InterBase
QLI Reference

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

This book contains information on the InterBase GDML and SQL expressions, statements and commands you can use in **qli**.

Who Should Read this Book

You should read the *Qli Reference* manual if you want to use the interactive versions of InterBase's data manipulation languages. This book is a companion to the *Qli Guide* and assumes you have read that book, or are experienced with InterBase.

Using this Book

This book contains the following chapters:

Chapter 1	Introduces the book.
Chapter 2	Discusses the expressions you can use in **qli**.
Chapter 3	Discusses the GDML and SQL statements and commands you can use in **qli**.

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 manuals 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 font`	Indicates user-supplied values and example code: • $run sys$system:iscinstall • add field population_1950 long

commalist	Indicates that the 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.

Chapter 1
Introduction

This chapter describes how this manual is organized.

Overview

The *Qli Reference* contains information about the GDML and SQL expressions, commands, statements and clauses you can use in **qli**.

Each entry has the following sections:

- Function, which describes what the statement,command or expression does
- Syntax, which provides a complete diagram including all options
- Options, which describes each option of the statement, command, or expression
- Example, which shows how to use the statement, command, or expression in a program
- Troubleshooting, which lists error messages and suggests corrective actions

- See Also, which refers you to related statements or expressions, or other sources of related information.

In addition, some entries contain a section describing usage. The usage section provides an in-depth discussion of how or why to use a GDML or SQL statement, command or expression.

Chapter 2
Qli Expressions

This chapter describes the following expressions you can use in **qli**.

Overview

Qli uses the following expressions:

- Boolean expression, which evaluates to true, false, or missing.
- Predicate, which selects the records to be affected by a statement.
- Record selection expression, which specifies the search and delivery conditions for record retrieval.
- Scalar expression, which is a symbol or string of symbols to calculate a value.
- Select expression, which specifies the search and delivery conditions for record retrieval.
- Value expression, which is a symbol or string of symbols from which InterBase calculates a value.

Boolean Expression

Function

A **Boolean expression** evaluates to true, false, or missing. It describes the characteristics of a single value expression (for example, a missing value) or the relationship between two value expressions (for example, *x* is greater than *y*).

Compound Boolean expressions are evaluated in the following order: **not**, **and**, and then **or**.

Syntax

```
boolean-expression ::={[not]conditional-
   expression|
conditional-expression and conditional-
   expression|
conditional-expression or conditional-expression}

conditional-expression ::=
{any|between|comparison|containing|missing|
matching|starting with|unique}
```

These Boolean expressions are described in the following sections

- Any condition
- Between condition
- Comparison condition
- Containing condition
- Missing condition
- Matching condition
- Matching/using condition
- Starting with condition
- Unique condition

Any Condition

Function

The **any** condition tests for the existence of at least one qualifying record in a relation or relations. This expression is true if the record stream specified by a RSE includes at least one record. If

you add **not**, the expression is true if there are *no* records in the record stream.

Use **any** if you want to establish that a record exists. As soon as InterBase finds one record that meets the search criteria, it stops.

Syntax

```
[not]any rse
```

Example

The following query prints the name of any state for which there are cities stored:

```
QLI>  for s in states with any c in cities over -
CON>  state
CON>  print s.state_name
```

Between Condition

Function

The **between** condition tests whether a value expression occurs between two other value expressions.

Syntax

```
value-expression [not]{between|bt}
value-expression-1 [and] value-expression-2
```

Example

The following query looks for cities with populations between 100,000 and 250,000:

```
QLI>  for cities with population between 100000 -
CON>  and 250000
CON>  print city, state, population
```

Comparison Condition

Function

The **comparison** condition describes the relationship between two value expressions.

Syntax

```
value-expression-1 relational-operator value-
expression-2
```

relational-operator
One of the operators listed in the following table:

Operator	Relationship
eq *or* = *or* ==	Equal
ne *or* <> *or* !=	Not equal
gt *or* >	Greater than
ge *or* >=	Greater than or equal
lt *or* <	Less than
le *or* <=	Less than or equal

Example

The following query looks for cities with populations less than 50,000:

```
QLI>  for cities with population < 50000
CON>  print city, state, population
```

Containing Condition

Function

The **containing** condition conducts a case-*in*sensitive search for the presence of a substring anywhere in a *value-expression*. It evaluates to true if the substring is contained in the expression. If the value of the value expression is missing, the result is missing.

The **containing** condition also works with blobs, searching every segment in a blob for an occurrence of the quoted string.

Qli recognizes **ct** and **cont** as synonyms for **containing**.

Syntax

```
value-expression-1 [not]{containing|ct|cont}
value-expression-2
```

Example

The following query looks for cities with the substring "ville" somewhere in their name:

```
QLI>  print cities with city containing 'ville'
```

The following query prints states that entered the Union in January:

```
QLI>  print states with statehood containing 'JAN'
```

The following query looks for a COMMENTS entry in the CROSS_COUNTRY relation that contains the substring "varied":

```
QLI>  for cross_country with comments -
CON>  containing 'varied'
CON>  print area_name -
CON>  | ': ' | city | ', ' | state (-), skip,
CON>  col 10, comments (-)
```

Matching Condition

Function

The **matching** condition conducts a case-sensitive search for the presence of a substring containing the wildcard characters * and ?. The asterisk matches an unspecified run of characters, while the question mark matches a single character.

Note

If you have set matching language using the set matching language command, the definitions in that pattern are used in place of the * and ?. This may affect case-sensitivity.

Syntax

```
value-expression-1 [not] matching value-
expression-2
```

Examples

The following query looks for cities with the string "ton" following any number of other characters:

```
QLI>  print cities with city matching '*ton*'
```

The following query looks for states with the state abbreviation equal to "N" followed by exactly one character:

```
QLI>  print states with state matching 'N?'
```

Matching Using Condition

Function

The matching using condition lets you define your own wildcard search characters.

