about the position of your substring or its case, you must use **containing**.

- **matching** to test for the presence of a string with wildcards in a value expression. The asterisk (*) wildcard matches a run of zero or more characters, and the question mark (?) matches a single character. **Matching** is case-sensitive.

```
QLI> print city, state, population of
CON> cities with city matching '*ton*'

          CITY            STATE POPULATION
========================= ===== ==========
Washington                 DC        638333
Baton Rouge                LA        346029
```

```
Boston                          MA          562994
Trenton                         NJ
Charleston                      WV
Houston                         TX         1595138
Burlington                      VT
Charleston                      SC
⇓
QLI>
QLI> print capital, state of states with state matching 'N?'


         CAPITAL           STATE
========================== =====
Lincoln                     NE
Carson City                 NV
Concord                     NH
Trenton                     NJ
Santa Fe                    NM
Albany                      NY
Raleigh                     NC
Bismarck                    ND
QLI>
```

- **matching** with the **using** option to test for the presence of a string using a wildcard search pattern you define. The ability to define your own wildcard characters lets you eliminate conflicts between **qli**'s wildcards and those you are used to in you operating environment. The wildcard pattern can be defined using the following characters:

Character	Operation
?	matches any single character
[]	defines a class of character
~	defines a class for exclusion from the search
*	modifies previous definition or class: indicates zero or more occurrences
@	treats the next character as literal

To build a definition of a wildcard, use the following syntax:

```
USING '<character-being-defined>=<definition-characters>'
```

For example, suppose you want the plus sign (+) to represent zero or more occurrences of any character in your matching pattern. You can define the plus sign as a wildcard and use it in an RSE as follows:

```
QLI> print city, state, population of
CON> cities with city matching '+ton+' using '+=?*'

          CITY             STATE POPULATION
========================== ===== ===========
Washington                  DC        638333
Boston                      MA        562994
Trenton                     NJ
Charleston                  WV
Houston                     TX       1595138
Burlington                  VT
Charleston                  SC
 ⇓
QLI>
```

You can make a match case-insensitive by preceding the pattern definition with **-s(**. For example:

```
QLI> print city of cities with city
CON> matching "=+TON+" using -s(+=?*)
 CITY
=========================
Washington
Boston
Trenton
Charleston
Houston
Burlington
 ⇓
QLI>
```

If you commonly use a non-standard matching language you may want to define it as part of your **qli** environment to avoid typing the matching language in every query in which it is used. To add the matching language to your **qli** environment, use the **set matching_language** statement.

For example, the following expression uses the wildcard definition characters to define the same matching behavior as the default provided by InterBase.

```
set matching_language '-s(*=?*,?=?)'
```

For best results, use the **set matching_language** command in a startup file or in a procedure to establish the matching pattern for all matching patterns.

To see what the matching definition language is defined as in the current database, use the **show matching_language** statement. To clear the matching language definition and return to the **qli** default, use the **set no matching_language** statement.

Note

When you have defined a matching language, all **matching** statements use the language you defined unless you explicitly include a **using** *language* clause.

For more information on defining search patterns, see the description of the **matching using** statement in the GDML section of the *Programmer's Reference.*

- **any** to test for the existence of at least one qualifying record in a relation or relations. This expression is true if the record stream specified by the RSE includes at least one record. For example, the following query prints the state names of any state that has in it at least one ski area:

```
QLI> for s in states with any ski_areas over state
CON> print s.state_name

          STATE
          NAME
=========================
Massachusetts
New Hampshire
Vermont
QLI>
```

- **unique** to test for the existence of exactly one qualifying record. This expression is true if the record stream specified by the RSE consists of one and only one record. The following query displays the number of states for which there is only one city stored for that state:

```
QLI> print count of states with unique cities over state

 COUNT
=======
     22
QLI>
```

- **missing** to test for the absence of a value in database field expression. The following query displays STATES records for which the value of the CAPITAL field is missing:

```
QLI> print state_name of states with capital missing
```

```
          STATE
          NAME
========================
District of Columbia
QLI>
```

Understanding Missing Values

A field that has no value assigned is said to have a *missing value*. Any field can have a missing value. Rather than storing a value for the field, InterBase sets a flag indicating the field has no assigned value. Unless you specify otherwise, **qli** prints blank spaces for numbers, characters, and dates, and nothing for blobs. Fields with missing values are listed last in the sort order.

In addition to the sorting order, missing values have the following characteristics:

- If you perform statistical operations involving a field that is missing for a given record, that record is ignored in the computation.
- A missing value never equals another value, even if the other value is also missing.
- If a missing value is compared to another value, even another missing value, the result of the query is always false.
- A field with a missing value cannot participate as a connecting field in an equijoin or non-equijoin because it never tests true for equality or any other relationship.

For example, many of the records in the CITIES relation do not have a value for the POPULATION field. This absence does not mean that no one lives there. Instead, it indicates the atlas from which the census data for the CITIES relation was taken did not have a listing for that city. Cities with no stored population have the missing value flag set for that field. The following query lists cities for which a population value is missing:

```
QLI> print city, state, population of cities with
CON> population missing

          CITY             STATE POPULATION
========================== ===== ===========
Hartford                    CT
Harrisburg                  PA
Lansing                     MI
Raleigh                     NC
```

```
        ⇓
QLI>
```

See the chapter on defining fields in the *Data Definition Guide* for more information about alternate missing values.

Sorting Records

You often want records displayed in a specific order. InterBase normally returns records in random order. To retrieve records in a particular order, you can instruct InterBase to sort the records by one or more fields or *sort keys*. The order of the sort depends on the datatype of the field, as follows:

- Numeric fields sort numerically, from smallest to largest unless you specify a descending order.
- Date fields sort from earliest to latest, unless you specify a descending sort order.
- Alphanumeric fields are sorted by the ASCII collating sequence. This means that a comes before b and that abc comes before abd, but all uppercase letters sort before all lower case letters unless you specify that the sort should be case-insensitive. The descending option is also available for alphanumeric keys.

The following query sorts the CITIES relation by decreasing population:

```
QLI> for cities sorted by desc population
CON> print city, state, population


          CITY             STATE POPULATION
========================== ===== ===========
New York                    NY       7071639
Chicago                     IL       3005072
Los Angeles                 CA       2966850
Philadelphia                PA       1688210
Houston                     TX       1595138
⇓
QLI>
```

The *sort clause* lets you have as many sort keys as you want.

Specifying a Sort Order

Each sort key can specify whether the sorting order of the sort key is ascending (the default order for the first sort key) or descending. If you do not specify whether a particular sort key is ascending or descending, **qli** assumes that you want the order specified for the most recent key. Therefore, if you list several sort keys but only include the keyword **descending** for the first key, all keys are assumed to be in descending order until you specify **ascending** to reverse the order.

Case-insensitive sort

You can also control whether a sort is case sensitive or not. The default behavior is to sort alphanumeric keys by their ASCII collating position, with records starting with capital letters sorted before records starting with lowercase letters. For example, suppose a user storing records for the MAYORS relation entered values of both D and d for the PARTY_AFFILIATION field. Now you want a list of mayors, sorted by party affiliation. The following example shows imaginary results of such a query:

```
QLI> print last_name, party of mayors sorted by
CON> party, last_name

LAST
NAME                 party
==================== ======

Allison              D
Arrington Jr.        D
Koch                 D
Bradley              I
Rogers               I
Erickson             R
Voinovich            R
Abolt                d
Briare               i
Deese
Olson
⇓
```

To specify a sort that ignores case, use the **anycase** option. For example, the following shows the above query with the **anycase** sort option:

```
QLI> print last_name, party of mayors sorted by
CON> anycase party, last_name

LAST
NAME                 party
==================== ======

Abolt                d
Allison              D
Arrington Jr.        D
Koch                 D
Bradley              I
Briare               i
```

```
Rogers              I
Erickson            R
Voinovich           R
```

To explicitly specify the default behavior, specify the **exactcase** option for the sort clause.

Limiting Number of Records Retrieved

If you are not interested in seeing the full record stream for a query, GDML allows you to specify a number of records to retrieve, using the *first-clause* with a numeric argument. The following example displays the information for the first five records of the STATES relation, sorted alphabetically, according to state name:

```
QLI> print first 5 states sorted by state_name

       STATE
STATE  NAME            AREA        STATEHOOD   CAPITAL
===== ================ ========= ============ ==============
 AL   Alabama          51609     14-DEC-1819  Montgomery
 AK   Alaska           586400     3-JAN-1959  Juneau
 AZ   Arizona          113909    14-FEB-1912  Phoenix
 AR   Arkansas         53102     15-JUN-1836  Little Rock
 CA   California       158693     9-SEP-1850  Sacramento
QLI>
```

The **first 5** clause in GDML limits the display to just five records, while the **sorted by** clause tells **qli** to retrieve the records in sorted order. In some cases, there may be multiple sorting fields.

When you use the *first-clause*, you usually want to sort the record stream; otherwise, you retrieve *n* random records, and not necessarily the same records each time.

Although the syntax calls for a numeric value, you can use any value expression that fits. For example, you might ask for the

- First 3-1 states
- First (variable1 - variable2) states
- First *.'a number, any number' states

In all cases, the value expression must evaluate to a number, so first five states is not legal. **Qli** rounds off any fractional portion; therefore, first 4.2 states becomes first 4 states.

Retrieving Unique Values

The RSE's *reduced-clause* retrieves only the unique values for a field from a record stream. When you ask for a unique record stream for a field or fields, InterBase considers the list of fields and eliminates records that do not have a unique combination of values for those listed fields.

For example, the following river returns the number of records in the RIVER_STATES relation that have values in the STATES field:

```
QLI> print count of river_states

COUNT
======
152
QLI>
```

However, if what you want to know is how many states have rivers in them, the above query cannot tell you, since it counts multiple occurrences of the same value in the STATE field. The following query uses the reduced to clause to retrieve only the first occurrence of a value for the STATE field:

```
QLI> print count of river_states reduced to state

COUNT
=====
 50
QLI>
```

You can also query for a unique value using the **distinct** keyword in a **print** statement. For example:

```
QLI> print distinct state of river_states
```

The following query returns unique pairs of rivers and states:

```
QLI> print distinct state, river of river_states
```

Joining Relations

The RSE's *cross-clause* creates dynamic relationships by matching up records from two or more relations in the same database. Preceding sections have already presented several examples of joining relations using the *cross-clause*. For example, the following query joins the STATES and CITIES relations on the STATE field:

```
QLI> for cities cross states over state sorted
CON> by state, city
CON> print city, state_name, latitude, longitude


                              STATE
       CITY                   NAME              LATITUDE LONGITUDE
==================== ====================== ======== =========
Juneau                       Alaska            58 18N    134 25W
Birmingham                   Alabama           33 31N    086 48W
Montgomery                   Alabama           32 23N    086 19W
Little Rock                  Arkansas          34 45N    092 17W
Phoenix                      Arizona           33 27N    112 04W
Fresno                       California        36 44N    119 47W
Los Angeles                  California        34 04N    118 15W
⇓
QLI>
```

Joining More than Two Relations

Unlike most other RSE clauses, the *cross-clause* can be repeated. To include more than two relations, add cross-clauses. The following query joins the CITIES, STATES, and TOURISM relations and displays the city, state name and zip code of the city that contains the tourism office for the state. Because the TOURISM relation contains a record with postal or zip code field for only those cities with state offices of tourism or chambers of commerce, the query picks up the code for a state's tourism office and displays it for every city in that state that has an office for tourism.

```
QLI> for c in cities cross s in states over state cross
CON> t in tourism over state, city
CON> print c.city, s.state_name, t.zip
               STATE
CITY           NAME                 ZIP
================ ============== =====
Juneau         Alaska               99811
Montgomery     Alabama              36130
Little Rock    Arkansas             72201
```

```
Phoenix         Arizona              85004
Sacramento      California           95808
⇓
QLI>
```

Using Nested For Loops to Produce Outer Joins

Occasionally you will want to combine data from two relations, matching values where possible, but not excluding records that have no match. This capability is called an *outer join*. **Qli** does not directly support an outer join, but you can achieve the same results using **for** loops to combine statements. For example, the following statement prints all of the states, and then prints baseball teams for states that have them:

```
QLI> for s in states sorted by s.state
CON> begin
CON>   print s.state_name
CON>   for b in baseball_teams over state sorted by b.team_name
CON>   print b.team_name, b.home_stadium
CON> end

        STATE              TEAM                  HOME
        NAME               NAME                STADIUM
==================    ============ ==========================
Alaska
Alabama
Arkansas
Arizona
California             A's              Oakland Coliseum
                       Angels           Anaheim Stadium
                       Dodgers          Dodger Stadium
                       Giants           Candlestick Park
                       Padres           Jack Murphy Stadium
                       ⇓
QLI>
```

For this query, you could have used a **then** in place of the **begin-end** block to connect the two actions of this **for** loop. For example:

```
QLI> for s in states sorted by s.state
CON> print s.state_name then
CON> for b in baseball_teams over state sorted by b.team_name
CON> print b.team_name, b.home_stadium
```

Joining Relations from Two Databases

To join relations in different databases, InterBase requires the relations to be joined exist in the same database. Thus, if you have a database that contains customer information, and you want to print the population of cities that contain customers, use nested **for** loops to emulate a cross-database join. For example:

```
QLI> ready emp.gdb as emp
QLI> ready atlas.gdb as atlas
QLI> for cust in customers
CON>    for c in cities with city = cust.city and
CON>      c.state = cust.state
CON> print cust.customer, c.city, c.population
```

Other Ways to Combine Data

In addition to the **cross** operation and nested **for** loops, you can combine data from several relations through *aggregate value expressions*. Aggregate value expressions are expressions that operate on a collection of values from some field of a relation. Aggregate expressions include the **any** Boolean test, the **unique** Boolean test, the **first from** value expression. You cannot reference relations from multiple databases through these operations.

First Expression

Qli supports a value expression that searches specified relations for the first qualifying record, and then evaluates the value expression. If there are no qualifying records, you receive the error "no match for first value expression" unless you also supplied an **else** clause. Otherwise, InterBase evaluates the first value expression you provide in the context of the record it found. The result of the evaluation is returned as the value of the *first-expression* or the value specified in the **else** clause.

This expression is particularly useful if you store an abbreviated version of data in one relation and store the expanded version in another. For example, in the atlas database, the field STATE, which identifies states in most relations, contains the two-letter state abbreviation approved by the United States Postal Service. The full state name is kept only in the STATES relation in the STATE_NAME field. One way to get the full state name is through the *first-expression*. For example:

```
QLI> for cities with city = "Portland"
CON> print city, first state_name
CON> from states over state
Portland, Maine
```

```
Portland, Oregon
QLI>
```

Because not all cities have matching states, a more general form of this query should use an **else** clause to avoid errors:

```
QLI> for cities sorted by city
CON> print city, first state_name from
CON> states over state else "*** UNKNOWN ***"

CITY                 STATE
========================= =====
Albany                    NY
Albuquerque               NM
Kansas City               MO
St. Louis                 MO
⇓
QLI>
```

The *first-expression* can also be used in a Boolean expression or in an assignment statement. Suppose you want a list of cities in Missouri, but you cannot remember the two-letter abbreviation for that state:

```
QLI> print city, state of cities with
CON> state = first state from states with
CON> state_name ="Missouri"
CITY                 STATE
========================= =====
Jefferson City            MO
Kansas City               MO
St. Louis                 MO
QLI>
```

When you use a **first** expression in a stand-alone **print** statement, you must parenthesize the expression to distinguish it from the record selection expression's first-clause:

```
QLI> print first state from states with state_name = "Missouri"
CON> ??
** QLI error: expected "relation_name", encountered "?" **
QLI>

QLI> print (first state from states with state_name =
CON> "Missouri")
MO
QLI>
```

Formatting RSE Output

Qli allows you to format output in a variety of ways. You can format the way values are displayed using edit strings, described in the next section. You can specify column headers other than the default headers to make output more meaningful. Changing column headers is described later in this chapter.

For more sophisticated formatting, qli includes the report writer, a utility for formatting query output. The report writer is described in Chapter 8, *Writing Reports*.

Using Edit Strings to Format Output

You can change the way data is formatted using an *edit string*. An edit string specifies one of the following formats for a value:

- Alphabetic
- Numeric
- Date

For example, without an edit string, **qli** formats POPULATION field from CITIES as follows:

```
QLI> print city, population of cities with state = 'CA'
           CITY            POPULATION
========================== ===========
Sacramento                      275741
Fresno                          218202
Los Angeles                    2966850
San Diego                       875538
San Francisco                   678974
QLI>
```

An edit string on the POPULATION field provides a more legible display. The edit string is introduced with the **using** keyword. For example:

```
QLI> print city, population using z,zzz,zz9 -
CON> of cities with state = 'CA'
           CITY            POPULATION
========================== ==========
Sacramento                     275,741
Fresno                         218,202
Los Angeles                  2,966,850
San Diego                      875,538
San Francisco                  678,974
QLI>
```

The z in the edit string represents a digit, possibly blank-filled, and the 9 represents a digit. Of course, if the field is missing, **qli** prints only blanks for the field.

Formatting Dates

Edit strings are useful in printing dates in something other than **qli**'s default date format ("15-Jan-1990"). The following query prints today's date using an edit string.

```
QLI> print "today" using
CON> w(15)" the "dd"th of "m(10)" in the year "y(4)
Friday the 15th of January in the year 1990
QLI>
```

The above query illustrates the following points about displaying and formatting dates:

- The statement includes date format directives that specify how to display a date. For example, w(15) means print the full weekday name.
- Words inside quotes are printed literally, including blank spaces.
- There can be no space between a format directive and a quoted string.

Of course, when you print the date, the result will be different than this example. Furthermore, if you incorporate this edit string in a query, it yields the incorrect forms "1th", "2th", "3th", "21th", and so on.

Using edit strings to format numeric, alphabetic, and date output is described in the section on field attributes in the *Qli Reference*.