Syntax

```
matching value-expression using 'control-string'

control-string ::= [prequalifier][definition-
commalist][postqualifier]

prequalifier ::= [-S(|+S(]

definition ::= wildcard=definition-character
            [definition-character...]
postqualifier ::= [)]
```

Options

value-expression
Specifies the expression for which the substring search occurs.

prequalifier
The prequalifier string **-S(** disables case sensitivity of the *value-expression* in the **matching** clause. The prequalifier string **+S(** enables case sensitivity of the *value-expression* in the **matching** clause.

definition
Specifies the character (punctuation or symbol) you want to define and sets it equal to one or more of the characters in the following table:

Definition Character	Operation
?	Matches any single character
[]	Defines a class of character
*	Modifies previous definition or class: indicates zero or more occurrences
@	Treats the next character as literal
~	Excludes the following character or class of characters

A class of characters can be a list or range of characters that you specify inside the square brackets. For example, the range [0-9] or list [0123456789] represents any digit. If you define a class with &=[0-9A-Za-z], the ampersand represents all alphanumeric characters. The class definition of [~0-9] means any non-numerals.

postqualifer
The postqualifier) is optional.

Example

The following example searches for cities that have "ton" somewhere in their name. The **matching/using** clause defines "+" as zero or more occurrences of any single character:

```
for c in cities with c.city matching '+ton+' using
   '+=?*' print city
```

Missing Condition

Function

The **missing** condition tests for the absence of a value in an expression. It is true if the value of *value-expression* is missing.

Unless you specify otherwise in the field's definition, **qli** prints blanks for numbers, characters, and dates, and nothing for blobs. See the *Data Definition Guide* for more information about defining alternate missing values.

Syntax

value-expression [**is**] [**not**] {**missing** | **null**}

Example

The following query looks for states that have a missing value for the CAPITAL field:

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

Starting With Condition

Function

The **starting with** condition conducts a case-sensitive search for the presence of a substring at the beginning of a value expres-

sion. It evaluates to true if the first characters of the value expression match the substring.

Syntax

```
value-expression-1 [not] {starting|st} [with]
value-expression-2
```

Example

The following query looks for cities that start with the string "New":

```
QLI>  print states with state_name -
CON>  starting with 'New'
```

Unique Condition

Function

The **unique** condition tests for the existence of exactly one qualifying record. This expression is true if the record stream specified by the RSE consists of only one record. If you add **not**, the condition is true if there is more than one record in the record stream or if the record stream is empty. The format of the *unique-condition* follows:

Syntax

```
[not] unique rse
```

Example

The following query prints the names of states that have only one ski area:

```
QLI>  for s in states with unique ski in -
CON>  ski_areas over state
CON>  print s.state_name
```

Troubleshooting

See the discussion of errors and error handling in Chapter 1 of the *Qli Guide*.

See Also

See the entries in this manual for:

- value expression
- RSE
- predicate

Predicate

Function

The *predicate* clause is used to select the records to be affected by the statement. It is used in the *where-clause* of the SQL statements **delete** and **update**, and in the *select-expression*.

Syntax

```
predicate::={condition|condition and predicate|
             condition or predicate|
not predicate}
condition::={between-condition|compare-condition|
             exists-condition|like-condition|
             null-condition|(predicate)}
```

The following sections describe the five conditions of the *predicate* clause:

- Between condition
- Compare condition
- Exists condition
- Like condition
- Null condition

Between Condition

Function

The *between-condition* specifies an inclusive range of values to match.

Syntax

```
database-field [not] between scalar-expression-1
and scalar-expression-2
```

Example

The following query displays the CITY and STATE fields from cities with populations between 100,000 and 125,000:

```
QLI> select city, state from cities where -
CON> population between 100000 and 125000
```

Compare Condition

Function

The *compare-condition* describes the characteristics of a single scalar expression (for example, a missing or null value) or the relationship between two scalar expressions (for example, *x* is greater than *y*).

Syntax

```
{scalar-expression comparison-operator scalar-
expression|scalar-expression comparison-operator
(column-select-expression)|scalar-expression is}
[not] null

comparison-operator::={=|^=|<|^<|<=|^>|>=}

column-select-expression::= select [distinct]
scalar-expression from-clause [where-clause]
```

Example

The result of the following query displays all fields from CITIES records for which the POPULATION field is not missing:

```
QLI> select * from cities where
CON> population is not null;
```

Exists Condition

Function

The *exists-condition* tests for the existence of at least one qualifying record identified by the **select** subquery. Because the *exists-condition* uses the parenthesized **select** statement only to retrieve a record for comparison purposes, it requires only wildcard (*) field selection.

A predicate containing an *exists-condition* is true if the set of records specified by *select-expression* includes at least one record. If you add **not**, the predicate is true if there are *no* records that satisfy the subquery.

Example

The following query tests to see if at least one record that satisfies the condition exists:

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

In Condition

Function

The *in-condition* lists a set of scalar expressions as possible values. The **in** operator allows you to equate a value to any of sevarl values. You can use the **in** operator in subqueries.

Syntax

```
scalar-expression [not] in (set-of-scalars)

set-of-scalars::= {constant-commalist|column-
select-expression}

column-select-expression::=select [distinct]
expression from-clause [where-clause]
```

Example

The following query selects records from the CITIES relation with city names that are in the specified set:

```
QLI> select city, state, population from cities -
CON> where city in ('Boston', 'Providence',
'Albany')
```

Like Condition

Function

The *like-condition* matches a string with the whole or part of a field value. The test is case-sensitive.

Syntax

```
database-field [not] like scalar-expression
[escape character]
```

Options

scalar-expression

Usually represents an alphabetic or numeric literal, and can contain wildcard characters. Wildcard characters are:

- The underscore, _, which matches a single character
- The percent sign, %, which matches any sequence of characters, including none. You should begin and end wildcard searches with the percent sign so that you match leading or trailing blanks.

escape *character*

Tells **qli** to treat the next character as itself rather than as a wildcard. This allows you to search for strings that contain "%" or "_."

Example

The following query displays all fields from STATES record in which the CAPITAL field contains the string "ville" preceded or followed by any number of characters:

```
QLI> select * from states where capital like
'%ville%';
```

The following query uses the **escape** clause to find strings containing "@."

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

Troubleshooting

See the discussion of errors and error handling in the *Qli Guide.*

See Also

See the entries in this chapter for:

- select expression
- scalar expression
- **delete**
- **update**

Record Selection Expression (RSE)

Function

The **record selection expression** specifies the search and delivery conditions for record retrieval.

Syntax

```
[first-clause] record-source [with-clause]
[reduced-clause][sorted-clause]

record-source::={relation-clause|cross-clause}

relation-clause::=[context-variable in]
relation-name

cross-clause::=relation-clause cross
record-source [over field-name-commalist]
```

The following sections describe the six clauses of the record selection expression:

- First clause
- Relation clause
- Cross clause
- With clause
- Reduced clause (project)
- Sorted clause

First Clause

Function

The **first** clause limits the records in a stream to the number you specify with an integer.

Syntax

```
first integer-expression
```

Options

integer-expression

Specifies the number of records you want to include. **Qli** truncates any fractional portion of the integer. Unless you sort the

record stream when you use the *first-clause*, *n* records are selected at random.

Example

The following query uses a *first-clause* and a *sorted-clause* to display the two youngest states:

```
QLI> for first 2 states sorted by descending
statehood
CON> print state_name |
CON> '  was admitted to the Union on  ' | statehood
Hawaii was admitted to the Union on 21-AUG-1959
Alaska was admitted to the Union on 3-JAN-1959
```

Relation Clause

Function

The *relation-clause* identifies the target relation.

Syntax

```
[context-variable in] [database-handle.]
relation-name
```

context-variable

Used for name recognition, and is associated with a relation. A context variable can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

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

database-handle

Identifies the database for multiple database access.

Example

The following statement uses a *relation-clause*, a *cross-clause*, and a *sort-clause* to display only those states in which the capital is not the largest city:

```
QLI> for s in states cross c in cities over state
CON> cross
CON> cs in cities with cs.state = c.state and
CON> cs.city = s.capital and
CON> cs.population < c.population -
CON> sorted by s.state -
CON> reduced to s.state, s.capital -
```

```
CON> print s.state_name,
CON> ' contains cities larger than ', s.capital
```

Cross Clause (Join)

Function

The *cross-clause* performs a join operation. It creates dynamic relationships by matching up records from two or more relations in the same database.

The relationship can be based on the equality of common fields (equijoin), inequalities (non-equijoin), or the absence of relationships (cross product). Unlike most other *rse* clauses, *cross-clause* can be repeated to include as many relations as are necessary.

Syntax

cross *relation-clause* [**over** *field-name-commalist*]

Options

over
Equates a field in one relation with a field in another, like a *with-clause*. The *field-name* must be exactly the same in both relations. Otherwise, you must use the *with-clause*, even if both fields are based on the same global field.

Examples

The following query displays the names of cities that are larger than the capitals of their states:

```
QLI> for s in states cross c in cities over state -
CON> cross
CON> cs in cities with cs.state = c.state and
CON> cs.city = s.capital and
CON> cs.population < c.population -
CON> sorted by s.state, c.city
CON> print c.city, s.state_name,
CON> ' is larger than ', s.capital
```

The following query uses two *relation-clause*s and a *cross-clause* to list a ski area, city, and state in which it is located:

```
QLI> for s in states cross ski in ski_areas over
CON> state
CON> print ski.name, ski.city, s.state_name
```

The following query does the same thing as the preceding query, but uses an explicit join condition in place of the **cross** shortcut:

```
QLI> for s in states cross ski in ski_areas with
CON> s.state = ski.state
CON> print ski.name, ski.city, s.state_name
```

The following query displays SKI_AREAS records that duplicate another ski area record in everything but the TYPE field:

```
QLI> for s1 in ski_areas cross s2 in ski_areas with
CON> s1.name = s2.name and
CON> s1.state = s2.state and
CON> s1.city = s2.city and
CON> s1.type > s2.type
CON> print s1.state, s1.name
```

Because there are no duplicate records in the SKI_AREAS relation, this query does not return any records.

With Clause

Function

The *with-clause* specifies a search condition or combination of 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.

Syntax

```
with boolean-expression
```

Options

boolean-expression
Specifies a valid Boolean expression used to select records.

Example

The following query specifies an explicit join condition:

```
QLI> for s in states cross ski in ski_areas with
CON> s.state = ski.state
CON> print ski.name, ski.city, s.state_name
```

Reduced Clause (Project)

Function

The *reduced-clause* performs a project operation, retrieving only the unique values for a field.

When you ask for a record stream projected on a field, InterBase considers a list of fields and eliminates records that do not have a unique combination of values for the listed fields. If you include a **reduced** clause in an RSE, you can reference only the fields listed in the **reduced** clause and statistical expressions in **print** statements driven by the RSE. Do not erase or modify records whose RSE includes a **reduced to** clause.

Syntax

```
reduced [to] dbfield-expression-commalist
dbfield-expression ::= [context.variable]
field-name
```

Example

The following query uses a *reduced-clause* to list the states in which there are ski areas:

```
QLI> print ski_areas reduced to state
```

Note that this query returns only the values of the STATE field, ignoring all other fields.

Sorted Clause

Function

The *sorted-clause* orders the output, returning the record stream sorted by the values of one or more sort keys.

Syntax

```
sorted [by] sort-key-commalist
sort-key ::= [ascending|descending]
[anycase|exactcase] dbfield-expression
dbfield-expression ::= [context.variable]
field-name
```

Options

sort-key
Specifies the field or fields on which you want to sort. You can sort a record stream alphabetically, numerically, by date, and by any combination of these. The *sort-clause* lets you have as many sort keys as you want.

Each sort key can specify whether the sorting order is **ascending** (the default order for the first sort key) or **descending**.

The sorting order is "sticky"; that is, if you do not specify whether a particular sort key is **ascending** or **descending**, InterBase 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, InterBase sorts all keys in descending order.

The sort key can also specify whether a sort is case sensitive or not. A case sensitive sort (**exactcase**) sorts capital letters before lowercase letters. A case insensitive sort (**anycase**) does not. The default is **exactcase**.