Specifying Column Headers

When **qli** displays the results of a query, it uses the field name as the column header. If the field name contains an underscore, **qli** prints the column header on two or more lines. For example, the following query prints some vital statistics about baseball stadiums in the City of New York:

```
QLI> for baseball_teams with city = 'New York'
CON> print home_stadium, left_field, center_field, right_field

      HOME                    LEFT        CENTER        RIGHT
    STADIUM                   FIELD        FIELD        FIELD
======================== =========== =========== ===========
Yankee Stadium                   312         417         310
Shea Stadium                     338         410         338
QLI>
```

You can change the column header. For example, the following query provides column headers for each of the fields:

```
QLI> for baseball_teams with city = 'New York'
CON>  print home_stadium ('Home Stadium'),
CON>  left_field ('Left'), center_field ('Center'),
CON>  right_field ('Right')

     Home Stadium            Left       Center       Right
========================= ========== ========== ==========
Yankee Stadium                  312        417        310
Shea Stadium                    338        410        338
QLI>
```

If the header you provide is wider than the field value, **qli** expands the column width to accommodate the header. However, the column header may be so wide that the display wraps or disappears off the right side of the window or screen. You can specify line breaks for the header to avoid this problem:

```
QLI> for baseball_teams with city = 'New York'
CON>  print home_stadium ('Home Stadium'),
CON>  left_field ('Left'/'Field'/'Wall'),
CON>  center_field ('Center'/'Field'/'Wall'),
CON>  right_field ('Right'/'Field'/'Wall')
                              Left       Center       Right
                             Field        Field       Field
      Home Stadium            Wall         Wall        Wall
========================= ========== ========== ==========
Yankee Stadium                  312        417        310
Shea Stadium                    338        410        338
QLI>
```

This option is also useful when you concatenate values and want to have a meaningful header. For example, the following query concatenates the CITY and STATE field from the BASEBALL_TEAMS relation to provide a location:

```
QLI> for baseball_teams sorted by city, state with league = 'N'
CON> print city | ', ' | state ('LOCATION'),
CON> team_name ('TEAM'), seating

          LOCATION                    TEAM          SEATING
=============================== =============== ==========
Atlanta, GA                     Braves                53046
Chicago, IL                     Cubs                  37272
Cincinnati, OH                  Reds                  52392
```

```
Houston, TX                    Astros          45000
Los Angeles, CA                Dodgers         56000
Montreal, QUE                  Expos           59149
New York, NY                   Mets            55300
Philadelphia, PA               Phillies        66744
Pittsburgh, PA                 Pirates         54429
San Diego, CA                  Padres          58671
San Francisco, CA              Giants          58000
St. Louis, MO                  Cardinals       50100
QLI>
```

Finally, you can request that no header be printed:

```
QLI> for baseball_teams with league = 'N' sorted by team_name
CON> print team_name (-), home_stadium (-), seating (-)

Astros          Astrodome                          45000
Braves          Atlanta-Fulton County Stadium      53046
Cardinals       Busch Stadium                      50100
Cubs            Wrigley Field                      37272
Dodgers         Dodger Stadium                     56000
Expos           Olympic Stadium                    59149
Giants          Candlestick Park                   58000
Mets            Shea Stadium                       55300
Padres          San Diego Jack Murphy Stadium      58671
Phillies        Veterans Stadium                   66744
Pirates         Three Rivers Stadium               54429
Reds            Riverfront Stadium                 52392
QLI>
```

Qli also supports the **report** statement for more sophisticated formatting. See Chapter 8 for more information.

Accessing Array Data Examples

The records rain_array and temperature_array have been added to *atlas.gdb* to produce array examples. You can use the following queries to produce the examples in QLI:

```
QLI> for cities print city, state, rain_array [1973,1]
```

CITY	STATE	RAIN ARRAY [1973,1]
Juneau	AK	4.37
Indianapolis	IN	2.27
Montgomery	AL	
Phoenix	AZ	0.13
Little Rock	AR	5.66
.	.	.
.	.	.

```
QLI> for cities with rain_array [1973,1] not missing print city,
state, rain_array [1973,1]
```

CITY	STATE	RAIN ARRAY [1973,1]
Juneau	AK	4.37
Indianapolis	IN	2.27
Phoenix	AZ	0.13
Little Rock	AR	5.66
.	.	.
.	.	.

```
QLI> for cities with rain_array [1973,1] > 4 print city, state,
rain_array [1973,1]

                                          RAIN
                                          ARRAY
         CITY                STATE      [1973,1]
========================== ===== ================
Juneau                      AK      4.37
Little Rock                 AR      5.66
Sacramento                  CA      6.87
Atlanta                     GA      8.89
Jackson                     MS      4.59

QLI> for cities with temperature_array [1973,1] not null
CON> print city, state, temperature_array [1973,1]

                                       TEMPERATURE
                                          ARRAY
         CITY                STATE      [1973,1]
========================== ===== ================
Juneau                      AK      18.9
Indianapolis                IN      30.7
Phoenix                     AZ      51.2
Little Rock                 AR      39.7
Sacramento                  CA      44.3
 .                           .       .
 .                           .       .

QLI> for cities with rain_array [1973,1] > 4 or
CON> temperature_array [1973,1] > 70
CON> print city, state, rain_array [1973,1], temperature_array
[1973,1]

                                    RAIN           TEMPERATURE
                                    ARRAY          ARRAY
CITY                        STATE [1973,1]        [1973,1]
========================== ===== ======== ================
Juneau                      AK      4.37             18.9
Little Rock                 AR      5.66             39.7
Sacramento                  CA      6.87             44.3
```

```
Atlanta                    GA      8.89              41.4
Honolulu                   HI      0.67              72.9
Jackson                    MS      4.59              44.3

QLI> for cities with rain_array [1973,1] > 4 or
CON> temperature_array [1973,1] > 70
CON> print city, state, rain_array [1973,1] ("Rainfall"/"Jan
1973"),
CON> temperature_array [1973,1] ("Temperature"/"Jan 1973")

                                        Rainfall       Temperature
CITY                       STATE        Jan 1973        Jan 1973
========================== ===== ================ =============
Juneau                     AK           4.37              18.9
Little Rock                AR           5.66              39.7
Sacramento                 CA           6.87              44.3
Atlanta                    GA           8.89              41.1
Honolulu                   HI           0.67              72.9
Jackson                    MS           4.59              44.3
```

For More Information

Refer to the *Qli Reference* for syntax for the following statements and expressions:

- **print** statement
- *value* expression
- *boolean* expression
- *RSE*

Refer to the GDML section of the *Programmer's Guide* for information on the **matching using** statement.

Refer to the chapter on field definition in the *Data Definition Guide* for information on defining alternate missing values and using edit strings and column headers in field and relation definitions.

Chapter 4
Accessing Data Using SQL

This chapter describes using interactive SQL to retrieve data from an InterBase database. This chapter is not intended to be a detailed discussion of SQL. For more information about SQL, refer to one of the many books written about SQL.

Overview

Using interactive SQL in **qli**, you can:

- Read values from and write values to the database
- Update values in the database
- Define, delete, or modify data definitions

The following sections describe reading data using SQL.

Writing an SQL Select Expression

The *select expression* specifies the search and delivery conditions for record retrieval. It lets you:

- Identify the target relation and desired fields
- Specify search conditions
- Perform a join operation
- Perform a project operation

Selecting Relations and Fields

The *select expression* identifies the relations from which the record stream is created. At its simplest, it specifies only a relation name. The following query specifies the CITIES relations:

```
QLI> select * from cities
```

The * in the select expression is SQL shorthand that represents all fields in the relation in the order in which they were defined. To request specific fields, replace the asterisk with a list of field names. For example:

```
QLI> select capital, state_name from states order by state
```

Specifying SQL Search Conditions

Often you want only a subset of the records in a relation. When you can describe the records you want by comparing values in the records to values you specify, InterBase selects and returns only those records you have described.

The following query returns only those records for which it is true that the value of the STATE field in the record is "CA":

```
QLI> select city, latitude, longitude from cities -
CON> where state = 'CA'

          CITY             LATITUDE LONGITUDE
========================== ======== =========
Sacramento                 38 32N   121 30W
Fresno                     36 44N   119 47W
Los Angeles                34 04N   118 15W
San Diego                  32 43N   117 09W
San Francisco              37 45N   122 27W
QLI>
```

This search condition results in a value of "true," "false," or "missing" for each record in the CITIES relation. An expression in which the value is true, false, or missing is called a *predicate*. The predicate in an SQL select expression begins with the keyword **where**.

The predicate in the above example uses an arithmetic test for equality (state = "CA"). SQL supports these arithmetic comparisons:

- Equality, expressed as **eq** or **=**
- Inequality, expressed as **<** or **lt**, **>**, or **gt**
- Range, expressed as **between** or **bt**

Combining Search Conditions

If your query has more than one search condition, such as cities with a population greater than 1,000,000 and at an altitude above 200 feet, the truth of the Boolean test depends on the combination of search conditions:

```
QLI> select city, state, latitude, longitude from -
CON> cities where population > 1000000 and altitude > 200

          CITY             STATE LATITUDE LONGITUDE
========================== ===== ======== =========
Detroit                     MI   42 23N    083 05W
```

```
Chicago                        IL    41 50N     087 45W
QLI>
```

Only those CITIES records for which both conditions are true will be returned. **Qli** supports logical **not**, **and**, and **or**, evaluating them in that order except when you put an expression in parentheses to change the order of evaluation.

Using SQL Operators

In addition to the standard equality, inequality, and range operators, **qli** supports:

- **all** to compare a value specified with a list of values returned by a query or subquery. For example, this query returns information on rivers that are longer than all rivers in British Columbia:

```
QLI> select river, source, length from rivers where
CON> length > all (select length from rivers where
CON> source = "BC")

RIVER                              SOURCE  LENGTH
================================== ======= ======
Mississippi                         MT      2348
Mississippi-Missouri-Red Rock       MT      3710
Missouri                            MT      2315
Missouri-Red Rock                   MT      2533
```

  This query uses a subquery to first find all of the rivers in Britich Columbia. The results of the subquery are than compared to other rivers in the database. Rivers that are longer than the longest river in British Columbia are returned.

- **any** to compare a value with a list of values returned by a query or subquery. For example the following query returns information on rivers that are longer than any one river in British Columbia:

```
QLI> select river, source, length from rivers where
CON> length > any (select length from rivers where
CON> source = "BC")

RIVER                       SOURCE  LENGTH
=========================   ======  ======
Rio Grande                  CO      1885
Yukon                       BC      1979
Colorado                    CO      1360
Arkansas                    CO      1459
Mississippi                 MN      2348
Mississippi, Upper          MN      1171
```

```
Canadian                       CO      906
Churchill                      ONT     1000
⇓
QLI>
```

You can substitute the SQL logical operator **some** for **any** to produce the same results.

- **exists** to test for the existence of at least one qualifying record in a relation or relations. This expression is true if the record stream specified by the select expression includes at least one record. For example, the following select expression finds the states that have ski_areas in them:

```
QLI> select state_name from states s where exists -
CON> (select * from ski_areas where state = s.state)

          STATE
          NAME
=========================
Massachusetts
New Hampshire
Vermont
QLI>
```

- **like** to test for a field that contains a substring. It is case-sensitive, so you must know whether the field contains upper, lower, or mixed case characters. The like operator takes two wildcard characters: the percent sign (%) wildcard, which matches a run of characters, and the underscore (_), which matches a single character:

```
QLI> select city, state from cities -
CON> where city like '%ton%'

          CITY             STATE
========================== =====
Washington                  DC
Boston                      MA
Trenton                     NJ
Charleston                  WV
Houston                     TX
Burlington                  VT
Charleston                  SC
QLI>
```

Although the **like** operator is most useful for character string comparisons, it can be used with dates and numbers as well. For example, the following query returns the names and dates of entry of states that entered the union in July:

```
QLI> select state, statehood from states where
CON> statehood like "%JUL%"

 STATE            STATEHOOD
======          ===========
ID                3-Jul-1890
NY               26-Jul-1788
WY               10-Jul-1890
```

When you search for a date string, the string should be uppercase, even if the date is displayed in lowercase type.

To search for a substring that includes a percent or underscore character, use the **escape** clause with the **like** comparison. The escape clause lets you define a special character that tells qli to treat the next character literally.

For example, the following query retrieves the names of all cities from the CITIES relation because the percent character matches all names:

```
QLI> select city from cities where city like "%"
```

The following query retrieves no records because it specifies that the percent character be treated literally rather than as a wildcard, and no city is named %.

```
QLI> select city from cities where city like "@%" escape "@"
```

For more information on defining escape characters, refer to the description of the **select** statement in the *Qli Reference*.

- **in** to test for a value that equals one of several other values. The list of values must be parenthesized and separated by commas. The syntax is logically identical to separate equality tests combined with **or** operators:

```
QLI> select city, latitude, longitude from cities where
CON> state in ( CT, RI, MA, NH, VT, ME )

          CITY             LATITUDE LONGITUDE
========================== ======== =========
Hartford                   41 46N    072 40W
Augusta                    44 17N    069 50W
Boston                     42 20N    071 05W
Concord                    43 13N    071 34W
```

```
Providence                    41 50N     071 25W
Montpelier                    44 16N     072 34W
Burlington                    44 28N     073 14W
Portland                      43 41N     070 18W
Presque Isle                  46 42N     068 01W
QLI>
```

You can use the **in** operator to compare a value against the results of a subquery. Subqueries are described later in this chapter

- **null** to test for the absence of a value in a field. A field for which no data is stored is called *missing* or *null*, and its value is treated as unknown. **Qli** prints a missing value as spaces, but a comparison of the field value to spaces returns false, even if the value you are comparing is also missing. To search for a missing value, use the **is null** predicate. The following query displays STATES records for which the value of the CAPITAL field is missing:

```
QLI> select state_name from states where capital is null

          STATE
          NAME
=========================
District of Columbia
QLI>
```

Missing values are further described in the section *Understanding Missing Values*, later in this chapter.

Operating on Groups of Records

You can group records by using the **group by** clause. The **group by** clause groups records by common values. For example, the following query finds the average population for all of the cities in one state, and returns one record for each state:

```
QLI> select state, avg(population) from
CON> cities group by state

STATE     POPULATION
=====    ===========
AK          7000
AL        231135
AR         15846
AZ        789704
CA        1003061
```

```
⇓
QLI>
```

The **having** erator operates on groups of records in much the same way **where** operates on individual records. You provide a search condition or combination of search conditions. InterBase then evaluates the condition for each group of records (for example, all CITIES records with a value of "NY" for the STATE field).

The following statement limits possible cities to those that fall in the specified range, sorts them into groups, and chooses only those groups with large average populations:

```
QLI> select avg (population), state
CON>    from cities -
CON>    where latitude_degrees between 33 and 42  -
CON>       and longitude_degrees between 79 and 104 -
CON>   group by state -
CON>   having avg (population) > 40000

POPULATION    STATE
==========    =====
284413        AL
158461        AR
492365        CO
425022        GA
191003        IA
⇓
QLI>
```

Having selects groups of records, while **where** eliminates individual records. You can use *subqueries* to obtain the same results. For information on writing subqueries, see the section titled *Subqueries* later in this chapter.

Understanding Missing Values

A field that has no assigned value is said to have a *missing value*. Any field can have a missing value. Rather than storing a value for the field, InterBase sets a flag indicating the field has no assigned value. Unless you specify otherwise in the field's definition, **qli** prints blank spaces for numbers, characters, and dates, and nothing for blobs. Fields with missing values are listed last in the sort order.

In addition to the sorting order, missing values have the following characteristics:

- If you perform statistical operations involving a field that is missing for a given record, that record is ignored in the computation.

- A missing value never equals another value, even if the other value is also missing.
- If a missing value is compared to another value, even another missing value, the result of the query is always false.
- A field with a missing value cannot participate as a connecting field in an equijoin or non-equijoin because it never tests true for equality or any other relationship.

For example, many of the records in the CITIES relation do not have a value for the POPULATION field. This absence does not mean that no one lives there. Instead, it indicates the atlas from which the census data for the CITIES relation was taken did not have a listing for that city. Cities with no stored population have the missing value flag set for that field. The following query lists cities with populations less than 200,000 or for which the population is missing:

```
QLI> select city, state, population from cities where -
CON>  population is null -
CON>  order by population

          CITY             STATE POPULATION
========================== ===== ==========
Hartford                    CT
Harrisburg                  PA
Lansing                     MI
Raleigh                     NC
⇓
QLI>
```

See the *Data Definition Guide* for information about defining default values for missing values.

Sorting Records

You often want records in a specific order. InterBase normally returns records in random order. To retrieve records in a particular order, you can instruct InterBase to sort the records by one or more fields or *sort keys*. The order of the sort depends on the datatype of the field, as follows:

- Numeric fields sort numerically, from smallest to largest unless you specify a descending order.
- Date fields sort from earliest to latest, unless you specify a descending sort order.
- Alphanumeric fields are sorted by their ASCII collating sequences. This means that a comes before b and that abc comes before abd, but all uppercase letters sort before all lower case letters unless you specify that the sort should be case_insensitive. The descending option is also available for alphanumeric keys.

To specify a sorting order, use the **order by** option for the select expression. The following query sorts the cities in California by population:

```
QLI> select city, population from cities where
CON> state = "CA" order by population

          CITY              POPULATION
========================== ============
Fresno                       218202
Sacramento                   275741
San Francisco                678974
San Diego                    875538
Los Angeles                 2966850
QLI>
```

The **order by** clause lets you have as many sort keys as you want. To have additional sort keys, separate them by commas. For example:

```
QLI> select city, state, population from
CON> cities where state like 'C%' -
CON> order by state, population

CITY           STATE       POPULATION
============== ======      ==========
Fresno         CA          218202
Sacramento     CA          275741
San Diego      CA          678974
⇓
QLI>
```

Specifying a Sort Order

Each sort key can specify whether the sorting order of the sort key is ascending (the default order for the first sort key) or descending. If you do not specify whether a particular sort key is ascending or descending, **qli** assumes that you want the order specified for the most recent key. Therefore, if you list several sort keys but only include the keyword **descending** for the first key, all keys are assumed to be in descending order until you specify **ascending** to reverse the order.

Retrieving Unique Values

The select expression's **distinct** clause retrieves only the unique values for a field from a record stream. When you ask for a distinct record stream for a field or fields, InterBase considers the list of fields and eliminates records that do not have a unique combination of values for those listed fields.

For example, the following query returns all states in the RIVER_STATES relation :

```
QLI> select state from river_states
```

State names are repeated since many states have more than one river stored in this relation. To list the names of states with rivers only once, use the **distinct** clause, as follows:

```
QLI> select distinct state from river_states

STATE
=====
 AK
 AZ
 CA
 CO
 CT
⇓
QLI>
```

Note

By default, the **distinct** clause orders the records returned by the distinct field. To override this behavior, specify a different sorting order using the **order by** clause.

Do not update or delete records if their selection criteria include a **distinct** clause. Because the **distinct** clause identifies a random occurrence of a value, you could get unexpected results if you use **distinct** in an update or delete expression.