Example

The following query uses a *first-clause*, a *relation-clause*, and a *sorted-clause* to display the two "youngest" states:

```
QLI> for first 2 states sorted by descending
CON> statehood
CON> print state_name |
CON> '  was admitted to the Union on  ' | statehood
```

The following statement displays the names of states and capitals in which the capital is not the largest city:

```
QLI> for s in states cross c in cities over state
CON> cross
CON> cs in cities with cs.state = c.state and
CON> cs.city = s.capital and
CON> cs.population < c.population -
CON> sorted by s.state -
CON> reduced to s.state, s.capital
CON> print s.state_name,
CON> ' contains cities larger than ', s.capital
```

Troubleshooting

See the discussion of errors and error handling in the *Qli Guide*.

See Also

See the entries in this chapter for:

- Boolean expression
- Value expression

Scalar Expression

Function

The *scalar-expression* is a symbol or string of symbols used in predicates to calculate a value. InterBase uses the result of the expression when executing the statement in which the expression appears.

You can add (+), subtract (-), multiply (*), and divide (/) scalar expressions. Arithmetic operations are evaluated in the normal order of addition, subtraction, multiplication, division. You can use parentheses to change the order of evaluation.

The concatenation operator (|) is a formatting convenience. For example, if your **select** command includes a list of value expressions separated by commas, **qli** displays the field values in columnar order, padding out things like varying string fields with blanks. However, you can use the concatenation operator and constants to print a more legible display.

Syntax

```
scalar-expression ::= [-]scalar-value
[arithmetic-operator scalar-expression]
scalar-value ::={field-expression |
constant-expression | statistical-function |
(scalar-expression) }
arithmetic operator::={+|-|*|/||}
```

These scalar expressions are described in the following sections

- Database field expression
- Constant expression
- Statistical function

Database Field Expression

Function

The *field-expression* references a database field.

Syntax

```
[relation-name.|view-name.|alias.]
   database-field
```

Options

relation-name
view-name
alias

Specifies the relation, view, or alias (synonym for a relation or view) in which the field is located. The alias is assigned to a relation or a view in a *select-expression*.

The expression can be of the form *alias*.* or *relation*.* This way you can request all the fields from a relation without having to list them all.

Examples

The following query displays all fields from the CITIES record that represents Boston:

```
QLI> select * from cities where city = 'Boston'
```

The following query displays selected fields from the same record:

```
QLI> select population, altitude, latitude,
CON> longitude -
CON> from cities where city = 'Boston'
```

The following query displays selected fields from CITIES with a population greater than 1,000,000:

```
QLI> select city, state, population from cities -
CON> where population > 1000000
```

The following query joins records from the CITIES and STATES relations:

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

The following query selects cities with a population within 100,000 of the population of Boston:

```
QLI> select city, state, population, altitude from
CON> cities where
CON> population between
CON> (-99999) + (select total (population) from
CON> cities where city = 'Boston') and
CON> 99999 + (select total (population) from
CON> cities where city = 'Boston')
```

Constant Expression

Function

The *constant-expression* specifies a string of ASCII digits interpreted as a number or as a string of ASCII characters.

Syntax

```
{integer-string|decimal-string|float-string|
ascii-string}
```

Options

integer-string
Written as signed or unsigned decimal integers without decimal points. For example, the following are integers: *-14, 0,* and *9.*

decimal-string
Written as signed or unsigned decimal integers with decimal points. For example, the following are decimal strings: *-14.3, 0.021,* and *9.0.*

floating-string
Written in scientific notation (that is, *E-format*). A number in scientific notation consists of a decimal string mantissa, the letter *E*, and a signed integer exponent. For example, the following are floating numerics: *7.12E+7* and *7.12E-7.*

ascii-string
Qli accepts single quoted (') or double quoted (") characters. It accepts unquoted ASCII strings containing only letters. An unquoted string is converted to uppercase. The ASCII printing characters are shown in the following table:

Characters	Description
A—Z	Uppercase alphabetic
a—z	Lowercase alphabetic
0—9	Numerals
! @ # $ % ^ & * ( ) _ - + = ‘ ~ [ ] { }	Special characters

Example

The following query displays selected fields from CITIES with a population greater than 1,000,000:

```
QLI> select city, state, population from cities -
CON> where population > 1000000
```

Statistical Function

Function

A *statistical-function* is an expression that calculates a single value from the values of a field in a relation, view, join, or group of records.

Syntax

```
{count (*)|
function-name (scalar-expression) |
function-name (distinct) field-expression
function-name ::= {count | sum |avg | max | min}
```

count (*)
Returns the number of qualifying records in a relation.

count
Returns the number of unique values for the field. You must specify distinct.

sum
Returns the sum of values for a numeric field in all qualifying records.

avg
Returns the average value for a numeric field in all qualifying records, eliminating null values.

max
Returns the largest value for the field.

min
Returns the smallest value for the field, ignoring null values.

Example

The following example returns a count of records in the CITIES relation, the maximum population, and the minimum population of cities in that relation:

```
QLI> select count ( * ), max (population),
CON> min (population) from cities
```

Troubleshooting

See the discussion of errors and error handling in the *Qli Guide*.

See Also

See the entry in this chapter for predicate.

Select Expression

Function

The *select-expression* specifies the search and delivery conditions for record retrieval.

Syntax

```
select-clause [where-clause]
```

The following sections describe the four clauses of the select expression:

- Select clause
- Where clause
- Grouping clause
- Having clause

Select Clause

Function

The *select-clause* lists the fields to be returned and the source relation or view.

Syntax

```
select [distinct] scalar expression-commalist
from from-item-commalist

from-item ::= relation-name [alias]
```

Options

distinct
Causes InterBase to perform a projection of the qualifying records on the scalar expressions listed. No combination of values appears more than once.

alias
The optional *alias* is used for name recognition, and is associated with a relation. An alias can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character. **Qli** is not sensitive to the case of the alias. For example, it treats "B" and "b" as the same character.

Example

The following query projects the SKI_AREAS relation on the STATE field:

```
QLI> select distinct state from ski_areas
```

Where Clause

Function

The *where-clause* specifies search conditions or combinations of search conditions.

When you specify a search condition or combination of conditions, the condition is evaluated for each record that might qualify.

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.

Syntax

```
where predicate
```

Options

predicate
Specifies the predicate used to select records.

Examples

The following query returns CITIES records for which the POPULATION field is not missing:

```
QLI> select city, state, population from cities -
CON> where population is not null
```

The following query joins two relations on the STATE field for cities whose population is not missing:

```
QLI> select c.city, s.state_name from cities c,
CON> states s -
CON> where c.state = s.state and
CON> c.population not missing
```

The following query prints the name, state, and population of cities that are not the largest in their state:

```
QLI> select c1.city, c1.state, c1.population from
CON> cities c1 where
CON> exists (select * from cities c2 where
```

```
CON> c2.state = c1.state and
CON> c2.population > c1.population )
```

The following query prints the name, state, and population of states which are larger than the average city in their state:

```
QLI> select c1.city, c1.state, c1.population from
CON> cities c1 where
CON> c1.population > (select avg (population) from
CON> cities c2 where
CON> c2.state = c1.state) order by state
```

Grouping Clause

Function

The *grouping-clause* partitions the results of the *from-clause* or *where-clause* into control groups, each group containing all rows with identical values for the fields in the *grouping-clause*'s field list. Aggregates in the *select-clause* and *having-clause* are computed over each group. The *select-clause* returns one row for each group.

The aggregate operations are count (**count**), sum (**sum**), average (**avg**), maximum (**max**), and minimum (**min**). See the entry for *scalar-expression* in this chapter.

Syntax

```
group by database-field-commalist
```

Options

database-field
Specifies the field the values of which you want to group. Each set of values for these fields identifies a group.

Example

The following request provides a total population by state of municipalities stored in the CITIES relation, but includes only those cities for which the latitude and longitude information has been stored, which are located in states whose names include the word "New", and where the average population of cities in the state exceeds 200,000 people:

```
QLI> select sum (c.population), s.state_name
CON> from cities c, states s -
CON> where s.state_name like '%New%' and
CON>  c.latitude is not null and
CON>     c.longitude is not null and
```

```
CON>      c.state = s.state -
CON>  group by s.state -
CON>  having avg (population) > 200000
```

Having Clause

Function

The *having-clause* specifies search conditions for groups of records. If you use the *having-clause*, you must first specify a *grouping-clause*.

The *having-clause* eliminates groups of records, while the *where-clause* eliminates individual records. Generally speaking, you can use subqueries to obtain the same results. The main advantage to the use of this clause is brevity. However, some users may find that a more verbose query with subquery is easier to understand.

Syntax

```
having predicate
```

Example

The following cursor provides a total population by state of municipalities stored in the CITIES relation, but includes only those cities for which the latitude and longitude information has been stored, which are located in states whose names include the word "New", and where the average population of cities in the state exceeds 200,000 people:

```
QLI> select sum (c.population), s.state_name
CON>  from cities c, states s -
CON>  where s.state_name like '%New%' and
CON>     c.latitude is not null and
CON>     c.longitude is not null and
CON>     c.state = s.state -
CON>  group by s.state -
CON>  having avg (population) > 200000
```

Troubleshooting

See the discussion of errors and error handling in the *Qli Guide*.

See Also

See the entries in this chapter for:

- Predicate
- Scalar expression

See the entry for **select** in Chapter 3.

Value Expression

Function

The *value-expression* is a symbol or string of symbols from which InterBase calculates a value. InterBase uses the result of the expression when executing the statement in which the expression appears.

Syntax

```
value-expression ::= {arithmetic-expression |
dbfield-expression | first-expression |
format-expression | numeric-literal-expression |
prompting-expression | quoted-string-expression
| running-expression | statistical-expression
|username-expression}
```

The following sections discussion the nine options of the value expression:

- Arithmetic expression
- Database field expression
- First expression
- Format expression
- Numeric literal expression
- Prompting expression
- Quoted string expression
- Running expression
- Statistical expression
- Username expression

Arithmetic Expression

Function

The *arithmetic-expression* combines value expressions and arithmetic operators.

You can add (+), subtract (-), multiply (*), and divide (/) value expressions in assignment statements. Arithmetic operators are evaluated in the normal precedence of addition, subtraction, mul-

tiplication, division. Use parentheses to change the order of evaluation. You can use the concatenation operator (|) to combine field values in record selection expressions.

Syntax

```
value-expression-1 {+ | - | * | / | | }
value-expression-2
```

Example

The following statement includes an arithmetic value expression that calculates and displays the altitude in meters:

```
QLI> for c in cities cross s in states over state
CON> print c.city, s.state_name | ' is situated at
' |
CON> c.altitude * 0.3048 | ' meters above sea
level.'
```

Database Field Expression

Function

The *dbfield-expression* references database fields. This expression can occur in several clauses of an RSE and Boolean expression.

Syntax

```
[context-variable.] field-name
```

context-variable
Qualifies the database field for multi-relation operations. You must declare a context variable for a relation in the *relation-clause* of the record selection expression.

Example

The following statement uses database field expressions to display the city and state, an arithmetic value expression that calculates and displays the altitude in meters, a numeric literal expression (0.3048) used in the arithmetic operation, and two quoted string expressions:

```
QLI> for c in cities cross s in states over state
CON> print c.city, s.state_name |
CON> ' is situated at ' |
CON> c.altitude * 0.3048 | ' meters above sea
level.'
```

First Expression

Function

The *first-expression* forms a record stream and evaluates an expression. InterBase finds the first qualifying record in the record stream. If the stream is empty, it returns an error unless you supply an **else** clause. Otherwise, InterBase evaluates *value-expression-2* in the context of the record it found. The result of the evaluation is returned as the value of *first-expression* or the value specified in the **else** clause.

If you use the *first-expression* in a **print** command, you must enclose it in parentheses. Otherwise, **qli** assumes that you are using the *first-clause* of the record selection expression.

Syntax

```
first value-expression-1 from rse
```

Example

The following query returns the abbreviation for a specific state:

```
QLI> print city, state of cities with
CON> state = first state from states with
CON> state_name = 'Missouri'
```

Format Expression

Function

The *format-expression* forces a value to an alphanumeric representation using the specified edit string. You can use the **format** expression to specify a format when moving a value to a text field or variable, or to specify formats for parts of a concatenation.