Joining Relations

Queries can create dynamic relationships by matching up records from two or more relations in the same database. The following example prints the population of capital cities. It matches records from the CITIES relation with records from the STATES relation. Resulting records must have common values in the STATE field of each relation. The value in the CITY field of the CITIES relation must also match the value in the CAPITAL field of the STATE relation.

```
QLI> select city, state_name, population from
CON> cities, states where capital = city and
CON> state = state and population is not null

                    STATE
CITY                NAME                POPULATION
=================== =================== ==========
Montgomery          Alaska              177857
Juneau              Alaska                7000
Phoenix             Arizona             789704
Little Rock         Arkansas            158461
Sacramento          California          275741
 ⇓
QLI>
```

Using Aliases

Because the field STATE occurs in both CITIES and STATES, the reference becomes ambiguous. To clearly identify a field reference, you can prepend the relation name to a field name. For example, the above query can be written as follows to clearly identify field references:

```
QLI> select city, state_name, population from
CON> cities, states where capital = city and
CON> cities.state = states.state and population is not null
```

For relations with short names like CITIES and STATES, using the relation name to qualify the field is easy. However, if a relation name is long, or you must qualify several fields, the request can become very cumbersome. SQL provides *aliases* as a shortcut for identifying fields. The alias is a temporary variable (the scope is the query) used for name recognition. It can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character. The following query contain examples of aliases:

```
QLI> select c.city, s.state_name from cities c, states s-
CON> where c.state = s.state -
CON> order by s.state
```

Qli is not sensitive to the case of the alias. For example, it treats "B" and "b" as the same character.

Joining More than Two Relations

Your query can include as many relations as are necessary. For example, the following query joins the CITIES, STATES, and TOURISM relations. Because the TOURISM relation contains records for only those cities with state offices of tourism or chambers of commerce, the query picks up the zip code for a state's tourism office and displays it for every city in the state.

```
QLI> select c.city, s.state_name,t.zip -
CON> from cities c, states s, tourism t -
CON> where c.state = s.state and s.state = t.state -
CON> and c.city = t.city

                  STATE
CITY              NAME           ZIP
================= ============== =====
Juneau            Alaska         99811
Montgomery        Alabama        36130
Little Rock       Arkansas       72201
Phoenix           Arizona        85004
Sacramento        California     95808
 ⇓
QLI>
```

Joining a Relation to Itself

Some problems require that you include two copies of a single relation in a join. Such a join is called a reflexive join or a self-join, although other relations may be included in the join. For example, suppose you want to print the names and populations of cities that are larger than the capital of their states. Since the STATES relation keeps only the name of the capital city, not its population, you have to lookup the population in the CITIES relation. The following query returns the population of capital cities:

```
QLI> select city, population, state_name from
CON> cities c1, states s where
CON> s.state = c1.state and c1.city = s.capital
```

```
                                    STATE
CITY                  POPULATION    NAME
====================  ==========    ==========
Montgomery             177857       Alabama
Juneau                 7000         Alaska
Phoenix                789704       Alaska
Little Rock            158461       Arkansas
Sacramento             275741       California
⇓
QLI>
```

To find the name of cities larger than their state capitals, add another copy of the CITIES relation, using a different alias to distinguish it from the first copy. The second copy should match the first in state name, but have a larger population than the first.

```
QLI> select c2.city, c2.population, s.state_name from
CON> cities c1, cities c2, states s where
CON> s.state = c1.state and c1.city = s.capital and
CON> c2.state = c1.state and c2.population > c1.population
```

```
                                    STATE
CITY                  POPULATION    NAME
====================  ==========    ==========
Birmingham              284413      Alabama
Los Angeles            2966850      California
San Diego               875538      California
San Francisco           678974      California
New Orleans             557615      Louisiana
⇓
QLI>
```

Using Subqueries

Some queries are unnatural or impossible to express as joins. For example, you cannot get a list of states with a smaller than average area by joining states to itself. You can easily get that list with a *subquery*. A subquery is a query within a query that is evaluated first. The results of the evaluation are then passed to the main query. For example, in the following query, the subquery determines the average area for records in the STATES relation. The average is then compared to the area of each state in the STATES relation. The query returns the names of states having an area less than the average state area.

```
QLI> select state_name from states where
CON> area > (select avg (area) from states)
```

The preceding example demonstrates a simple, one-level subquery. None of its values depend on values in the main query. Often, values in a subquery do depend on values from the main query. Suppose you wanted a list of states for which there were three or more cities stored in the CITIES relation:

```
QLI> select state_name from states where
CON> 3 <= (select count (*) from
CON> cities where cities.state = states.state)
```

You can use subqueries to build complex requests. The following query prints the state name, capital, and largest city of states whose area is larger than the average area of states that have at least one city within 100 feet of sea level:

```
QLI> select s1.state_name, s1.capital, c1.city from
CON> states s1, cities c1 where
CON>  s1.state = c1.state and
CON>  c1.population = (
CON>      select max (population) from
CON>      cities c2 where c2.state = c1.state) and
CON>      s1.area > (
CON>          select avg (s2.area) from
CON>          states s2 where exists ( select *
CON>              from cities c3 where
CON>              c3.state = s2.state and
CON>              c3.altitude <= 100 ))

STATE
NAME                 CAPITAL                      CITY
==================== ============================ ====================
Alaska               Juneau                       Juneau
Arizona              Phoenix                      Phoenix
```

```
Colorado            Denver              Denver
Utah                Salt Lake City      Salt Lake City
Texas               Austin              Houston
Nevada              Carson City         Las Vegas
New Mexico          Santa Fe            Albuquerque
California          Sacramento          Los Angeles
Minnesota           St. Paul            Minneapolis
Nebraska            Lincoln             Omaha
Oregon              Salem               Portland
Kansas              Topeka              Wichita
QLI>
```

Note

As relations re-appear in subqueries, aliases are the only way to identify a field with the appropriate instance of its relation.

For More Information

Refer to the *Qli Reference* for the syntax of:

- scalar expression
- select expression
- predicate
- **select** statement

Chapter 5
Writing Data

This chapter discusses field assignments for storing and modifying records, assignments to variables, and deleting records.

Overview

InterBase supports full read and write access to databases with both its GDML and SQL variants. The following sections describe the different methods for assigning values, modifying existing data, and writing the data to the database.

The concepts are the same whether you are using GDML or SQL. Examples of each are used to illustrate concepts.

Assigning Values in Qli

Qli supports several ways of assigning values:

- Automatic prompting for values
- An *assignment* statement for use in **store** and **modify** statements, and to assign values to **qli** variables
- The SQL **set** statement for use in **insert** and **update** statements
- A *restructure* statement that lets you copy all data or a subset from one relation to another

The following sections discuss these options.

Automatic Prompting for Values

Qli's automatic prompting is the simplest way to assign values to fields for the GDML variant of **qli**. It is not available for use with SQL.

For a **store** statement, all you need do is type *store* followed by the relation name. **Qli** then prompts you for a value for each field. For example:

```
QLI> store ski_areas
Enter NAME: 128 Median
Enter TYPE: N
Enter CITY: Lexington
Enter STATE: MA
QLI>
```

If you provide a value that is larger than the datatype of the field, **qli** refuses to accept the value and prompts you again for the field value. For example:

```
QLI> store ski_areas
Enter NAME: Train tracks
Enter TYPE: N
Enter CITY: N. Andover
Enter STATE: Massachusetts
Input value is too long
Re-enter STATE: MA
QLI>
```

If the relation contains a blob field, **qli** automatically calls your default editor for the assignment. Use the editor as you normally do to enter text. When you exit from the editor, **qli** writes the contents of the buffer to the blob field. However, if the blob field for the records in the relation contains non-ASCII data, such as object files or other binary data, you might not want to edit it using your editor. If that is the case, skip the

blob field by making explicit assignments as described in the following section. Use embedded GDML in a program to write to such blob fields.

To see the current value of a field you want to modify, use the **print then modify** construction. For example:

```
QLI> for ski_areas with name = 'Birchwood Acres'
CON> print then modify city, state

NAME             TYPE  CITY
================ ====  ======================
Birchwood Acres   N      Manchester
Enter CITY: Gloucester
Enter STATE: NH
```

Using Prompting Expressions

A very useful value expression, especially in procedural functions, is the *prompting value expression*. Figure 5-1 shows the format of an *assignment* statement with a prompting value expression.

Figure 5-1. Assignment with Prompting Value Expression

database-field	=	*. quoted-string

The value expression in Figure 5-2 is a string that **qli** displays when you make assignments. For example, when storing records for the SKI_AREAS relation, as described in a previous example, the automatic prompting does not provide you with the possible values for a field. A prompting expression you supply can give more information, as follows:

```
QLI> store ski_areas using
CON> begin
CON> name = *.'name of new ski area'
CON> type = *.'type of skiing ([N]ordic, [A]lpine, [B]oth)'
CON> city = *.'city where area is located'
CON> state = *.'state where area is located'
CON> end
Enter name of new ski area: Train tracks
Enter type of skiing ([N]ordic, [A]lpine, [B]oth): N
Enter city where area is located: N. Andover
Enter state where area is located: MA
QLI>
```

As it does in automatic prompting, **qli** supplies the Enter for prompting expressions.

Modifying a Record

To modify a record, use the **modify** statement. To use **modify**, type *modify*, the name of the field or fields you want to change, and a record selection expression to select the record to change. For example:

```
QLI> modify city, state of ski_areas with name =
CON> 'Birchwood Acres'
Enter CITY: Manchester
Enter STATE: MA
QLI>
```

If the field to be changed contains blob data, **qli** calls your default editor with the contents of the selected record's blob field in the editing buffer. Make the change to the text and exit. As with the preceding **store** example, avoid editing non-ASCII blob data with your editor.

If you choose not to store or modify a record for which you have already supplied some, but not all, values in answer to the storage prompts, use an end-of-file key to terminate input. If you were providing values to a **store** statement, the record is not stored. If you were modifying a field or fields, the fields for which you already supplied new values retain their old values.

Prompting Assignment Defaults

If you only press Return in response to a prompt in **qli**, that field is stored as missing. If you enter one or more blank spaces plus a carriage return, **qli** stores blank spaces. For example, the following **store** statement assigns the missing value for the NAME field with a carriage return and two blank spaces to STATE with two spaces and a carriage return:

```
QLI> store ski_areas
Enter NAME: <press Return>
Enter TYPE: N
Enter CITY: Ward Hill
Enter STATE:
QLI> print ski_areas with name missing and state = '  '
        NAME          TYPE           CITY           STATE
==================== ==== ======================== =====
                     N    Ward Hill
QLI>
```

When you are modifying a field and you press carriage return at the prompt, a missing value is assigned to the field. If you want to retain the original value for the field,

either retype the value, or abort the modify operation and construct a modify statement using direct assignment. Direct assignment is described in the next section.

See the section later in this chapter titled *Assigning Missing Values* for more information about handling missing values.

Storing Multiple Records

Qli lets you repeat commands or procedures. For example, to store ten records for the SKI_AREAS relation, use the **repeat** statement with a numeric argument with the **store** statement:

```
QLI> repeat 10 store ski_areas
Enter NAME: Wake's Farm
Enter CITY: Chelmsford
Enter STATE: MA
Enter TYPE: N
Enter NAME:
          ⇓
QLI
```

As you can see, **qli** goes from one record's values to the next without indicating you have started storing a new record. You can use the **repeat** statement for repetitive operations such as data entry. You might also want to include the whole operation, including the **repeat** statement, in a procedure, as discussed in Chapter 7.

Using the Assignment Statement

The *assignment* statement gives you more explicit control over assignments than **qli**'s automatic prompting feature. For example, in a **store** statement, you can assign values to only those fields you want in whatever order you choose. As seen earlier, the automatic prompting feature requires assignments to all fields in the order specified in the data definition for the relation.

The *assignment* statement has the general form shown in Figure 5-2.

Figure 5-2. Qli Assignment Statement

database-field	=	value-expression

The form of the value expression depends on the datatype. Assignments to character string and date fields should be quoted, numeric datatypes do not need quoted, and blob fields require the editor. These considerations are discussed below in the section titled "Datatype Notes."

If the field value contains alphanumeric characters, you must enclose it in single (') or double (") quotation marks. If you do not quote the string, **qli** assumes that the literal is a field name or local variable and returns an error if the field is unknown. For example, if you assign the unquoted value *Detroit* to a field, **qli** returns the following message:

```
QLI> for cities with city = 'Detrit'
CON> modify using city = Detroit
Error: "DETROIT" is undefined or used out of context
QLI>
```

However, quoting the value expression fixes this problem:

```
QLI> for cities with city = 'Detrit'
CON>modify using city = 'Detroit'
QLI>
```

For assignments to more than one field during a **store** or **modify** statement, use the **begin-end** statement to structure a block of assignments. For example:

```
QLI> store states using
CON> begin
CON>  state = 'MV'
CON>  state_name = 'Morava'
CON>  area = 542
CON>  statehood = '19 DEC 87'
CON>  capital = 'Brno'
CON> end
QLI>
```

When you are storing a record using assignment statements, any fields not referenced by an assignment are stored with a missing value. When modifying records, any field not referenced is left as it was before the modify operation.

Begin-End Blocks

The word **begin** in the last query in the preceding section starts a *begin-end* block, two or more **qli** statements that are grouped into a single compound statement. You can use begin-end blocks wherever the syntax calls for a single statement, but where you need multiple statements. The begin-end block in an assignment is introduced with the **using** keyword.

Both **store** and **modify** take a single statement as their "action," so to assign several values, you must group the assignments in a begin-end block. Begin-end blocks can be nested arbitrarily deep.

Using For Loops

If you prefer to have **qli** do a little more of the work involved in storing new records, you can include the record storage in a **for** loop. That way you can provide an RSE to find a record you want and store a new record by assigning values from the old record to the new record.

Storing Multiple Records Using a For Loop

You may find yourself modifying dozens of new records. Rather than enter multiple **modify** statements, you can use **a for** loop combined with a **modify** statement. For example, suppose the U.S. converts to the metric system. You could convert the values in the altitude field to meters, as follows:

```
QLI> for cities modify using altitude = altitude/3.28
QLI> for cities with state = ca print city, altitude

CITY             ALTITUDE
================ =========
Sacramento
Fresno              90
Los Angeles         46
San Diego            3
San Francisco       18
```

SQL-Style Assignment

SQL does not support automatic prompting for values. Instead, **qli** provides true assignments in:

- Record storage.

  The SQL **insert** statement requires a separate field target list and value as ment list. For example, the following statement stores a new SKI_AREAS re

```
QLI> insert into ski_areas -
CON> (name, city, state, type) -
CON> values ('Balmont Farm', 'Canaan', 'NY', 'N')
QLI>
```

  If you leave a field in the relation out of the target list, that field is assigned missing value. Be very careful with the order of the value list. Since values assigned to fields in the order in which they are listed, an error in the value can produce unexpected results.

- Record modification.

  The SQL **update** statement uses the **set** assignment substatement. The follo statement updates the SKI_AREAS record just stored:

```
QLI> update ski_areas -
CON>set type = 'B' -
CON>where city = 'Canaan' and state = 'NY'
QLI>
```

Datatype Notes

As discussed earlier in this chapter, the datatype of your input is sometimes important when you are assigning values. If the datatype of the field to which you asign a value is:

- Numeric, list the number without spaces. For example, the following statement reflects the sinking of Baton Rouge:

```
QLI> for cities with city = 'Baton Rouge'
CON> modify using altitude = 41
QLI>
```

- Character, quote the string. For example, the following statement corrects a non-existent misspelling in the STATES relation:

```
QLI> for states with state = 'LA'
CON> modify using capital = 'Baton Rouge'
```

- Date, quote the string.
- Blob, let **qli** prompt you for a value, or place the word **edit** on the right side of the assignment.

The following two sections discuss assignments to date and blob fields, respectively.

Date Fields

InterBase supports a date datatype with a range of 1 January 100 to 11 December 5941. In **qli**, all you have to do to assign a value to a date field is provide a quoted string that looks something like a date. For example, if you assign the value "1-15" or "1/15" to a date field, the InterBase access method defaults to the current year, converts the literal to an internal representation of 1 January 1990, and then stores the date.

Qli accepts several date formats. For example, the following dates are acceptable forms of data entry for 15 January 1990:

```
"1-15-90"
"1/15/90"
"1/15/1990"
"January 15, 90"
"January 15, 1990"
"15.1.90"
```

However, if you assign an unquoted string such as 1-15 or 1/15 to a date field, the punctuation is interpreted as arithmetic operators. In this case, InterBase stores -15 or 0.20424782, respectively. You can avoid this misinterpretation by always quoting strings in assignments to date fields.

Relative Dates

Qli also recognizes relative dates. For example, suppose today is January 15, 1990. The following values equate to the specified dates:

- **yesterday** assigns the value 14-Jan-1990
- **today** assigns the value 15-Jan-1990
- **now** assigns the value *current time on* 15-Jan-1990
- **tomorrow** assigns the value 16-Jan-1990

For example:

```
QLI> modify statehood of states with state_name = "Iowa"
Enter STATEHOOD: yesterday

STATEHOOD
==========
14-Jan-1990
```

Other facts to remember about the date datatype are:

- Punctuation in dates is ignored, except when you use periods as separators. If you use dot separators (for example, '15.1.90') InterBase assumes a "day—month" ordering.
- InterBase accepts three-letter abbreviations of months, such as "APR" and "JUL,". However, any shorter abbreviations are subject to first match. Thus, "J 22" is interpreted as "January 22," although you may have intended "July 22."
- If you do not specify the century in the year, InterBase looks at the current year. Consider the dates "7/22/20" and "7/22/40." If the difference between the current year and the year in the date is greater than or equal to 50, **qli** assumes that you mean the next century. Therefore, "7/22/20" is stored as "7/22/2020." However, "7/22/40" is stored as "7/22/1940." Storage of "7/22/40" will be accepted as input for 1940 until 1990. After that year, anyone assigning a date in 1940 without specifying the *19* will find that InterBase stored a date in the year 2040.
- The value for missing dates is 17 November 1858, although **qli** returns blanks for date fields that are missing.

Blob Fields

A *blob* is a type of field whose structure is not specified to the database. Blobs are best suited for the storage of unformatted data such as text or images, and for data whose format does not correspond to relational structures. For example, blobs can be used to hold graphic images. Blobs can also be arbitrarily large, while string fields are

restricted to approximately 32,000 bytes. However, qli cannot display a string field larger than 255 characters.