Syntax

```
format value-expression using edit-string
```

Example

The following examples use the formatting value expression:

```
QLI> declare a char[10]
QLI> a = 'today'
QLI> print a
today
QLI> a = format 'today' using w(9)
QLI> print a
Thursday
QLI> print 'Today is ' | 'today' using w(9)
** QLI error: Error converting string "Today is
today" to date **
```

```
QLI> print 'Today is ' | format 'today' using w(9)
Today is Thursday
QLI>
QLI> report ....
CON> at top of page print col 55, "Page " | format
running count using Z9,
CON> skip, column_header
```


Numeric Literal Expression

Function

The *numeric-literal-expression* represents a decimal number as a string of digits with an optional decimal point.

Syntax

```
[+ | -] string[.string]
```

Example

The following statement uses database field expressions to display the city and state, an arithmetic value expression that calculates and displays the altitude in meters, a numeric literal expression (0.3048) used in the arithmetic operation, and two quoted string expressions:

```
QLI> for c in cities cross s in states over state
CON> print c.city, s.state_name | ' is
CON> situated at ' |
CON> c.altitude * 0.3048 | ' meters above sea
CON> level.'
```

Quoted String Expression

Function

The *quoted-string-expression* represents a string of ASCII characters enclosed in single (') or double (") quotation marks. The following table shows the ASCII printing characters:

Characters	Description
A—Z	Uppercase alphabetic
a—z	Lowercase alphabetic

Characters	Description
0—9	Numerals
! @ # $ % ^ & * ( ) _ - + = ‘ ~ [ ] { }	Special characters

Example

The following statement includes two quoted string expressions:

```
QLI> for c in cities cross s in states over state
CON> print c.city, s.state_name | ' is situated at
' |
CON> c.altitude * 0.3048 | ' meters above sea
level.'
```

Running Expression

Function

The *running-expression* calculates a running count for a control item or a running total for an expression.

Ordinarily, **running count** is used to count records in a record stream, but when you use it with the report writer, it also counts other control items. For example, the report writer phrase *at top of page print running count* prints the current page number. Similarly, *at top of state print running count* prints the number of occurrences of the control field.

Example

The following example uses the running value expression:

```
QLI> print running count, state_name of states -
CON> sorted by statehood

QLI> print running total population, city,
CON>population of cities where state = 'NY'
```

Statistical Expression

Function

The *statistical-expression* calculates a value based on a value expression.

If a field value included in *value-expression* is missing for a record, that record is not included in the calculation. For **average**, **max**, and **min**, if the record stream created by *rse* is empty, the value of the statistical expression is missing. For **total** and **count**, if the record stream is empty, the total is 0.

Syntax

```
{statistical-operation value-expression of rse |
count of rse}
statistical-operation ::= {average | max | min |
total }
```

Example

The following query uses several statistical expressions:

```
QLI> for states with any cities over state with
CON> altitude * 0.3048 > 1000
CON> print state_name | " incorporates at least one
kilometer high city. ",
CON> skip, col 5, "The average height of cities in
" | state | " is " |
CON> average altitude of cities over state | " feet
and the maximum is " |
CON> (max altitude of cities over state), skip
```

Username Expression

Function

The *username-expression* is a value expression that automatically picks up the username or login of the person running the program. Combined with a trigger that automatically stores the username of users storing or modifying records, you can keep track of who does what to which records. See the *Data Definition Guide* for more information about triggers.

Syntax

```
rdb$user_name
```

rdb$user_name

A value expression to which is assigned the username or login. This expression can only be used in RSEs, and cannot be qualified with a context variable.

Example

The following statement picks up the username and uses an RSE that selects records based on the value of the USER_NAME field:

```
for employees with user_name = rdb$user_name
   print emp_name, user_name
```

Troubleshooting

See the discussion of errors and error handling in the *Qli Guide*.

See Also

See the entries in this chapter for:

- Boolean expression
- Record selection expression

Chapter 3
Qli Statements and Commands

This chapter contains entries for the InterBase **qli** statements.

Overview

Requests made to **qli** fall into two categories:

- Statements

  Statements generally retrieve and alter data, or affect the order of execution of statements. For example, the GDML **for**, **print**, **store**, **modify** and **erase** statements and the SQL **select**, **update**, **insert** and **delete** statements retrieve and alter data.

  Statements that affect the order of execution include the **if/else**, **repeat** and **begin/end** statements.

- Commands.

  Commands affect the operating environment. Commands include the **ready** and **finish** commands, and the metadata change requests like **define relation** and **alter table**.

The distinction between commands and statements is important when you are working with compound statements. Many statements can enclose other statements. For example, the **if/else** statement has two branches: true and false. Each branch takes a statement which executes if the condition matches the branch. Only statements can be used in an **if/else** statement, between a **begin-end**, in a **repeat** loop or in the loop of a **for** statement.

Each of the requests below is defined as a command or statement in its description.

Abort	Accept	Alter table
Assignment	Begin-end	Commit
Copy procedure	Create database	Create index
Create table	Create view	Declare
Define database	Define field	Define index
Define procedure	Define relation	Delete
Delete metadata	Delete procedure	Drop database
Drop index	Drop procedure	Drop table
Drop view	Edit	Edit procedure
Erase	Exit	Field attributes clause
Finish	For	For form
Grant	Help	If-else
Insert	List	Modify
Modify field	Modify index	Modify relation
Prepare	Print	Quit
Ready	Rename procedure	Repeat
Report	Restructure	Revoke
Rollback	Select	Set
Shell	Show	Spawn
Store	Then clause	Update

Abort

Function

The **abort** statement aborts your current request and brings you back to the **qli** prompt. You can use this statement in procedures to respond to error conditions.

Syntax

```
abort value-expression
```

Options

value-expression
The value expression that is evaluated in the context of the current RSE. This is the value expression that prints out in the error message that the abort statement generates.