The blob provides unstructured data with many of the advantages commonly claimed for structured data in database management systems. Although a blob field in a record looks more like a sequential or stream file than, say, like a part number, it is under full transaction control, can be maintained by the same utilities as more structured data, and can be manipulated with minor variants of the **qli** statements used for the structured data.

Writing to a blob field containing ASCII characters is very straightforward in **qli** because it calls the editor when you want to store or modify a blob field. For example, the following **modify** statement relies on **qli** to prompt for values. When it gets to the blob field, GUIDEBOOK, it starts an editing session:

```
QLI> modify guidebook of tourism with state = 'NY'
```

Qli calls your default editor so that you can enter the blob data for the GUIDEBOOK field. Enter text and exit when you are finished.

You can also make assignments to blobs explicitly, as in the following example. As you can see, the *assignment* statements for the blob fields, OFFICE and GUIDEBOOK, equate them with the keyword **edit**, thus instructing **qli** to call the editor to enter data:

```
QLI> store tourism using
CON> begin
CON>  state = 'MV'
CON>  zip = '21868'
CON>  city = 'Brno'
CON>  office = edit
CON> guidebook = edit
CON> end
QLI>
```

Once you type the carriage return after **end**, **qli** calls your default editor so that you can enter the blob data for the OFFICE field. Enter text and exit when you are finished.

Qli then calls your default editor to enter the blob data for the GUIDEBOOK field. Enter text and exit when you are finished.

You can include the contents of another blob field by establishing context for another blob field and giving its name as the argument to **edit**:

```
QLI> for tour1 in tourism with state = 'NY'
CON> store tour2 in tourism using begin
CON> tour2.city = 'Brooklyn'
CON> tour2.guidebook = edit tour1.guidebook
```

```
CON> end
QLI>
```

Qli calls your default editor so that you can edit the blob data for the GUIDEBOOK field from the outer **for** loop. Edit the text and exit when you are finished.

When the editor starts up, the contents of the GUIDEBOOK blob field from the outer **for** loop are in the editing buffer. You edit the text. When you exit from the editor, TOUR1.GUIDEBOOK remains as it was, but TOUR2.GUIDEBOOK now contains the version of TOUR1.GUIDEBOOK you edited. If you quit the editor rather than exit, **qli** overwrites the former value of the blob field with the contents of the other blob field. To restore the old value of the blob field, you must roll back the update.

Note

Not all types of blobs are interpreted by **qli**. For example, you can define your own blob subtype in **gdef**. **Qli** tries to filter such a blob into a readable form. If it cannot, you will not be able to read or access the blob data through **qli**. However, you can write a blob filter to translate data into a more readable format. For more information on blob filters and user-defined blob subtypes, refer to the chapter on blob filters in the *Programmer's Guide*.

Assigning Missing Values

Chapter 3 discusses missing values in detail. To review, when you retrieve a field whose value is missing, **qli** returns blanks for all datatypes except blobs, for which it returns nothing.

In **qli**, the simplest way to assign a missing value is to press Return at the storage or modification prompt. This method is discussed earlier in this chapter in the section titled *Prompting Assignment Defaults*. If you are using explicit assignment statements, the easiest way to store a field as missing is to omit an assignment to that field. For example:

```
QLI> store cities using
CON> begin
CON> city = 'Chelmsford'
CON> state = 'MA'
CON> end
QLI>
```

This **store** statement omits assignments to the ALTITUDE, POPULATION, and the fields that combine to make up the LATITUDE and LONGITUDE fields. InterBase interprets those fields as having missing values.

You can also store a missing value by assigning **null** to the field. For example:

```
QLI> store cities using
CON> begin
CON> city = 'Tyngsboro'
CON> state = 'MA'
CON> altitude = null
CON> latitude_minutes = null
CON> latitude_degrees = null
CON> latitude_compass = null
CON> longitude_minutes = null
CON> longitude_degrees = null
CON> longitude_compass = null
CON> end
QLI>
```

If you use **qli**'s prompting feature and want to assign the missing value to a field, press Return in response to the prompt. Do not type "null" in response to a storage prompt, because **qli** stores the string "null" as the value if the word null fits the datatype and length of the field, or re-prompts you for a valid value if the string "null" is invalid.

Assigning Missing Values Using Gdef

Another way to assign a missing value when you use the prompting feature is to define a substitute missing value using **gdef**. For example, you might define a missing value of "-1" for the LATITUDE_DEGREES and LATITUDE_MINUTES, fields, and a missing value of "x" for the LATITUDE_COMPASS field. Whenever you store these missing values, InterBase automatically assigns the missing value to the field. You can assign this substitute missing value using a **store** or **modify** statement instead of omitting the assignment or assigning **null**, or when you use **qli**'s prompting feature. Make sure that the substitute value could not be mistaken for a legitimate non-missing value for the field. For example, a positive substitute missing value for the LATITUDE_DEGREES field is not a wise choice.

In summary, if you are doing data entry in **qli** and want to store fields as missing, you should do one of the following:

- Press the Return key in response to the prompt for the field value.
- Use the full form of the **store** statement and do not assign anything to any field you want to store as missing.
- Use the **null** assignment.
- Assign the substitute missing value to the field.

Troubleshooting Assignment Problems

You may encounter the following problems when you assign values in **qli**:

- A value you tried to store or modify did not fit. Check the field's characteristics and try again.
- A field value you tried to assign violated the uniqueness of an index that includes that field. Try another value.
- A value could not be converted into the database field's datatype.
- You tried to store or modify a date field with a value outside the range 1 January 100 to 11 December 5941, or an invalid date such as 29 February 1986. Try a value within the valid range. If the range is not adequate, you cannot use the date datatype.
- You tried to store data directly into a blob. Use the **edit** option described above in the section title *Blob Fields*.
- A field value you tried to store or modify violated a validation expression clause for a field. Check the valid values and try again.
- An operation you attempted violated a security class. If you created and maintain the database you are using, check any security classes you may have defined. Otherwise, ask the owner of the database about security classes that he or she may have defined.
- An operation you attempted violated a trigger condition for a modify or store operation. If you created and maintain the database you are using, check any triggers that you may have defined for the relation to which you wrote. Otherwise, ask the owner of the database about triggers he or she may have defined.

See the chapter on security in the *Data Definition Guide* for more information about validation expressions, security classes, and triggers.

Declaring User Variables

Qli supports a declare statement that allows you to declare two types of user variables, global and local.

Defining Global Variables

Variables defined at the **qli** prompt are *global* variables. Global variables you declare are available throughout your **qli** session. You do not have to have a database readied to use global variables.

The following commands declare two variables and display their attributes:

```
QLI> declare state_code char [2]
QLI> declare city_pop long
QLI> show variables
Variables:
        STATE_CODE                text, length 2
        CITY_POP                  long binary
QLI>
```

Assign values to the variables at the QLI> prompt:

```
QLI> state_code = 'MA'
QLI> print state_code
STATE
CODE
=====
MA
QLI>
```

Note

You cannot define a variable of the type blob or array.

You can use variables in **qli** just as you do in programs. For example, the following commands prompt for a value for the STATE_CODE variable and use the variable in a record selection expression:

```
QLI> state_code = *.'two-character state code'
Enter two-character state code: NY
QLI> for cities with state = state_code
CON> print city, state, population
          CITY            STATE POPULATION
========================= ===== ==========
Albany                     NY
```

```
Buffalo                        NY          357870
New York                       NY         7071639
QLI>
```

Variables are also very useful in procedures, which are discussed in Chapter 7.

Defining Local Variables

If you declare a variable within a **begin-end** block, it is a local variable, and its scope is that block. You can use the variable within that block and other blocks nested in it, but as soon as you leave the block, the local variable disappears. If a local variable has the same name as a global variable, only the local variable is visible in the block where it is declared. When you leave that block, the global variable becomes available again, with the same value it had when you declared the conflicting local variable. Although the terms "local" and "global" suggest that there are only two levels of variable, you can declare variables at any level in nested **begin-end** blocks. Their scope is the current block and any blocks nested within the current block.

Deleting Records

To erase a record in **qli**, select the record you want to delete with an RSE. For example, the following query selects all (non-existent) records for ski areas in Florida as the records to be erased:

GDML	`QLI> erase ski_areas with state = 'FL'`
SQL	`QLI> delete from ski_areas where state = 'FL'`

The following GDML statement erases the same records, but selects them in the **for** command:

```
QLI> for ski_areas with state = 'FL'
CON> erase
QLI>
```

You can delete records from a multi-relation view only if there is an *erase* trigger for the view. See the chapter on security in the *Data Definition Guide* for more information about defining triggers.

Do not delete records through a record selection expression that includes a **reduced** or **distinct** clause.

Be very careful with line continuation when using erase. For example, if you type:

```
QLI> erase ski_areas
QLI>
```

all the records for the SKI_AREAS relation are erased. You might really have meant to continue:

```
QLI> erase ski_areas -
CON> with state = FL
```

For More Information

For more information, refer to the *Qli Reference* for the syntax of:

- *assignment*
- **store**
- **modify**
- **insert**
- **update**

For information on defining security classes, refer to the *Data Definition Guide*.

For information on writing blob filters, refer to the *Programmer's Guide*.

Chapter 6
Defining Metadata

This chapter describes using **qli** to define, modify, and delete databases, relations, fields, and indexes.

Overview

Qli supports an interactive subset of **gdef** capabilities, the data definition and modification compiler, and a subset of SQL expressions for defining data. The first three sections describe defining metadata using the InterBase data definition language (DDL); the remaining sections describe performing the same tasks using SQL data definition expressions.

Caution

Because the behavior of **qli**'s data definition commands differs considerably from its data manipulation statements, *please read this chapter completely before trying the examples.* In particular, be sure you have read and understood the section titled *Interactive Data Definition and Transactions*. Otherwise, you may delete data you wanted to keep.

The following table shows what interface to use for defining metadata. An **x** indicates that you can define a feature in the specified interface. Note for example, that triggers must be defined in **gdef**, and views can be defined using SQL, but not GDML.

Table 6-1. Overview of InterBase Data Definition

Object	**QLI** GDML	**QLI** SQL	**GDEF**
Database	x	x	x
Relation/Table	x	x	x
Global fields	x		x
Local field	x	x	x
Index	x	x	x
Security	x	x	x
Trigger			x
View		x	x

Defining Metadata Using GDML

This section describes using GDML operations through the **qli** interface to define, modify, and delete metadata. Using GDML expressions, you can define the following objects through the **qli** interface:

- Database
- Fields
- Relations
- Indexes

Defining a Database

You can create a new empty database through **qli** using the **define database** command. **Qli** creates a new database and opens it for use. Unlike **gdef**, **qli** overwrites existing databases on operating systems that do not support multiple versions of files. The only option you can supply is the database handle. For example:

```
QLI> define database expenses.gdb as expenses
QLI> show all
Database "expenses.gdb" readied as EXPENSES
```

Note

A database handle is declared only for the **qli** session. It is not stored as part of the database definition. A database handle should be unique to avoid confusion. If you have readied more than one database, specified database handles, and you want information from a specific database, you can use the database handle associated with that database with the **show** command:

```
show <db_handle.entity>
```

For example, if you assigned the handle "emp" and want to display the relations from the "emp" database, use this command.

To create a database with a page size other than 1024 bytes, you must use **gdef**. You can directly modify the description and security class of a database in **qli**, by modifying the system relation RDB$DATABASE. For example, the description field is a blob field intended to hold text describing the database:

```
QLI> modify rdb$description of rdb$database
```

Qli calls your default editor so you can enter a description. Enter text and exit when you are done.

If you have more than one database readied, specify a database handle to modify the correct database. For example:

```
QLI> modify rdb$security_class of atlas.rdb$database
```

Note

When you modify the system relation directly, you do not see any modifications to the database definitions until the database is finished and then readied again.

Defining Global Fields

Using DDL, you can define a field independently or as part of a relation definition. Either way, InterBase regards the field as a *global* field. This means that once the field is defined, it can be added to relation definitions.

Global fields are created in **qli** with the **define field** command. In its simplest form, it includes only the field name and datatype:

```
QLI> define field description blob
QLI> define field identifier long
```

Datatypes are fully described in the chapter on defining fields in the *Data Definition Guide*.

You can also define a field with a scale factor, edit_string, or query_name:

```
QLI> define field cost long scale -2 edit_string -
CON> "$$$,$$$,$$$.99"
QLI> define field department_number char [2] -
CON> query_name dept

QLI> show global fields


Global fields for database EXPENSES:
    DESCRIPTION                    blob
    IDENTIFIER                     long binary
    COST                           long binary, scale -2
    DEPARTMENT_NUMBER              text, length 2
```

If you want to use other field options, such as segment length for blob fields, subtype, query header, position, validation, or missing values, you must use **gdef**.

Modifying Global Fields

You can change any characteristic of a global field you define with **qli** using the **modify field** command. However, you cannot change a datatype to or from a blob. The following command changes the datatype from long to short and adds an edit string to the IDENTIFIER field:

```
QLI> modify field identifier short edit_string "ZZ9"
```

Deleting Global Fields

You can delete fields in **qli** using the **delete field** command. However, you cannot delete a global field used in relations until you have removed all references to it, either by dropping the field from relations or by dropping the relations.

Note

The keywords **delete** and **drop** are synonymous in the context of **qli** data definition.

The following example deletes the DESCRIPTION field, then displays the remaining global fields in the expenses database:

```
QLI> delete field description
QLI>
QLI> show global fields
Global fields for database EXPENSES:
    IDENTIFIER                      short binary
    COST                            long binary, scale -2
    DEPARTMENT_NUMBER               text, length 2
```

Note

When you drop a field from a relation, *you lose all data in the field.*

Defining Relations

To create a relation with **qli**, use the **define relation** command. You should include whatever fields you want in the order you want them to appear. If you have previously defined global fields, you can include them by reference or use the **based_on** clause to rename them locally. For example:

```
QLI> define relation trips
CON> start_date date,
CON> end_date date,
CON> purpose blob,
CON> results blob,
CON> trip_id based_on identifier,
CON> cost
QLI>
```

The **based_on** clause creates a duplicate data definition with a new name you specify. The **based_on** clause is described further in the section *Copying Relations*.

Modifying Security Classes

To add a description or security class to the relation, you must directly modify the RDB$DESCRIPTION or RDB$SECURITY_CLASS field of the RDB$RELATIONS system relation record right after committing the original change. For example:

```
QLI> modify rdb$description of rdb$relations with
CON> rdb$relation_name = "TRIPS"
```

Qli calls your default editor so that you can enter a description. Enter text and exit when you are done. For more information on the system relation RDB$RELATIONS, see the appendix on system relations in the *Data Definition Guide*.

You can also define a relation using fields from previously defined relations. In this database, the relation STOPS uses several of the fields from the TRIPS relation:

```
QLI> define relation stops
CON> start_date,
CON> end_date,
CON> trip_id based on identifier,
CON> stop_id based on identifier,
CON> hotel char [30]
QLI> show stops
    STOPS
        START_DATE              date
        END_DATE                date
```

```
TRIP_ID                 short binary
STOP_ID                 short binary
HOTEL                   text, length 30
```

The only options you can include on fields are an edit string and query name. You must use **gdef** if you want to use other options or include computed fields in your relation.

Copying Relations

To use a copy of a relation for a template for a similar relation, use the **based_on** clause to modify the **define relation** command. This copies field names and field characteristics from the old relation to new relation. It does not copy triggers, indexes, security, or data. To copy data, use the **restructure** operation, described in the next section.

For example, suppose you have become an avid alpine skier and you want to store information on downhill ski areas. Rather than create a new relation, you can copy the CROSS_COUNTRY relation definition and modify it to store information on your favorite slopes. The **based_on** clause of the **define relation** command lets you create the new relation definition in a single step:

```
QLI> define relation downhill_slopes
CON> based on relation cross_country
```

You can also define a new relation based on a relation in another database. For example, suppose a database called new_england.gdb holds information on recreation in New England. To copy a relation that stores data about hiking trails in New England, you only need to specify database handles in the **define relation** statement, as follows:

```
QLI>  ready new_england.gdb as NE
QLI> ready atlas.gdb as atlas

QLI> define relation atlas.hiking_trails
CON> based on relation NE.hiking
```

The following considerations apply when copying a relation:

- If you copy a view, it is copied as a relation.
- If you copy an external relation, the new relation is defined as an internal relation.

Copying Data within a Database

Data can be moved between relations using the **restructure** operation. The restructure operation is useful for:

- Copying the data from a relation in a database to a new location in the same or another database
- Storing the data selected by a record selection expression as a relation
- Importing external data into a relation in a database

The restructure operation equates a relation with another relation or a record selection expression. In response, **qli** stores the selected records into the relation. For example, suppose you want a relation that stores information on cross country ski areas in Massachusetts. Using the **based on** clause, you can copy the CROSS_COUNTRY relation definition to the MA_CROSS_COUNTRY relation:

```
QLI> define relation ma_cross_country
CON> based on cross_country

QLI> show ma_cross_country
  MA_CROSS_COUNTRY
      AREA_NAME           varying text, length 20
      CITY                varying text, length 25
      STATE               varying text, length 4
      PHONE               text, length 10
      NUM_TRAILS          long binary
  ⇓
QLI>
```

To copy the data to the new relation MA_CROSS_COUNTRY, set the new relation equal to a record selection expression that extracts the appropriate data from the CROSS_COUNTRY relation:

```
QLI> ma_cross_country = cross_country with state = "MA"
```

To see a list of the cross country ski areas in Massachusetts, type:

```
QLI> print area_name, city of ma_cross_country sorted
CON> by area_name

          AREA
          NAME         CITY
====================== ==================
Brodie Mountain        New Ashford
Bucksteep Manor        Washington
Canoe Meadows          Pittsfield
Cummington Farm STC    Cummington
```

```
⇓
QLI>
```

Note

When using the restructure operation to move data between relations, the source of the existing relation and the new relation do not have to be exactly the same. **Qli** identifies common fields and ignores all others.

Copying Data between Databases

The restructure operation can also be used to transfer data between relations in different databases. To transfer data from one database to another, you must have a relation in each database that has common fields. For example, suppose you have a CROSS_-COUNTRY relation defined in a database called new_england.gdb. To transfer data from the CROSS_COUNTRY relation in the atlas database to the CROSS_ COUNTRY relation in the new_england.gdb database, specify the database handles:

```
QLI> ready atlas.gdb as atlas
QLI> ready new_england.gdb as new_england

QLI> new_england.cross_country = atlas.cross_country
```

Importing Data from an External Relation

An *external relation* is a relation whose data resides in an external file rather than in an InterBase database. External relations can store data composed of fixed-length fields, such as phone numbers or state name abbreviations. You cannot define an external relation in **qli**. Defining external relations using **gdef** is described in the *Data Definition Guide*.

To import data from an external relation, you must define a relation corresponding to the external relation. For example, suppose you have a database with the following definitions:

```
define database "sample.gdb";
define field first_name char [10];
define field last_name char [10];

define relation PERSONAL_EXT external_file "external.file"
kids char [1] query_name number_of_kids,
first_name,
last_name,
dob char [10] query_name date_of_birth,
crlf  char [1];
```

```
define relation PERSONAL
first_name
last_name,
number_of_kids short;
date_of_birth date;
```

The internal relation must closely correspond to the external relation. The following rules apply to restructuring data between an internal and an external relation:

- The fields containing data you want to import must be of fixed length and cannot be separated by delimiters. You can use delimiters if you define fields as part of your data definition that will act as placeholders for source fields with delimiters.
- The fields in the external relation must all be of the type (fixed) character.
- All the fields in the external relation must be included in the restructuring.
- If the record is terminated by a carriage return, an extra field must be defined at the end of the relation which is of length 1. This serves as a place holder for the Return or new line character.

  For example, the CRLF field of the PERSONAL_EXT relation is a placeholder for the carriage return at the end of each line of the external file.
- The restructure operation only moves data between fields of the same name or the same query name. This distinction allows you to translate character fields to date or integer fields.

  For example, the DOB field of the external relation stores a birthdate as fixed characters. By assigning the query name date_of_birth, the data from the DOB field can be imported to the DATE_OF_BIRTH field in the internal relation and translated to a date string.

Now you can use the restructure operation to transfer the data from PERSONAL_EXT to PERSONAL:

```
QLI> personal = personal_ext
```

Modifying Relations

The **modify relation** command adds, modifies, or drops fields from a relation. In a single **modify relation** command, you can add one field and drop another.
When you modify a field, you can change its edit string, query name, or the global field on which it is based.

```
QLI> modify relation stops
CON> drop hotel,
CON> modify stop_id edit_string "ZZ",
```

```
CON> add campground char [30]
QLI> show stops
    STOPS
        START_DATE              date
        END_DATE                date
        TRIP_ID                 short binary
        STOP_ID                 short binary
        CAMPGROUND              text, length 30
```

Deleting Relations

Use the GDML **delete** command to delete a relation. When you delete a relation from a database, you delete all data in the relation. You cannot rolback a **delete** command.

```
QLI> delete relation <relation-name>
```

Defining Indexes

Qli also includes commands to add, modify, and drop *indexes*. An index is a mechanism used to direct the way in which InterBase searches for records in order to optimize retrievals. When you add or modify an index on a large relation, there is a pause before the QLI> prompt reappears while InterBase builds an index. The following example creates an index on the STOPS field:

```
QLI> define index stop_idx1 for stops
CON> stop_id, trip_id
```

The STOPS index allows duplicate values in the index. To eliminate duplicate values, you can specify an index is unique by assigning the **unique** keyword. The following example creates a unique index for the TRIPS field:

```
QLI> define index trip_idx1 for trips unique
CON> trip_id
QLI> show indexes
Database "expenses.gdb" readied as EXPENSES
    TRIPS
        Index TRIP_IDX1 (unique)
            TRIP_ID
            index TRIP_IDX1 is NOT active
    STOPS
        Index STOP_IDX1
            STOP_ID
            TRIP_ID
```

Note

Defining a unique index while others are using the database may result in a "*duplicate key*" error. If you encounter this error, redefine the index when you have exclusive access to the database.

The **inactive** keyword specifies that the index should not be built or used. This index can later be set to **active**, at which time it is used for retrieval.

Navigating Using Indexes

Optional keywords enable you to create an index that improves the performance and efficiency of certain sorts. To make a descending sort more efficient, specify the **descending** option for the **define index** statement. For example:

```
QLI> define index teams for baseball_teams descending
CON> team_name
```

A descending index is especially useful if you frequently retrieve data from a large table in a "last in, first out" order. For example, you could execute a descending search sorted by date to retrieve the latest records first.

To make an ascending index more efficient, specify the **ascending** option for the **define index** statement, or omit the option entirely. For example:

```
QLI> define index teams for baseball_teams
CON> team_name
```

produces the same result as:

```
QLI> define index teams for baseball_teams
CON> ascending team_name
```

Modifying Indexes

Qli allows you to modify indexes using the **modify index** statement. For example, the index trip_idx1 is specified as inactive. The following example modifies the index trip_-idx1 to make it active:

```
QLI> modify index trip_idx1 active
QLI> show indexes
Database "expenses.gdb" readied as EXPENSES
    TRIPS
        Index TRIP_IDX1 (unique)
            TRIP_ID
    STOPS
        Index STOP_IDX1
            STOP_ID
            TRIP_ID
```

Deleting Indexes

To delete an index, use the **drop index** command followed by the index name. For example:

```
QLI> drop index trip_idx1
```

Note

In order to preserve database integrity, InterBase does not allow you to delete an index that someone else is using.

Defining Metadata Using SQL

This section describes using SQL operations through the **qli** interface to define, modify, and delete metadata. Using SQL expressions, you can define the following objects through the **qli** interface:

- Database
- Fields
- Relations
- Indexes
- Views

The following table summarizes supported SQL metadata operations and the keywords for each operation. If an operation is supported for an object, the associated cell holds an "x" value. If a cell is blank, the operation is not supported.

Table 6-2. Supported SQL Metadata Operations

Object	Create	Alter	Drop
Database	x		x
Column (global fields)			
Column in table (local field)	(create table)	(alter table)	(alter table)
Index	x		x
Security (grant/revoke)	x		x
Table	x	x	x
Trigger			
View	x		x

As the table shows, you cannot define global fields with SQL. Fields are defined within a relation. To define global fields, you must use DDL as described earlier in this chapter.

The following sections briefly describe defining metadata with SQL.

Defining and Deleting Databases

You can create a new database using the **create database** command followed by the database name:

```
QLI> create database "book_reviews.gdb"
```

To delete a database, use the SQL **drop database** command followed by the database name:

```
QLI> drop database "book_reviews.gdb"
```

Caution

When you drop a database, you *delete all metadata and data associated with the database.*

Defining, Modifying and Deleting Relations

You can use SQL statements to create, modify and delete relations from a database. You cannot define global fields using SQL. InterBase stores fields defined using the **create table** statement for use as global fields, but it renames the field, using the convention of RDB$ prepended to a system-generated number.

The **create table** statement defines a relation and all local fields:

```
QLI> create table books(
CON> title varchar (50),
CON> author varchar (20),
CON> reference_number integer)
```

Note

SQL does not support the blob or array datatypes; to define a blob field you must use **gdef**. For information on blobs and how to define them refer to the *Data Definition Guide*.

A significant difference in defining relations using the DDL **define relation** statement and the SQL **create table** statement is the user access privileges assigned by default to the relation. If you define a relation using DDL, all users of the database have access privileges for the relation. In order to limit access, you must use InterBase security (described in the *Data Definition Guide*), or use the SQL **grant** and **revoke** statements (described later in this chapter). If you use SQL, however, only you as the creator have access privileges to the table. In order to make the table available to other database users, you must use the **grant** and **revoke** statements to assign privileges.

To modify a relation, use the **alter table** statement to add or delete a field or fields from the relation:

```
QLI> alter table books add
CON> category varchar (12), -
CON> drop reference_number
```

To delete a relation, use the statement **drop table**:

```
QLI> drop table books
```

Deleting a table deletes all data stored in the table. It also deletes any associated indexes or views that refer to the table.

Defining and Deleting Indexes

The **create** and **drop** statements are supported for indexes. You can define either a unique or non-unique index for a relation. The following is an example of a non-unique index for the rivers relation in the atlas.gdb database:

```
QLI> ready atlas.gdb as atlas
QLI> create index riv on rivers (river)
```

A unique index eliminates duplicate index values. To define a unique index, add the **unique** keyword to the definition:

```
QLI> create unique index city on cities (city, state)
```

To delete an index, use the **drop index** command followed by the index name:

```
QLI> drop index riv
```

Navigating Using Indexes

Optional keywords enable you to qualify an index so sorts can be performed more efficiently. To make a descending sort more efficient, specify the **descending** option for the **define index** statement:

```
QLI> create descending index teams_1 on
CON> baseball_teams (team_name)
```

To make an ascending sort more efficient, specify the **ascending** option for the **define index** statement, or omit the option entirely:

```
QLI> create index teams_2 on
CON> baseball_teams (team_name)
```

This query produces the same result as the following query:

```
QLI> create ascending index teams_2 on
CON> baseball_teams (team_name)
```

Defining and Deleting Views

In QLI, you can NOT create views using either GDML **define view** or SQL **create view**. Views can be created only with **gdef**, embedded SQL, or DSQL. After you use **create** or **define view**, which stores the view's BLR representation, you can use that view to access data in QLI, preprocessed programs in SQL or GDML, DSQL programs, or OSRI programs.

If you create a view using **gdef**, a textual view definition is stored in addition to the BLR representation of the view. This textual definition is displayed when you use the QLI **show** command, and it allows **gdef -extract** to build a view definition in the DDL file it creates. If you extracted a view that was written in SQL, **gdef** would not be able to redefine that view and you would have to recreate it through a separate SQL operation.

To delete a view, use the **drop view** statement:

```
QLI> drop view mile_high_city
```

Views are also deleted if they refer to a relation that has been deleted.

Assigning SQL Grant and Revoke Security Privileges

Qli supports the SQL privilege statements **grant** and **revoke** for controlling user access to tables. Only the owner (that is, the creator of the table) may initially grant privileges using the **grant** and **revoke** security statements. Additionally, only the owner of a table can alter the table definition (that is, the metadata).

SQL **grant** and **revoke** statements are the only security statements you can define in **qli**. Any table defined using the SQL **define table** statement follows the SQL standard behavior of being accessible only to the table creator. This means if you create a table, you must explicitly grant access to the table in order for other users to have privileges for the table.

The following sections describe how to:

- Grant privileges
- Revoke privileges
- Secure a database

The examples for the following sections are all based on the expenses.gdb database, defined in previous sections. You must create the relations in order to own them and have the authority to assign privileges.

Granting Privileges

To grant a user access to a table, you enter a **grant** statement specifying the:

- Privilege
- Table name
- User name or names

For example, the following statement grants read-only privileges for the TRIPS relation to two users:

```
QLI> grant select on trips to juliec, danag
```

Privileges you can grant are listed in the following table.

Table 6-3. Grant Privileges

Privilege	**Access provided**
All	Read, write, update, and delete rows of a table.
Select	Read rows of a table.
Delete	Delete rows of a table.
Insert	Put new rows in a table.
Update	Modify columns of an existing row of a table.

To grant a privilege to all users of the database, you can specify **public** rather than list all user names. For example:

```
QLI> grant all on stops to public
```

You can assign more than one privilege in a statement. For example, to provide read and write privileges to all users, type:

```
QLI> grant select, insert on stops to public
```

Finally, you can use a grant statement to assign grant authority to users. For example, you may want all supervisor-level users to grant access privileges at their discretion to their department members. Assigning a **with grant option** privilege to a user enables the user to grant privileges. The following example illustrates the **with grant option** privilege:

```
QLI> grant select on trips to charles with grant option
```

Charles is now authorized to grant select privileges to other users. A grantor may only grant the privileges for which he or she is authorized. Thus, Charles can only grant select privileges to other users.

Displaying Privileges

When you assign privileges to a relation using grant and revoke, InterBase creates an underlying InterBase security class. The name of the security class is SQL$*relation-name*. You can view the security class using the **show** command. For example, to view the security class on the STATES example:

```
QLI> show states
      STATE  varying text, length 4
      STATE_NAMEvarying text, length 25
      AREA   long binary
      STATEHOODdate
      CAPITALvarying text, length 25
  Security class sql$states

QLI> show security_class sql$states
      SQL$STATES
          ACL version 1
              person: LESLIE, privileges: (PCDWR)
              person: CHARLES, privileges: (R)
              person: DANAG, privileges: (R)
              person: JULIEC, privileges: (R)
QLI>
```

For more information on interpreting the security class listing, refer to the chapter on security in the *Data Definition Guide*.

Revoking Privileges

You can remove privileges from a user by using the **revoke** statement. For example, to revoke all user access to the states table, you enter the statement:

```
QLI> revoke all on trips from public
```

Revoking privileges can have a cascading effect, that is, one action causes other actions. For example, suppose that Charles, having been assigned granting privileges, grants select authority to Diane. If you revoke Charles' privileges, Diane's privileges are also revoked. For example:

```
QLI> revoke select on trips from charles
```

To further complicate the example, if Diane has also been granted select privileges from another source, she retains those privileges when Charles loses his. For example, suppose Charles, when he had the authority to do so, granted Diane select and insert privileges for the trips table:

```
QLI> grant select, insert on trips to diane
```

Now Elizabeth grants Diane select and update privileges:

```
QLI> grant select, update on trips to diane
```

Diane now possesses the authority to read, write, and update data. Now, however, Charles changes jobs and his privileges are revoked:

```
QLI> revoke all on trips from charles
```

Charles' authority to grant privileges is also revoked, thus the select and insert privileges should be taken from Diane. However, since she is authorized to read and update data through Elizabeth, Diane only loses her insert privilege.

For more information on the syntax for the **grant** and **revoke** statements, refer to the *Qli Reference*.

Securing a Database

InterBase provides another security model that enables you to secure databases, relations, views, and fields. This security model, which uses security classes to control user access to database structures, is fully described in the *Data Definition Guide*.

Caution

Once you create or secure a table using an SQL statement, you cannot use InterBase security classes on that table.

You cannot define security classes through **qli**. You must define them using **gdef**. For information on defining security classes, refer to the *Data Definition Guide*.

Occasionally, the two security systems can collide and cause some confusion. For information on integrating the two approaches, refer to the *Data Definition Guide*.

Interactive Data Definition and Transactions

Unlike direct updates to the system relations, changes you make to the metadata with **qli** using its metadata manipulation commands are reflected immediately, both in the database and in **qli**'s internal image of the database. This happens because **qli**'s metadata commands are executed under a special metadata transaction that commits immediately.

A side-effect of the special transaction is your regular transaction may not see changes you made to system relations with **qli**'s metadata commands. Committing your regular transaction "resynchronizes" your view of the system relations. Consider the following example:

```
QLI> ready expenses.gdb
QLI> define index idx1 for trips unique
CON> trip_id
QLI> show indexes
Database "expenses.gdb" readied as QLI_0
    STOPS
        No indices defined
    TRIPS
        Index IDX1 (unique)
            TRIP_ID
QLI> print rdb$index_name of rdb$indices with rdb$index_name =
CON> 'IDX1'
QLI> commit
QLI> print rdb$index_name of rdb$indices with rdb$index_name =
CON> 'IDX1'

           RDB$INDEX
             NAME
==============================

IDX1
QLI>
```

Before the regular transaction was committed, the new index is visible with the **show indexes** command. This command also operates under the special transaction, but the **print** statement is controlled by the regular transaction. However, the index does exist and is used to optimize queries. Once the regular transaction is committed and restarted, the new index is visible to it. Similarly, when you define a new relation, you can immediately store records into it and retrieve records from it, but you are not able to see it by looking directly at RDB$RELATIONS system relation until you commit the regular transaction.

Because interactive data definition in **qli** is executed through a special transaction that is committed implicitly on completion, you *cannot* rollback an interactive metadata command. For the **define** and **modify** commands, you can undo the action by dropping or re-modifying the relation, field, or index. However, if you drop a relation, or drop a field from a relation, all data in the relation or field is lost.

For More Information

For more information about:

- Data definition and modification using **gdef**, see the *Data Definition Guide*.
- Defining metadata using SQL statements, refer to the *Data Definition Guide*.
- Syntax for the **grant** and **revoke** statements, refer to the *Qli Reference*.
- Securing a database and its structures, refer to the *Data Definition Guide*.

Chapter 7
Using Procedures

This chapter describes the creation, use, and deletion of stored procedures in **qli**.

Overview

As you start to use **qli** on a regular basis, you might find you frequently execute the same sequences of commands or statements. InterBase allows you to define and store a sequence of commands or statements as a *procedure*. A procedure can be an entire **qli** statement, several or many statements, a **repeat** statement, or invocations of other procedures. Invoking a procedure causes it to be expanded and executed as if it had been typed directly to **qli**.

Procedures let you:

- Eliminate the repetitive typing of commonly executed commands and of sequences of commands.
- Provide end-users with an interface requiring only their response to prompts, thus shielding them from data manipulation languages.

- Develop complex applications very quickly.

Procedures can utilize all the capabilities of **qli** including prompting, forms, complex datatype handling, conditional branching, report writing, and so on.

Procedures are generally associated with a particular database, and are by default stored in the database itself. Therefore, when you ready a database, you have access to all the procedures defined for that database. Type the following command to list the procedures in the atlas database:

```
QLI> show procedures
Procedures in database "/usr/castor/atlas.gdb (QLI_0):
   DISPLAY CITIES
   FIX
   GEO_CITIES
   HAVING_EXAMPLE
   MAJOR_CITIES
       ⇓
QLI>
```

Defining Procedures

You can define a procedure in **qli** with the **define procedure** command or by invoking the default editor followed by a procedure name. Using the editor is the easier of the two approaches.

Using Your Default Editor

To define a procedure using your default editor, type **edit** and a name for the procedure at the QLI> prompt.

```
QLI> edit store_ski
```

Your default editor pops up at this point. Enter a procedure definition. For example, the following procedure prompts you for the information neccessary to store a new ski area:

```
repeat *."number of repetitions"
store ski_areas using
begin
name = *."name of paradise"
city = *."the municipality"
state = *."the state code"
type = *."nordic (N), alpine (A), or both (B)"
end
```

Exit from the editor.

The asterisk in the assignment statements calls for **qli** to prompt you for a value. The period and quoted string following the asterisk is the text with which you want to be prompted.

Qli supplies inserts the word Enter before the text string. For example, the assignment **."number of repetitions"* becomes `Enter number of repetitions:` when **qli** executes the assignment.