Example

The following example looks for cities that meet particular conditions. If the condition is not met, the abort statement ends the request and returns a simple message to notify the user:

```
QLI>  for cities with population not missing -
CON>  sorted by altitude -
CON>  begin -
CON>     if population < 1000000 print city,
CON>     population else
CON>     abort city |
CON> "is too big for a small town boy"
CON>  end
** QLI error: Request terminated by statement:
New York is too big for a small town boy **
```

Troubleshooting

You may encounter the folowing message when you use the abort statement:

- "value-expression" is undefined or used out of context

  You specified a value expression that doesn't exist. Check the spelling and try again.

See Also

See the entry in this chapter for value expression.

Accept

Function

Qli displays assignments to form fields at an **accept** statement. If you omit assignments to form fields, those fields are displayed as missing.

Syntax

```
accept [(quoted-string)] [field-name-commalist]
```

Options

quoted-string
The quoted string provides a tag line which prints at the bottom of a form.

field-name-commalist
The field name commalist is a list of form field names that are made available for input.

Example

The following example displays a form to accept the input of a state code, and then displays a form to data from cities in that state:

```
QLI> for form f in cities
CON> begin
CON> accept ("Enter state code, then <enter>") CON>
state
CON>  for c in cities with c.state = f.state
CON>  begin
CON>  f.state = c.state
CON>  f.city = c.city
CON>  f.altitude = c.altitude
CON>  f.latitude = c.latitude
CON>  f.longitude = c.longitude
CON>  accept ("Hit <enter> to continue or <f1> to
stop")
CON>  end
CON> end
```

Troubleshooting

You may encounter the following messages when you use the **accept** statement:

- field <name> is not defined in form <name>
  A field that you mentioned is not in the form.

- no context for accept statement

  Your program lacks an outer **for form** statement.

See Also

See the entry in this chapter for:

- **for form**
- **for menu**

Alter Table

Function

The **alter table** statement drops a field from a table or adds a field to a table. Unlike the "standard" SQL **alter table** statement, **gpre** lets you perform multiple drops and adds in one statement.

Syntax

```
alter table table-name operation-commalist
operation::= {add field-name datatype[not null]|
drop field-name}
```

Options

table-name
Specifies the table you want to change.

field-name
Names the field you want to add or drop. If you add a field to a table, the field name must be unique among all field names in the table.

For a list of datatypes, see the entry in this chapter for **create table**.

not null
Disallows the null or missing value as a valid value for this field.

Example

The following statements alter tables by adding and dropping fields:

```
QLI>  alter table states add
CON>  type_of_govt char(3), add
CON>  capital varchar(25);

QLI>  alter table cities
CON>  drop population;
```

Troubleshooting

See the discussion of errors in Chapter 1 of the Qli Guide

See Also

See the discussion on defining metadata in the *Qli Guide*.

Assignment

Function

The **assignment** statement assigns values to fields in the modify and store statements, or values to variables. The *Qli Guide* contains a detailed discussion of the assignment statement.

Syntax

Field Assignment:

```
dbfield-expression-1 = {value-expression|edit
[dbfield-expression-2]}
dbfield-expression::=
[context-variable.]field-name
```

Variable Assignment:

```
variable-name = value-expression
```

Options

dbfield-expression-1
Specifies the field to receive a value. The context-variable optionally names the relation.

value-expression
Specifies the value you want to assign to dbfield-expression. This value can be a quoted literal, a reference to another field, a prompting value expression (*.'your own prompt'), an aggregate, computation, statistical expression, or the word null to assign the missing value.

dbfield-expression-2
Specifies the blob field whose contents you would like to edit for assignment to dbfield-expression-1.

variable-name
Identifies the variable to which you want to assign a value. The variable must have been declared with a **declare variable** command.

value-expression
Specifies the value you want to assign to variable-name.

Examples

The following example stores a record, using a **begin-end** statement to structure a compound statement for assigning values to each field:

```
QLI>  store cross_country using
CON>  begin
CON>     city = 'Andover'
CON>     state = 'MA'
CON>     area_name = 'Parker State Forest'
CON>     phone = '5085550123'
CON>     num_trails = 25
CON>     trails_set = 0
CON>     lighted = 0
CON>     instruction = 'N'
CON>     rentals = 'N'
CON>     repairs = 'N'
CON>     food = 'N'
CON>     lodge = 'N'
CON>     packages = 'N'
CON>     guided_tours = 'N'
CON>  end
```

The following example modifies a field value:

```
QLI> for c in cities with c.city = "Boston"
CON> modify using c.population =
CON> c.popu3zlation * 1.10
```

Troubleshooting

You may encounter the following messages when you use the assignment statement:

- *Operation failed on database "database-filename"* with any of the following secondary messages:
- *Arithmetic exception, numeric overflow, or string truncation*

  A value that you tried to store or modify did not fit. Check the field's characteristics and try again.
- *Attempt to store a duplicate value in a unique index*

  A field value that you tried to store or modify violated the "duplicates not allowed" restriction for an index that includes that field. Try another value.
- *Conversion error*

  This is a generic data conversion error that covers all but the following two cases:

- *Conversion error from string "out-of-range-date"*

  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.

- *Conversion to blob not supported*

  You tried to store non-blob data in a blob field. Use the edit option described above.

- *Validation error for field field-name, value "supplied-value"*

  A field value that you tried to store or modify violated the valid_if clause for a field. Check the valid values and try again.

- "String" is undefined or used out of context

  This is a **qli** message in response to an unrecognized string.

- *Execution terminated by signal*

  You probably issued an end-of-file command.

- *User aborted (WC -Q) edit operation (display manager/ Pad manager)*

  You probably exited from the editor during a blob assignment without doing anything. This informational message comes from outside **qli**, and means that the target blob will contain exactly the same value as the source blob.

- *No permission for "type" access to "object"*

  A security class exists for the specified object, and its access control list prohibits you from reading or writing that object.

- *Action cancelled by trigger to preserve data integrity*

  A trigger exists for an object you tried to modify or delete, and the corollary actions associated with the trigger prohibit that operation.