Using the Define Procedure Command

You can also define a procedure with the **define procedure** command. Enter the definition at the QLI> prompt, as follows:

GDML

```
QLI>  define procedure store_ski_2
CON>  store ski_areas using begin
CON>  name = *.'name of paradise'
CON>  city = *.'the municipality'
CON>  state = *.'the state code'
CON>  type = *.'nordic (N), alpine (A), or both (B)'
CON>  end
CON>  end_procedure
```

SQL

```
QLI>  define procedure store_ski_2
CON>  insert into ski_areas
CON>  (name, city, state, type)
CON>  values (*.'name of paradise',
CON>  *.'the municipality', *.'the state code'
CON>  *.'nordic (N), alpine (A), or both (B)')
CON>  end_procedure
```

Defining Generic Procedures

If you want to define generic procedures for your applications, and not specific to any particular database, you can define procedures in their "own" database, and ready the database of procedures whenever you work with a production database. This is useful if you define procedures for your own use, that may not be appropriate to store in a central database.

For example, you can create procedures that use the **spawn** or **shell** commands to exit temporarily from **qli**, perform some action at the shell or command language level, and return you to the QLI> prompt.

Examples of procedures include:

- Checking the operating system's time
- Listing files in a directory
- Sending a mail message

A more complex example would be using a procedure to update InterBase metadata, such as the position of a field in a relation.

The best way to include generic procedures in all database access is to use **qli**'s startup command file to ready the procedure database automatically when you invoke **qli**. Chapter 1 discusses the use of startup command files.

Running a Procedure

To execute a procedure, type a colon (:) followed by the procedure name.

For example, to run the STORE_SKI procedure, type a colon followed by STORE_SKI at the QLI> prompt. The STORE_SKI procedure prompts the user to enter information:

```
QLI> :store_ski
Enter number of repetitions: 1
Enter name of paradise: Wilderness Xtrack
Enter the municipality: Colebrook
Enter the state code: NH
Enter nordic (N), alpine (A), or both (B): N
QLI>
```

The procedure called TRIBUTARIES in the atlas database prints records from a reflexive join of the RIVERS relation. To run the TRIBUTARIES procedure, type:

```
QLI> :tributaries

BIG RIVER                          TRIBUTARY
============================================================

Arkansas                           Canadian
Arkansas                           Neosho
Arkansas                           Cimarron
Canadian                           North Canadian
Colorado                           Gila
Colorado                           Green
  ⇓
QLI>
```

Modifying Procedures

To modify a procedure, type **edit** and the procedure name at the **qli** prompt. For example:

```
QLI> edit store_ski
```

An editing session is started. Change the procedure as you see fit. For example, change the STATE assignment to read:

```
state = *.'2-character state code'.
```

Exit from the editor.

Renaming Procedures

To rename a procedure, use the **rename procedure** command. **Qli** starts a special transaction to rename the procedure. For example:

```
QLI> rename procedure store_ski to new_ski_area
```

Procedure names are stored by InterBase as all uppercase characters, regardless of how you enter them. Therefore, if you use the **for**, **modify**, **print** statements or their SQL counterparts to change a procedure name and do not have any success, check the spelling and be sure you used all uppercase characters in the record selection expression.

Copying Procedures

To create a second copy of a procedure, use the **copy procedure** command. **Qli** starts a special transaction to copy the procedure. For example:

```
QLI> copy procedure new_ski_area to store_ski
```

You can also copy a procedure from one database to another using the **copy procedure** command. To do so, you specify the database handle as part of the procedure name. For example:

```
QLI> copy procedure atlas.new_ski_area to map.store_ski
```

Deleting Procedures

To delete a procedure, use the **delete procedure** command. **Qli** starts a special transaction to delete the procedure. For example:

```
QLI> delete procedure new_ski_area
```

Storing and Identifying Procedures

Qli stores the procedure during a special transaction it creates for that purpose. Once it stores the procedure, it commits the special transaction. Meanwhile, the default transaction controlling the data continues unaffected by the completion of the special transaction. Thus, when you modify a procedure the previous version is replaced.

Procedures are associated with a specific database, so in multi-database applications you must take certain actions to be sure you use the correct procedure. By default, **qli** looks in the most recently used database for the invoked procedure. For example, consider the following sequence of commands:

```
QLI> ready atlas.gdb as atlas
QLI> ready map.gdb as map
QLI> :store_ski
```

This sequence causes **qli** to execute the procedure STORE_SKI from the map database. If there is no such procedure in the map database, **qli** next looks in the atlas database. In this example, if the procedure is defined in both databases and you want to use the copy from the atlas database, qualify the procedure name with the database handle. For example:

```
QLI> :atlas.store_ski
```

The following table summarizes the use of database handles with the procedure commands:

Table 7-1. Multiple-Database Access and Procedures

Operation	Result
:Procedure-name	Invokes the specified procedure. Searches through the databases from the most recently readied dtatbase back to the first readied databaselooking for the procedure name. If the procedure name is qualified with a database handle, it immediately looks in the specified database.
define procedure	Opens or creates the specified procedure in the most recently opened database, unless the procedure name was qualified with a database handle.
edit	Opens or creates the specified procedure in the most recently opened database, unless the procedure name was qualified with a datbase handle.

Table 7-1. Multiple-Database Access and Procedures continued

Operation	Result
delete procedure	Deletes the named procedure from the most recently opened database or reports an error if the procedure was not found,
rename procedure	Renames the procedure in the most recently opened database and reports an error if not found,unless the procedure name was qualified with a datbase handle. Both the old and new procedures must be in the same database.
copy procedure	Copies a procedure within the most recently opened database or reports an error if the old name does not exist or the new name already exists. Old, new, or both databases can be qualified.
show *procedure-name*	Displays all procedures in all readied databases with the specified procedure name. If the procedure name is qualified with a database handle, it shows only the procedure in the specified database.
show procedures	Lists procedure names for all readied databases

Performing Application Tasks by Procedure

Procedures in **qli** can take the place of many tasks usually performed by application programs. The following sections illustrate how tasks can be executed using procedures rather than application code.

Conditional Branching in Procedures

The functions described in the following procedure would be useful in a wide variety of applications, including sales, inventory, and manufacturing reports. Note the following information:

- Nested **if** statements cause the procedure to produce an appropriate message based on the data. In the following example, different messages are displayed for zero, one, or more small cities.
- For each small city, another set of nested **if** statements print different messages for small, very small, very small indeed, and totally negligible cities.
- Multiple statements are wrapped in a begin-end block so they act together.
- Local variables (that is, variables defined within a begin-end block) are used to represent values in if-else statements. For example, the *counter* variable defined in the following procedure, holds a value for the number of cities with a population that exceeds 100,000 in a state.
- The procedure includes formatting options to the **print** statement, such as **skip** and **col**, that enhance the display of output. Formatting terms are described in detail in Chapter 8, *Writing Reports*.

The following procedure searches through states relation looking for cities with fewer than a million residents. The following **show** command displays the full script of the procedure:

```
QLI> show urban_categories
Procedure URBAN_CATEGORIES in database "/usr/castor/databases/
atlas.gdb" (QLI_0)

for states sorted by state_name
 begin
 declare counter long
 counter = count of cities over state -
  with population > 100000
 if counter = 0
  print skip, state_name | ' has no big cities.' else
if counter = 1
```

```
  print skip, col 1, state_name | ' has one biggish city.' else
print skip, col 1, state_name | ' has several biggish cities.
For example:'
  for cities with population > 100000 and
      state = states.state sorted by population
      if population between 100000 and 500000 then
         print col 5, city | ' is no one-horse town' |
             ' (population = ' |  population | ').' else
      if population between 500001 and 1000000 then
         print col 5, city | ' is a large population center' |
             ' (population = ' |  population | ').' else
      if population between 1000001 and 5000000 then
         print col 5, city | ' deserves a major-league team' |
             ' (population = ' |  population | ').' else
         print col 5, city | ' is a world-class metropolis' |
             ' (population = ' |  population | ').'
end
```

Note

An **else** statement cannot begin a line unless the line that precedes it ends in a concatenation hyphen, signaling to **qli** the expression is not complete.

Running this procedure results in the following display:

```
QLI> :urban_categories

Alabama has several biggish cities.  For example:
    Montgomery is no one-horse town (population = 177857).
    Birmingham is no one-horse town (population = 284413).

Alaska has no big cities.

Arizona has one biggish city.
    Phoenix is a large population center (population = 789704).

Arkansas has one biggish city.
    Little Rock is no one-horse town (population = 158461).

California has several biggish cities.  For example:
    Fresno is no one-horse town (population = 218202).

⇓
QLI>
```

Prompting for Values in Procedures

The following procedure uses an interactive query to clean up cities that were stored without a population. At each city, the user is prompted to say whether the city should be modified, deleted, or ignored.

As in the previous example, multiple statements are wrapped in a begin-end block. You must use a begin-end block if there are more than one statements as the object of the if-else statement. For more information on begin-end blocks, see the *Qli Reference*.

```
QLI> show clean_up_cities
Procedure CLEAN_UP_CITIES in database "/usr/castor/databases/
atlas.gdb" (QLI_0)

begin
  declare upd char [1];
  print skip, col 1, "Cleanup our cities."
  print skip, col 1,
      "Type D to delete the city, M to change its
         population,", skip,
      "or L to leave it alone"
  for cities with population missing
  begin
      print skip
      upd = "X"
      repeat 50
      begin
          if upd not in ("D", "M", "L") then
          begin
              print col 1,
                  "What do you want to do to " | city | " " |
                    state | "?"
              upd = *."D[elete] / M[odify] / L[eave it be]"
              if upd = "D"
              begin
                  print col 1, city | " " | state | " is gone"
                  erase
              end else
              if upd = "M"
                  modify using
                  population = *."new value for population" else
              if upd = "L"
                  print col 1, "ok by me" else
              print "Bad guess. Try again"
```

```
        end
      end
    end
  end
```

Running this procedure results in the following display:

```
QLI> :clean_up_cities

Cleanup our cities.

Type D to delete the city, M to change its population,
or L to leave it alone

What do you want to do to Dover DE?
Enter D[elete] / M[odify] / L[eave it be]: X
        Bad guess. Try again
What do you want to do to Dover DE?
Enter D[elete] / M[odify] / L[eave it be]: M
Enter new value for population: 53000

What do you want to do to Tallahassee FL?
Enter D[elete] / M[odify] / L[eave it be]: D
Tallahassee FL is gone

What do you want to do to Boise ID?
Enter D[elete] / M[odify] / L[eave it be]: L
ok by me

What do you want to do to Springfield IL?
Enter D[elete] / M[odify] / L[eave it be]: 1
        Bad guess. Try again
What do you want to do to Springfield IL?

⇓
QLI>
```

Using Operating System Command Procedures

Qli supports *external command procedures*. An external command procedure is a sequence of **qli** commands and statements stored by the host operating system's file system. For example, you can create procedures that ready and finish databases.

You must store external command procedures as files rather than as **qli** procedures. To do so, create a file of **qli** operations using your system editor. If you are already in **qli**, use the **spawn** or **shell** commands to escape from **qli**, create a file of **qli** operations, and then logout of the subprocess or shell when you are finished.

For example, the following statements show how to use the shell command on a Sun to create an external procedure that readies the atlas.gdb database as atlas. Type the shell command to escape to the operating system level:

```
QLI> shell
$
```

Open an edit buffer and type:

```
ready atlas.gdb as atlas
```

Exit the editor, saving the file as *ready_pro*.

```
$ Ctrl-D (Or the end of file key for your system)
QLI>
```

Once you are back in **qli**, you can invoke the external procedure with an @-sign immediately followed by the name of the file containing the procedure. If the external file is located somewhere other than your current default or working directory, specify the pathname.

For example, to invoke the ready_atlas procedure:

```
QLI> @ready_pro
QLI>
```

If you already have a database readied with the handle atlas, you should type **finish atlas** before executing ready_pro.

To see if the procedure worked, use the **show databases** command:

```
QLI> show databases
Database atlas.gdb is readied as ATLAS
⇓
```

Note

External command files are not under database control, so they are not backed up with the rest of the database.

For More Information

For more information about operations on procedures, see the *Qli Reference* for the syntax of:

- **copy procedure**
- **define procedure**
- **edit procedure**
- **delete procedure**
- **rename procedure**
- **declare** variable
- **begin-end** block
- **if-else** statement
- **print** statement
- field attributes
- assignment

Chapter 8
Writing Reports

This chapter describes the use of **qli**'s report writer. The report writer allows you to format the output of database queries.

Overview

Qli provides an integrated report writer that builds on your knowledge of qli's data manipulation languages and lets you quickly develop a wide range of reports.

Components of a Report

At its simplest, a report consists of:

1. The **report** command
2. A record selection expression
3. A print list

The following simple report displays all records in the SKI_AREAS relation:

```
QLI> report ski_areas sorted by state
CON> print city, state, name
CON> end_report
          CITY             STATE         NAME
========================== ===== ====================
Carlisle                    MA   Great Farm
Groton                      MA   Birchwood Acres
Waterville Valley           NH   Waterville Valley
New Ipswich                 NH   Windblown
Mt. Washington              NH   Bretton Woods
Dixville Notch              NH   Wilderness
Stowe                       VT   Epson Hills
Stowe                       VT   Mt. Mansfield
Stowe                       VT   Trapp Family Lodge
QLI>
```

Generating a Report

To generate a report, you use the **report** command to invoke the report writer. The **report** command requires you to specify a record selection expression as the source for the report. There are a variety of options for the **report** command providing you with formatting control over the RSE output. For example, you can specify:

- The length and width of the page
- A header to print at the beginning of the report
- A header and footer to print on every page
- Control groups (for example, cities by state)
- Aggregate values for control groups

You can direct the reports to your screen, through an external command utility (for example, **print** or **lpr**), or to a file.

Using Procedures to Run Reports

When you first write a report, you may find that it is not quite what you want. You decide to re-type the report and add the features you want, make changes to the print list, or change the record selection expression. For this reason, you might want to incorporate the report in a procedure. Having the text of the report in a procedure means that you can go back and edit the statements to do what you want. It also means that you can save the report for future use.

Defining a Report Procedure

If you have typed nothing since the abov **report** command, type **edit** to get the last command. The editing buffer contains the entire three-line report. Preface it with a **define procedure** statement and terminate it with an **end_procedure** tag, thus yielding the following:

```
define procedure ski_list
report ski_areas sorted by state
print city, state, name
end_report
end_procedure
```

Now you can run the procedure SKI_LIST and have it produce the simple report listing ski areas. Or you can start customizing it to be more informative. Edit the procedure and add a format option that prints a specified column header and inserts a new line after the field specified. The specified field must be a sort field in the RSE. The **at top of** control break is specified using the following form:

at top of	*sort field*	**print**	*database-fields*

The specified database fields are printed on one line; any print statement that follows prints the output beneath them. In the following example, the state names are isolated from the rest of the output and the control breaks are nested.

```
QLI> edit ski_list
```

The editor starts up with contents of SKI_LIST. Change it to read as follows:

```
report ski_areas sorted by state
      at top of state print state
      print city, type, name
end_report
```

Exit from the editor and run the report. To execute a report procedure, type a colon followed by the procedure name, as follows:

```
QLI> :ski_list

STATE  CITY                 TYPE  NAME
====== ==================== ===== ====================
MA
       Carlisle              N     Great farm
       Groton                N     Birchwood Acres
NH
       Mt. Washington       B     Bretton Woods
       Waterville Valley    B     Waterville Valley
       Dixville Notch       B     Wilderness
       New Ipswich          N     Windblown
VT
       Stowe                B     Mt. Mansfield
       Stowe                N     Epson Hills
       Stowe                N     Trapp Family Lodge
```

Defining More Complicated Reports

More complicated reports might include control breaks for calculated aggregate values. For example, you might want to display cities listed by state and calculate the average altitude or population of those cities. The following report joins two relations, STATES and CITIES, prints cities whose population is not missing, and displays the count of cities in each state. To save typing, the MAJOR_CITIES report is included in the atlas database. To view the report, type:

```
QLI> show major_cities

Procedure MAJOR_CITIES in database "Atlas.gdb" (QLI_0)
report cities cross states over state -
          with population not missing -
  sorted by state, desc population
  set report_name =
  "M a j o r" /  "C i t i e s" / "(b y   s t a t e)"
  at top of state print state_name
  print city, population using z,zzz,zz9
  at bottom of state print col 10,
  count | " cities in " | state_name, skip
end_report
QLI>
```

The **report_name** expression centers the specified name and prints the name on three lines because of the slash separating MAJOR from CITIES, and CITIES from BY

STATE. The **at bottom of** expression inserts a control break before the output of the **count** statement.

Run the procedure:

```
QLI> :major_cities
                    M a j o r
                  C i t i e s
               (b y   s t a t e)
          STATE
          NAME                        CITY             POPULATION
========================== ========================== ==========
Alaska
                           Juneau                          7,000
          1 cities in Alaska
Alabama
                           Birmingham                    284,413
                           Montgomery                    177,857
          2 cities in Alabama
Arkansas
                           Little Rock                   158,461
          1 cities in Arkansas
Arizona
                           Phoenix                       789,704
          1 cities in Arizona
California
                           Los Angeles                 2,966,850
                           San Diego                     875,538
                           San Francisco                 678,974
                           Sacramento                    275,741
                           Fresno                        218,202
          5 cities in California
          ⇓
QLI>
```

Qli provides the **running count** and **running total** value expressions. You may find these useful in reports. Edit the MAJOR_CITIES procedure to include a running city and population count:

```
QLI> edit major_cities
```

The editor starts up. Add running expressions to the **print** statement.

```
report cities cross states over state -
  with population not missing -
  sorted by state, desc population
```

```
  set report_name =
      "M a j o r" / "C i t i e s" / "(b y   s t a t e)"
  at top of state print state_name
  print city, population using z,zzz,zz9,
  running count ('RUNNING'/'CITY'/'COUNT'),
  running total population ('RUNNING'/'TOTAL'/'POPULATION')
  at bottom of state print col 10,
  count | " cities in " | state_name, skip
end_report
```

Note

If you run the modified procedure on a terminal, set the screen to be 132 rather than 80 columns. Otherwise, the running counts run off the edge of the screen. In addition, you may need to use the **set columns** command to specify the width of your window to **qli**.

For More Information

For more information about the report writer, see the entries in the *Qli Reference* for:

- **report** command
- **print** statement
- *value-expression*

For more information about procedures, see Chapter 7.

Chapter 9
Using Forms

This chapter describes the use of forms in **qli**.

Overview

The forms utility provides you with the ability to create an end-user interface to your application. Users can use forms to enter data to and retrieve data from a database.

A *form*, as the name implies, is a graphic template you can format to suit your needs for displaying and collecting information. Forms are generally based on a relation in your database.

You do not create forms in **qli**. However, **qli** can generate default forms based on relations. Creating forms using the forms editor, **fred**, is fully discussed in the *Forms Guide*. Qli allows you to edit forms and save your changes to the database.

Accessing Data through Forms

There are two ways that you can access forms in **qli**:

- The first method is to issue a statement that invokes forms by default each time you invoke a relation.
- The second way is to invoke a form explicitly, for particular cases where you want a form displayed.

Invoking Forms Automatically

The **set form** command causes **qli** to display a form or each data manipulation operation. To automatically invoke forms, use the **set** command with the **form** option. For example:

```
QLI> set form
QLI>
```

If you issue a GDML command to **print**, **store**, or **modify** records that does not provide a list of fields, **qli** searches the database for a form with the same name as the relation involved in the data manipulation. For example to view data about the Colorado river, type:

```
QLI> for rivers with river = "Colorado"
CON> print
```

```
        RIVER       Colorado
       SOURCE       CO
     OUTFLOW        Gulf of California, Mexico
       LENGTH       1360

 <ENTER> to continue, <PF1> to stop
```

Note

The instructions displayed at the bottom of the form are machine-specific. For example, the <ENTER> and <PF1> key commands apply to Apollo users. If you are using a Sun, the instructions tell you to:

```
<ENTER> or <R15> to continue, <R1> to stop
```

This manual uses the convention for the Apollo.

In the above example, **qli** painted a default form for the RIVERS relation, using the data specified in the record selection expression. The following rules outline how **qli** interprets a request for a form display:

- If a form exists for the relation, **qli** uses that form to paint the screen. The form may have been defined using **fred**.
- If no form exists for the relation, **qli** generates a default form. The form is the same one that **fred** generates as a default form.
- If the query involves more than one relation, **qli** uses the preceding two rules to find a form for the last relation referenced in the query. For example, if you join STATES and BASEBALL_TEAMS, in that order, **qli** looks for a form associated with BASEBALL_TEAMS.

To turn off the automatic form display, type the following command:

```
QLI> set no form
QLI>
```

Invoking Forms Explicitly

You may want to control which operations use a form or when they use a form. **Qli** provides the **using form** clause on the **print**, **store**, and **modify** statements to let you specify when to use forms on those operations, and a **for form** statement for structuring the interaction of forms with other operations.

For example, to display a form for the RIVERS relation, type:

```
QLI> for rivers with river = "Colorado"
CON> print using form
```

```
      RIVER        Colorado
     SOURCE        CO
    OUTFLOW        Gulf of California, Mexico
     LENGTH        1360

<ENTER> to continue, <PF1> to stop
```

Managing the Form Display

The previous example showed the **qli** default form. You cannot edit or restrict the form display; it includes all fields for the relation. If you want to limit or restrict the fields that are displayed use one of these options:

- Define a view with only those fields you want. See the chapter on defining meta-data for information on how to define a view.
- Modify the form or create a new form that uses only those fields you want. See the *Forms Guide* for more information.
- Associate a security class with the field. See the *Data Definition Guide* for more information.

The atlas.gdb database includes some pre-defined forms. To see what forms exist in your database, use the **show** command with the **forms** option, as follows:

```
QLI> show forms
Forms in database atlas
   CITY_POPULATIONS
   BASEBALL_TOWNS
   STATE_CITY
   CITIES
   STATES
```

The following example displays all records in the CITIES relation, using the predefined form named CITIES:

```
QLI> print cities using form
```

```
City:            Juneau
State:           AK
Population:             7000
Altitude:        99999999999
Latitude:        58 18N
Longitude:       134 25W

  <ENTER> to continue, <PF1> to stop
```

To view the next city, press the "ENTER" key. To get back to the QLI> prompt, press the "PF1"key. When you do so, **qli** returns the following message:

```
** QLI error: execution terminated by signal **
```

Note that the form displays the value "9999999999" for the "Altitude" field on the form. This display indicates that the altitude for that field is missing. **Qli** uses the following conventions for missing data, or when you are entering new data:

- "X" means alphanumeric data
- "9" means fixed integer data, with or without scale
- "F" means single or double floating data
- "D" means date data

You can also specify a form by name. For example, the following **print** statement requests the use of the form named CITIES to print records from the CITIES relation:

```
QLI> print cities using form cities
```

Specifying a form name, rather than relying on the default form, is especially important when you are working with multi-relation queries. An earlier section mentioned a query involving STATES and BASEBALL_TEAMS, such that the default form approach would lead **qli** to use a form associated with BASEBALL_TEAMS, the last referenced relation. Because the default form for BASEBALL_TEAMS may not refer to fields from the STATES relation, fields from STATES do not show up on the form.

The sample database has a form called BASEBALL_TOWNS that includes several fields from the BASEBALL_TEAMS relation and the STATE_NAME field from the STATES relation.

To view records through this form, type the following query:

```
QLI> for states cross baseball_teams over state
CON> print using form baseball_towns
```

Qli puts up the following form:

```
Team name:    Yankees          League:                A
City:         New York
State:        New York

Home stadium: Yankee Stadium
Seating:      57545       Left field:        312
Surface:      N         Center field:        417
                         Right field:        310
<ENTER> to continue, <PF1> to stop
```

Press the ENTER key to see more baseball teams or the PF1 key to return to the QLI> prompt.

Storing Data Using Forms

You can use a form to enter data into the database. For example, the following store statement invokes the CITIES form. Highlighted fields are editable. Select a field using the cursor keys and type in a new value. Values entered into a form are treated exactly as values entered in a **store** statement from the QLI prompt. They are not entered into the database until the current transaction is committed.

```
QLI> store cities using form cities
```

```
City:                XXXXXXXXXXXXXXXXXXXXXX
State:               XX
Population:          99999999999
Altitude:            99999999999
Altitude:            XXXXXXX
Longitude:           XXXXXXX

  <ENTER> to continue, <PF1> to stop
```

Qli leaves the cursor in the input area of the first field. To store a value:

1. Enter a value for the CITY field
2. Press the Return or Tab key to advance to the next field, or use the cursor keys to move in the direction you want to go. **Qli** moves you to the beginning of the input area you choose. If you want to skip a field, just press the Return key.
3. To finish storing records, press the Enter key.
4. To discard the record on the screen and exit the form, press the PF1 key.

To store multiple records, use the **repeat** statement with the **store** statement. For example:

```
QLI> repeat 2 store ski_areas using form
```

```
Name:      XXXXXXXXXXXXXXXXXXX
CITY:      XXXXXXXXXXXXXXXXXXX
STATE:     XX
TYPE:      X

<ENTER> to continue, <PF1> to stop
```

In this case, pressing the Enter key enables you to enter the second record, and pressing the PF1 key exits the form.

Modifying Data with Forms

Modifying data using a form is similar to storing data using a form. For example, if you want to modify data for the city of Juneau, type the following:

```
QLI> for cities with city = 'Juneau'
CON> modify using form
```

```
City:          Juneau
State:         AK
Population:    7000
Altitude:      99999999999
Latitude:      58 18N
Longitude:     134 25W

<ENTER> to continue, <PF1> to stop
```

To change the value of a field, use the cursor keys or the Return key to get to the field you want to change. Once you get to the field you want to change, type the new value, press the Return key, and **qli** changes the value. Type "Pf1"to exit the form and return to the QLI> prompt.

Note

Modifications made to data are not written to the database until the current transaction is committed.

Using Forms in For Loops

The **for_form** statement provides a way of staging input to a form, so that **qli** waits for input and redisplays the form with the entered data when you press the Pf1 key.

The **for_form** statement is similar to the **for** statement in that it includes a **qli** statement as a substatement. The substatement is usually a **begin-end** block containing one or more **accept** statements or form field assignments. See the entry in the *Qli Reference* for the **for_form** statement for the syntax of the **accept** statement.

The following example displays a form to accept the input of a state code, and displays a form to display the state name, area, and 1980 census data:

```
QLI> for_form f in baseball_towns
CON> begin
CON> accept ("Enter state code, then <f1>") state_name
CON> for b in baseball_teams cross s in states over
CON>      state sorted by state, team_name with s.state_name
CON>      st f.state_name
CON>  begin
CON>      f.team_name = b.team_name
CON>      f.league = b.league
CON>      f.home_stadium = b.home_stadium
CON>      f.seating = b.seating
CON>      f.surface = b.surface
CON>      f.left_field = b.left_field
CON>      f.center_field = b.center_field
CON>      f.right_field = b.right_field
CON>      accept ("Press <f1> to continue")
CON>      end
CON> end
QLI>
```

Editing Forms

You can edit forms in **qli** using the **edit form** statement followed by a form name. The **edit form** statement invokes the form specified and provides all of the editing features of **fred**, the forms editor.

For example, the following invokes the form editor for the form based on the RIVERS relation:

```
QLI> edit form rivers
```

```
    RIVER   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

   SOURCE   XXXX

  OUTFLOW   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

   LENGTH   9999999999


  EDIT:  Move cursor to object and hit <ENTER>
```

For details on using the forms editor, refer to the documentation for **fred**, the forms editor, in the *Forms Guide*.

Saving a Form

After editing a form, you can save the new version by committing the current transaction:

```
QLI> commit
```

Otherwise, roll back your changes to revert to the last saved version of the form.

For More Information

For more information about the statements discussed in this chapter, see the entries in the *Qli Reference* for:

- **set form** statement
- **store** statement
- **modify** statement
- **print** statement
- **for form** statement

For information on editing a form, refer to the *Forms Guide*.

Chapter 10
Understanding Transactions

This chapter describes **qli** transactions and how to control them.

Overview

In any programming environment, a related set of changes you make to the database should be made in their entirety or not at all. For example, if your program updates your employee database to reflect an across-the-board wage increase of 10 percent, you would prefer to see the program run to completion. Barring that, you would like to know how far the program got before it stopped, or better yet, you would like to roll back any changes that were made and try again later.

Controlling Database Activity with Transactions

Database systems deal with this problem of controlling and monitoring database activity by providing *transactions*. Transactions are bounded sets of statements that read from or write to a database. All changes made to the database in a transaction unit exe-

cute in their entirety, or none of them do. Therefore, if your program terminates before it finishes performing the complete set of related changes to the database, the database is restored to its state before the transaction started.

Transactions are also used to provide each user with a stable view of the data in the database. While working within a transaction, each user sees his own private, consistent view of data, even though changes may be occurring. When you modify or erase a record, the InterBase access method keeps the old version in addition to the new version you just created. Until the transaction is finished, you are shielded from any changes any other user may have made to the database. Thus every user works with their own virtual "snapshot" of the database and no user sees uncommitted changes.

Database Consistency and Concurrency

Ideally, each transaction should be free to use a database as if it were the only one present. Realistically, there is a trade-off between isolation and the amount of time transactions spend waiting for each other. The degree of isolation is called *consistency*. Consistency means the database remains completely stable because only one user can access a piece of data at one time. The degree to which other transactions can run simultaneously is called *concurrency*. While concurrency is the preferred behavior for transaction control, there may be cases when you want serialized transactions.

Database Consistency

The traditional way to guarantee complete consistency in a database is to lock records or whole relations whenever someone reads or writes to them. Therefore, whoever accesses a record is isolated from other people who might want to read or write the record at the same time

The unfortunate side-effect of locking records or relations to provide serializability is users exclude each other from entire relations. Drawbacks of this approach include:

- Transactions with no concern for up-to-the-minute data changes have to wait for updating transactions to finish. For example, suppose you have a read-only transaction whose function is to report on the state of a database at some point in time. Changes made after the transaction starts are of little consequence, yet the read-only transaction has to wait until the writing transaction releases its locks.
- Two transactions with conflicting requirements become deadlocked, each waiting for the other to unlock the record or table. The only solution is for one transaction to stop trying to make the change and restart.

Database Concurrency

A concurrency transaction provides complete consistency for read-only transactions without requiring readers to wait for updates or risk deadlock. Because updates create new versions of records, the older versions can always be made available to concurrent readers. Readers always see the record version that is consistent with the state of the database as it was at the start of their transaction. They never have to wait for updates to commit or rollback.

Unless you try to write a record that has been changed by another concurrent transaction, updates happen instantaneously. However, a transaction is not allowed to change a record if it cannot see the most recent version.

Therefore, a transaction can always read a record, but may get an "update conflict" error when it tries to modify or erase it. This error indicates that some concurrent transaction changed the record, or possibly even erased it. Because the current state of the record should influence your changes, your update request is denied. For example, suppose Charles and Diane both read the same record. The following sequence might occur

1. Charles modifies the record and commits his transaction, thereby updating the database.
2. Diane tries to modify the record but receives an error informing her that the record in question has been altered.
3. Diane must roll back her changes and read the record again.
4. The record now reflects Charles' modification and Diane is free to modify the record further.

Because Diane's new transaction starts after the newest version of the record was committed by Charles, changes she makes within her new transaction are accepted. This behavior provides great incentive for committing work frequently.

The concurrency model provides high, but not complete, consistency for read-write transactions. Interactions between several transactions over several records can create results that are not serializable. For more information on these potential problems, see the chapter on transactions in the *Programmer's Guide.*

The Qli Transaction Model

In **qli**, control over transactions is limited. **Qli** automatically starts a default transaction for each database you ready. This default transaction is a concurrency mode transaction. The transaction for each database continues until you explicitly terminate it by committing data to the database or by rolling back any changes you may have made

Consistency transactions are available in InterBase, but not from within **qli**. If you want to switch to a consistency mode of transaction control, you must do so using GDML statements in a program.

For information on transaction control in programs, refer to the chapter on transactions in the *Programmer's Guide.*

Starting and Stopping Transactions

A transaction is started each time you ready a database. The transactions continue until they are either committed or rolled back. Committing a transaction does the following:

- Updates the database, recording permanently any modifications made to the database since the transaction began. These modifications are now available to other users of the database.
- Begins a new transaction, returning a fresh "snapshot" of the database.

Rolling back a transaction does the following:

- Restores the database to its state prior to your transaction.
- Begins a new transaction, returning a fresh "snapshot" of the database.

Each of these options are described below.

Committing and Updating a Database

To commit a transaction and update your view of the database, use the **commit** command at the **qli** command level:

```
QLI> commit
QLI>
```

The following **qli** session readies a database, updates the selected records, and then ends the transaction by writing the changes to the database with the **commit** command:

GDML	`QLI>ready atlas.gdb`
	`QLI>for cities with state = "MI"`
	`CON>modify using population = population * 0.95`
	`QLI>commit`

SQL	`QLI>ready atlas.gdb`
	`QLI>update cities`
	`CON>set population = population * 0.95`
	`CON>where state = "MI`
	`QLI>commit`

Committing Without Updating a Database

There may be times when you want to write data to the disk, but you do not want to end your current transaction.

For example, suppose you are executing a query that stores 100 records for the CITIES relation. You could write a simple procedure as follows:

```
QLI> define procedure store_100_cities
CON>      begin
CON>          repeat 100 store cities
CON>      end
CON> end_procedure
```

You might not want to wait until all 100 cities are stored to commit the transaction. If you do, there is a higher risk some problem could make you abort the operation in the middle, losing the work you performed prior to the error.

InterBase allows you to include a commit statement within a statement committing work *without updating the view of the database.*

You can include the **commit** statement in the above procedure as follows:

```
QLI> define procedure store_100_cities
CON>      begin
CON>          repeat 100
CON>             begin
CON>              store cities
CON>          commit
CON>          end
CON>      end
CON> end_procedure
```

For more examples of how transactions are committed according to context, refer to description of the **commit** command in the *Qli Reference.*

Rolling Back a Transaction

Since changes you make to a database are not recorded permanently until you commit a transaction, you can undo all modifications made within the scope of a transaction before the transaction is committed.

To undo work, use the **rollback** command. A **rollback** command restores your database to the state that existed when the current transaction began. For example, the following statement readies a database (automatically beginning a transaction), and

deletes all of the records for the CITIES relation. The second statement rolls back the first, restoring all of the CITIES records.

GDML	`QLI> ready atlas.gdb`
	`QLI> erase cities`
	`QLI> rollback`

SQL	`QLI> ready atlas.gdb`
	`QLI> delete from cities`
	`QLI> rollback`

Note

Qli does not make a distinction between the standard **commit** command and the **commit** statement that does not update your view of the database. The **rollback** command throws away all changes made since the last commit, regardless of type.

Exiting Qli

When you want to end your **qli** session, you have a variety of options for ending the session and controlling the current transaction. If you have not explicitly committed or rolled back changes, here's what you can do:

- Explicitly close the database with a **finish** command. **Qli** automatically commits the current transaction before closing the database.

```
QLI> finish
```

  If you are working with more than one database, you can close each database individually. For example, the following statements close databases readied as atlas, mailer, and myrtle:

```
QLI> finish atlas
QLI> finish mailer
QLI> finish myrtle
QLI>
```

- Type **exit** or your system's end-of-file character to end your **qli** session. **Qli** automatically commits the current transaction before closing the database.

```
QLI> exit
QLI>
```

- Type **quit** to end your **qli** session. If you have made changes in the current transaction and have not committed the changes or rolled them back, the **quit** command prompts you to choose between committing the current transaction and

rolling back the database to the pre-transaction state. After you choose, **quit** executes the commit or rollback command, closes the database, and exits **qli**.

```
QLI> quit
Do you want to rollback your updates?
```

The following table summarizes options for committing or undoing a default transaction in **qli**:

Table 10-1. Transaction Control Commands

Command	Operation
commit	1. Makes the changes associated with the current transaction permanent. 2. Ends the current transaction. 3. Begins a new transaction.
rollback	1. Undoes the changes associated with the current transaction. 2. Ends the current transaction. 3. Begins a new transaction.
finish	1. Commits the current transaction or database specified. 2. Closes the current database or database specified.
exit	1. Commits the current transaction. 2. Closes the current database. 3. Ends the session.
quit	1. Prompts you to choose whether to rollback or commit changes made in current transaction. 2. Rolls back or commits the changes associated with the current transaction. 3. Closes the database. 4. Ends the session.

Special-Purpose Transactions

At any given time, **qli** can conduct two separate kinds of transactions, neither of which "knows" about the others. The first is the *default transaction* described in the previous section. A default transaction begins with a **ready**, **commit**, or **rollback** command and ends with any of those or an **exit**, **finish**, or **quit** command.

A *special-purpose transaction* is a transaction **qli** starts in response to commands that edit procedures, define procedures, and define or update metadata. **Qli** automatically commits this transaction for you as soon as it performs the requested action. Thus, if you define a procedure, but then decide to roll back whatever changes you made to your data, you do not lose your procedure definitions. However, this also means you cannot roll back any metadata definitions or updates, or any procedure definition, edit, deletion, or renaming.

Changes you make to the database under the control of the special-purpose transaction are committed immediately to the database, allowing you to use them immediately. They are only available to another user of the database if the database is readied after the changes are committed.

For example, suppose Gail and Dave each ready the database bug_reports at the same time. Dave is entering bug reports into the database. Gail defines a procedure for writing a report for bug-tracking. Dave does not have access to the bug report procedure until he concludes his session by finishing the database and then readying it again.

This mechanism ensures the database remains stable during any modification process. It is possible for conflicts to arise if the same metadata is affected. For example, if Gail and Dave each edit the same bug report, and Gail commits her transaction to the database, Dave receives a deadlock error if he tries to commit conflicting changes to the database. To resolve the conflict, Dave must finish the database and ready it again, so he is working with the updated bug report, then redo his edits and commit the changes.

For more information on avoiding conflicts and controlling transactions outside of **qli**, refer to the chapter on transactions in the *Programmer's Guide*.

For More Information

See the *Qli Reference* for the syntax of:

- **commit** command
- **rollback** command
- **prepare** command
- **finish** command

See the *Programmer's Reference* for the syntax of the **start_transaction** statement.

If you plan on using **qli** for a highly concurrent application, read the chapter on transactions in the *Programmer's Guide*.

Chapter 11
Converting Qli Statements to GDML or SQL

This chapter shows you how to convert **qli** statements into code that can be used in GDML and SQL programs.

Overview

Using **qli**'s **edit** command, you can lift statements directly out of your interactive session into an editing buffer, and from there include them in programs.

The following sections describe the differences that exist between **qli** statements and GDML or SQL code, and how to compensate for differences.

Converting from Qli to GDML

Qli's interactive version of GDML differs from the embedded version of GDML in these ways:

- Standalone **print**, **modify**, and **erase** statements are not supported. Instead, all record access is through **for** loops with substatements that update or display data, or through streams created by the **start_stream** statement.
- The **for**, **store**, and **modify** statements have corresponding **end_for**, **end_store**, and **end_modify** "tails."
- Context variables are required.
- The "print everything" default of **qli** is not supported. You must specify the fields you want to display.

Consider the following simple **print** statement in **qli**. It displays all fields from each record in the SKI_AREAS relation:

```
QLI> print ski_areas
```

The corresponding statement in a C program with embedded GDML follows:

```
for ski in ski_areas
  printf ("%s %s %s %s \n",
      ski.name, ski.city, ski.state, ski.type);
end_for;
```

The **for** loop can contain any number of host language and GDML statements, each of which is executed for each record in the stream.

If you regularly use **for** loops and context variables in **qli**, all you usually need to do to convert a **qli** statement into a statement you can include in a program is:

- Add the **end_*** tail
- Add some host language punctuation
- Add host language display statements

However, if your interactive sessions use standalone statements and eschew context variables, you have to re-work the statements to include **for** loops and add context variables in addition to the other changes.

What you might consider doing is use the standalone **qli print**, **store**, **modify**, and **erase** statements for quick *ad hoc* access, but use the more formal GDML syntax when you test logic for inclusion in programs or when you write procedures.

Converting from Qli to SQL

Qli's interactive version of SQL differs from the embedded SQL version in these ways:

- Embedded SQL distinguishes between single-record and multi-record operations. Single-record retrievals are called *singleton* selects, and use the same **select** statement as demonstrated throughout this manual. Multi-record retrievals are called *cursor* selects.
- If the query returns more than a single record or if you are unsure how many records might be returned, you must declare a *cursor* for the record selection. A cursor is a data declaration for embedded SQL. It lists the relations you want to access and the selection criteria for records you want to include.
- Embedded SQL requires the use of source code flags to signal that a statement belongs to SQL and not to the host language.

For example, consider the following SQL statement in **qli** that prints all fields from the SKI_AREAS relation:

```
QLI> select * from ski_areas
```

The following program extract:

- Declares a cursor
- Opens the cursor
- Fetches values from each SKI_AREAS record into host language variable
- Displays those values
- Closes the cursor

```
EXEC SQL
  INCLUDE SQLCA

main()
{

BASED_ON SKI_AREAS.NAME ski_name;
BASED_ON SKI_AREAS.CITY ski_city;
BASED_ON SKI_AREAS.STATE ski_state;
BASED_ON SKI_AREAS.TYPE ski_type;

EXEC SQL

  DECLARE ski_venues CURSOR FOR
  SELECT NAME, CITY, STATE, TYPE
  FROM SKI_AREAS;
```

```
EXEC SQL
  OPEN ski_venues;

EXEC SQL
  FETCH ski_venues
  INTO :ski_name, :ski_city, :ski_state, :ski_type;

while (SQLCODE == 0)

  {

  printf("%-26s %-26s %s %s\n", ski_name, ski_city, ski_state,
ski_type);

EXEC SQL

  FETCH ski_venues
  INTO :ski_name, :ski_city, :ski_state, :ski_type;
  }

EXEC SQL
  CLOSE ski_venues;
}
```

Because cursors select finite numbers of records, you must include end-of-file handling. Embedded SQL lets you provide error handling tied to the specific condition indicated by the value of the variable SQLCODE.

For More Information

For more information on GDML and SQL, see the *Programmer's Guide* and the *Programmer's Reference*.

Appendix A
Differences Between GDML and SQL

This appendix identifies features supported in **qli** and how they are implemented using GDML and SQL.

Overview

The following table can help you to determine when to use GDML or when to use SQL to access an InterBase feature through **qli**.

Table A-1. Differences between GDML and SQL

Feature	GDML	SQL
Display	print, list	select
Modify	modify	update
Store	store	insert
Erase	erase	delete
Restrict	with	where
Union	*not supported*	*supported in embedded SQL*
Null/Missing	missing	null
Case insensitive	containing	*not supported*
Unique	reduced to	distinct
Any	any	*not supported*
Exists	exists	exists
Null assignment	null, missing values	null
First n	first n	*not supported*
Aggregates	average, maximum, total, count	count, sum, avg, max, min
Group by	*supported in Report Writer*	group by
Table references	context variables	aliases

Appendix B
Sample Database Definition

The definition of the atlas database, which is used in many documentation examples, is shown below:

```
define database "atlas.gdb";

        {The atlas database is the sample database used
throughout the documentation set. It is based on a North
American atlas and gazeteer. Type "show relations" at the QLI
prompt for a listing of the relations in the database.}
  page_size 1024;


/* Global Field Definitions */

define field ALTITUDE          long;
define field AREA              long;
define field AREA_CODE         char [3];
define field AREA_NAME         varying [20];
define field CAPITAL           varying [25];
```

```
define field CENSUS_1950          long;
define field CENSUS_1960          long;
define field CENSUS_1970          long;
define field CENSUS_1980          long;
define field CENTER_FIELD         long;
define field CITY                 varying [25];
define field CODE                 varying [4];
define field COMMENTS             blob
                                   segment_length 60;
define field ELECTED_APPT         char [1]
  valid if (elect_appt = 'E'
    or elected_appt = 'A'
    or elected_appt missing);
ddefine field F1                  blob;
define field F2                   blob;
define field F3                   blob;
define field FIRST_NAME           varying [10];
define field FLAG                 char [1]
  valid if (flag = 'Y'
    or flag = 'N'
    or flag missing);
define field GUIDEBOOK            blob
                                   segment_length 60;
define field HOME_STADIUM         varying [30];
define field INCORPORATION        date
define field INIT_TERM            date;
define field LAST_NAME            varying [20];
define field LATITUDE             long;
define field LATITUDE_COMPASS     char [1]
  missing_value is "x";
define field LATITUDE_DEGREES     varying [3]
  missing_value is -1;
define field LATITUDE_MINUTES     char [2]
  missing_value is -1;
define field LEAGUE               char [1];
define field LEFT_FIELD           long;
define field LENGTH               long;
define field LOCATION             blob
                                   segment_length 60;
define field LONGITUDE            long;
define field MIDDLE_INITIAL       char [1];
define field NAME varying         [20];
define field NUM_TRAILS           long;
```

```
define field OFFICE              blob
                                 segment_length 40;
define field OUTFLOW             varying [30];
define field PARTY_AFFILIATION   char [1];
define field PHONE               char [10]
  edit_string "(xxx)Bxxx-xxxx";
define field POL_TYPE            char [1];
define field POPULATION          long;
define field POSTAL_CODE         char [10];
define field PROVINCE            varying [4];
define field RIGHT_FIELD         long;
define field RIVER               varying [30];
define field SEATING             long;
define field STATE               varying [4];
define field STATEHOOD           date;
define field STATE_NAME          varying [25];
define field SURFACE             char [1];
define field TEAM_NAME           varying [15];
define field TRAILS_LIGHTED      long
  query_name LIT;
define field TRAILS_SET          long;
define field TYPE                char [1]
  valid if (type = 'N' or
    type = 'A' or
    type = 'B');
define field YEAR                char [4];
define field YEAR_FOUNDED        char [4];
define field ZIP                 varying [10];

/* Relation Definitions */

define relation BASEBALL_TEAMS
   TEAM_NAME                     position 0,
   CITY                          position 1,
   STATE                         position 2,
   HOME_STADIUM                  position 3,
   LEAGUE                        position 4,
   LEFT_FIELD                    position 5,
   CENTER_FIELD                  position 6,
   RIGHT_FIELD                   position 7,
   SEATING                       position 8,
   SURFACE                       position 9;
```

```
define relation CITIES
    CITY                                          position 0,
    STATE                                         position 1,
    POPULATION                                    position 2,
    ALTITUDE                                      position 3,
    LATITUDE_DEGREES                              position 6
      query_name LATD,
    LATITUDE_MINUTES                              position 7
      query_name LATM,
    LATITUDE_COMPASS                              position 8
      query_name LATC,
    LONGITUDE_DEGREES
      based on LATITUDE_DEGREES                   position 9
      query_name LONGD,
    LONGITUDE_MINUTES
      based on LATITUDE_MINUTES                   position 10
      query_name LONGM,
    LONGITUDE_COMPASS
      based on LATITUDE_COMPASS                   position 11
      query_name LONGC,
    LATITUDE
      computed by (latitude_degrees | ' ' |
      latitude_minutes | latitude_compass)        position 4
    LONGITUDE
      computed by (longitude_degrees | ' ' |
      longitude_minutes | longitude_compass)      position 5;

define relation CROSS_COUNTRY
    AREA_NAME                                     position 0,
    CITY                                          position 1,
    STATE                                         position 2,
    PHONE                                         position 3
  edit_string "(xxx)Bxxx-xxxx",
    NUM_TRAILS                                    position 4,
    TRAILS_SET                                    position 5,
    TRAILS_LIGHTED                                position 6,
    INSTRUCTION
  based on FLAG                                   position 7
  query_header "INST",
    RENTALS
  based on FLAG                                   position 8
  query_header "RENT",
    REPAIRS
```

```
    based on FLAG                                   position 9
    query_header "REP",
      FOOD
    based on FLAG                                   position 10,
      LODGE
    based on FLAG                                   position 11
    query_header "BEDS",
      PACKAGES
    based on FLAG                                   position 12
    query_header "PKG",
      GUIDED_TOURS
    based on FLAG                                   position 13
    query_header "TOUR",
      COMMENTS                                      position 14;

define relation MAYORS
      CITY                                          position 0,
      STATE                                         position 1,
      PARTY_AFFILIATION                             position 3
    query_name PARTY
    query_header "party",
      INIT_TERM                                     position 4,
      ELECT_APPT                                    position 5,
      FIRST_NAME                                    position 6,
      MIDDLE_INITIAL                                position 7,
      LAST_NAME                                     position 8,
      MAYOR_NAME
    computed by (first_name | ' ' |
    last_name)                                      position 2;

define relation POLITICAL_SUBDIVISIONS
    CODE                                            position 0,
    NAME                                            position 1,
    AREA                                            position 3,
    INCORPORATION                                   position 4,
    CAPITAL                                         position 5,
    POL_TYPE;

define relation POPULATIONS
      STATE                                         position 0,
      CENSUS_1950                                   position 1,
      CENSUS_1960                                   position 2,
      CENSUS_1970                                   position 3,
      CENSUS_1980                                   position 4;
```

```
define relation POPULATION_CENTER
    YEAR                                        position 0,
    LATITUDE_DEGREES                            position 3,
    LATITUDE_MINUTES                            position 4,
    LATITUDE_COMPASS                            position 5,
    LONGITUDE_DEGREES
  based on LATITUDE_DEGREES                     position 6,
    LONGITUDE_MINUTES
  based on LATITUDE_MINUTES                     position 7,
    LONGITUDE_COMPASS
  based on LATITUDE_COMPASS                     position 8,
    LOCATION                                    position 9,
    LATITUDE
  computed by (latitude_degrees | ' ' |
  latitude_minutes | latitude_compass),
    LONGITUDE
   computed by (longitude_degrees | ' ' |
   longitude_minutes | longitude_compass);

define relation PROVINCES
    PROVINCE                                    position 0,
    PROVINCE_NAME
  based on STATE_NAME                           position 1,
    AREA                                        position 2,
    CAPITAL
   based on CITY                                position 3;


define relation RIVERS
    RIVER                                       position 0,
    SOURCE
   based on PROVINCE                            position 1,
    OUTFLOW                                     position 2,
    LENGTH                                      position 3;

define relation RIVER_STATES
    STATE                                       position 0,
    RIVER                                       position 1;

define relation SKI_AREAS
    NAME                                        position 0,
    TYPE                                        position 1,
    CITY                                        position 2,
    STATE                                       position 3;
```

```
define relation STATES
    STATE                                          position 0,
    STATE_NAME                                     position 2,
    AREA                                           position 3,
    STATEHOOD                                      position 4,
    CAPITAL
  based on CITY                                    position 5;

define relation TOURISM
    STATE                                          position 0,
    ZIP                                            position 1,
    CITY                                           position 2,
    OFFICE                                         position 3,
    GUIDEBOOK                                      position 4;

/* View  Definitions */

define view CITY_TON of c in cities
with c.city matching '*ton*'
  C.CITY                                           postion 0,
  C.STATE                                          postion 1,
  C.POPULATION                                     position 2;

define view LARGE_NON_CAPITALS of s in states
cross c in cities over state
cross cs in cities with cs.state = c.state and
cs.city = s.capital and cs.population < c.population
  C.CITY                                           position 0,
  S.STATE_NAME                                     position 1,
  S.CAPITAL                                        position 2;

define view LT_AVG_CITIES of c in cities
with c.population < average c1.population of c1 in cities
  C.CITY                                           position 0,
  C.STATE                                          position 1;

define view MIDDLE_AMERICA of c in cities
with c.longitude_degrees between 79 and 104
and c.latitude_degrees between 33 and 42
  C.CITY                                           position 0,
  C.STATE                                          position 1,
  C.ALTITUDE                                       position 2;
```

```
define view POPULATION_DENSITY of p in populations
cross s in states over state
    P.STATE                                    position 0,
    DENSITY_1950
  computed by (p.census_1950/s.area)           position 1,
    DENSITY_1960
  computed by (p.census_1960/s.area)           position 2,
    DENSITY_1970
  computed by (p.census_1970/s.area)           position 3,
    DENSITY_1980
  computed by (p.census_1980/s.area)           position 4;

define view PROVINCE_VIEW of p in political_subdivisions
with p.pol_type = 'P'
  PROVINCE FROM P.CODE                         position 0,
  PROVINCE_NAME FROM P.NAME                    position 1,
  P.AREA                                       position 2,
  P.CAPITAL                                    position 3;

define view SKI_CITIES of s in states
cross ski in ski_areas with s.state = ski.state
  SKI.NAME                                     position 0,
  SKI.CITY                                     position 1,
  S.STATE_NAME                                 position 2;


define view SKI_STATES of c in cross_country
reduced to c.state
  C.STATE                                      position 0;

define view SMALLER_CITIES of c in cities
with c.population < 500000
  C.CITY                                       position 0,
  C.STATE                                      position 1,
  C.POPULATION                                 position 2;

define view SMALL_CAPITAL_CITY of s in states
cross c in cities over state
cross cs in cities with cs.state = c.state and
cs.city = s.capital and cs.population < c.population
reduced to s.state, s.caital
  S.STATE_NAME                                 position 0,
  S.CAPITAL                                    position 1;
```

```
define view SMALL_CITY_TEAMS of b in baseball_team
cross c in cities with b.city = c.city and
b.state = c.state and b.seating > c.population / 10
   C.CITY                                          position 0,
   C.STATE                                         position 1,
   B.SEATING                                       position 2,
   C.POPULATION                                    position 3;

define view STATE_VIEW of p in political_subdivisions
with p.pol_type = 'S'
   STATE FROM P.CODE                               position 0,
   STATE-NAME FROM P.NAME                          postion 1,
   P.AREA                                          position 2,
   STATEHOOD FROM P.INCORPORATION                  position 3,
   P.CAPITAL                                       position 4;

define view VARIED_XC of c in cross_country
with c.comments containing 'varied'
   C.AREA_NAME                                     position 0,
   C.STATE                                         position 1,
   C.COMMENTS                                      position 2;

define view VILLES of c in cities
with c.city containing 'ville'
   C.CITY                                          position 0,
   C.STATE                                         position 1,
   C.POPULATION                                    position 2;

/* Index  Definitions */

define index BBT1 for BASEBALL_TEAMS unique
   TEAM_NAME,
   CITY;

define index DUPE_CITY for CITIES
   STATE;

define index CITIES_1 for CITIES unique
   CITY,
   STATE;

define index MAYORS_1 for MAYORS unique
   CITY,
   STATE;
```

```
define index RIV1 for RIVERS unique
  RIVER,
  SOURCE;

define index STATE_1 for STATES unique
  STATE;

define index XXX for TOURISM unique
  STATE;

/* Trigger Definitions*/

define trigger cascading_store for CROSS_COUNTRY
  pre store 0:
  begin
     if not any c in cities
        with c.city = new.city and c.state = new.state
     store x in cities
        x.city = new.city;
        x.state = new.state;
     end_store;
  end;
end_trigger;
```

E

F

G

H

Q

R

S

T

U

V

W