See Also

See the entries in this chapter for:

- **declare variable**
- **begin-end**
- **modify**
- **store**

Begin-End

Function

The **begin-end** statement describes a block of **qli** statements that act together.

Syntax

```
begin
   qli-statement
end
```

Options

qli-statement
Any of the **qli** statements or any procedure containing statements.You cannot use a command or a procedure containing commands as the action of an **if-else** statement.

Examples

The following example stores a record and uses a **begin-end** block for the assignment statements:

```
QLI> store cities using
CON>  begin
CON>     city = 'Shadkill'
CON>     state = 'NY'
CON>     population = 20000
CON>     altitude = 17
CON>  end
```

The following example defines a procedure that stores 30 cities:

```
QLI> define procedure store_30_cities
CON>  begin
CON>     repeat 30 store cities
CON>     print skip, "Well done!", skip
CON>  end
CON> end_procedure
```

Troubleshooting

You may encounter the following message when you use the **begin-end** statement:

Expected statement, encountered command

This message means you included a command in the **begin-end** block. Only the statements listed under *qli-statement* above should be used in the **begin-end** block. In general, actions that affect or report on the database environment (**ready**, **finish**,

show, and **set**) are commands and cannot be included in a **begin-end** block.

See Also

See the entries in this manual for any **qli** statement.

Commit

Function

Commit can be a command or a statement. It makes changes a permanent part of the database.

When typed to the **qli** prompt, **commit** is a command. It changes the transaction environment, letting others see your changes and letting you see changes that others have made to the database.

When you type **commit** inside another statement (e.g. **for**, **if-else**,or **begin-end**) **qli** executes a **commit** statement. A **commit** statement makes your changes permanent and available to others. It does not change your transaction environment. You cannot see changes that others have made to the database since your transaction began.

You can use **commit** in conjunction with the **prepare** statement to execute a two-phase commit. InterBase automatically executes such a commit when necessary, but, if required, you can control the two-phase commit explicitly. See the entry for **prepare** in this chapter.

Syntax

```
commit [database-handle-commalist]
```

Options

database-handle

Specifies a name that can be used to qualify database reference when you are using multiple databases. If you do not specify a database handle, the **commit** command affects all open databases.

If you assign a database handle when you ready the database, you can use the handle to limit the effect of the **commit** to specific databases. When you access more than one database in **qli**, InterBase automatically starts up separate subtransactions for each database. Each subtransaction is a single transaction. The optional *database-handle* lets you control these subtransactions explicitly by letting you commit or roll back transactions by database.

If you forgot to assign a database handle when you readied the database and run into a problem with a database while you have several open, do not despair; **qli** assigns a default handle

if you have not specified one. Type the following to find out the default database handle assigned by **qli**:

```
QLI>  show databases
   Database "atlas.gdb" readied as QLI_0
```

Qli displays the names of all available entities, including databases and handles. The default handles are of the form "QLI_n," where *n* is a numeric integer. Supply this handle as an argument to **commit**:

```
QLI>  commit qli_1
```

Examples

The following example readies a database and stores a record which starts a transaction. The final line commits the transaction:

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

The following example stores records in a loop, committing each one:

```
QLI> repeat 10 begin
CON> store river_states
CON> commit
CON> end
```

Troubleshooting

You may encounter the following message when you use **commit**:

Expected database handle, encountered <string>

You need a database handle. Check your typing, or use the **show databases** command to check the database handle.

See Also

See the entries in this manual for:

- **rollback**
- **finish**
- **prepare**

Copy Procedure

Function

The **copy procedure** command copies a stored procedure.

Syntax

```
copy procedure [database-handle.]procedure-name
[to] [database-handle.]procedure-name
```

Options

procedure-name
Specifies the procedure you want to copy.

database-handle
Specifies a name that can be used to qualify the procedure name when you are using multiple databases. If you do not specify a handle, **qli** looks at the most recently opened database for a procedure with the name you provided. If it can't find it there, **qli** continues its search backwards through the databases you opened.

If you forgot to assign a database handle and want to use one, use the default handle assigned by **qli**. Use the **show databases** command for a list of handles associated with each database.

Example

The following command copies a procedure:

```
QLI> copy procedure sunbelt_cities warm_cities
```

Troubleshooting

You may encounter the following message when you use the **copy procedure** statement:

Procedure <name> not found

The procedure does not exist as specified. Check your typing, or use the **show procedures** command for a list of procedures.

See Also

See the *Qli Guide* for a comprehensive discussion of procedures.

See also the entries in this chapter for:

- **define procedure**
- **delete procedure**
- **edit procedure**
- **rename procedure**

Create Database

Function

The **create database** statement creates a database and its system tables.

Syntax

```
create database quoted-filespec
[pagesize=integer]
```

Options

quoted-filespec

Specifies the database file. If the shell you regularly use is case-sensitive, make sure that you always reference the database file exactly as it is spelled out in the create database statement.

The file specification can contain the full pathname to another node in the network. File specifications for remote databases have the following form.

Table 3-1. File Specifications for Remote Databases

From	To	Syntax
VMS	VMS via DECnet	node-name::filespec
VMS	ULTRIX via DECnet	node-name::filespec
VMS	non-VMS and non-ULTRIX	node-name^filespec
ULTRIX	VMS via DECnet	node-name::filespec
Apollo	Apollo	//node-name/filespec
Everything Else	Whatever is left	node-name:filespec

pagesize=*integer*

Specifies a page size to override the default page size of 1024 bytes. You can create databases with page sizes of 1024, 2048, 4096, and 8192 bytes. The advantage of a larger page size is that it allows a more shallow "tree" structure in the index. Each index bucket is one page long, so longer pages mean larger buckets and fewer levels in the index hierarchy. If you will have more than 50,000 records in any one table, you should use a page size of 2048 rather than the default.

Example

The following statement creates a database in the current directory:

```
QLI>  create database 'personnel.gdb';
```

Troubleshooting

See the discussion on errors in the *Qli Guide*.

See Also

For more information about creating a database and for other database file options, see the *Data Definition Guide*.

For more information about SQL metadata operations, see the *Qli Guide*.

See also the entries in this chapter for:

- **create table**
- **create index**
- **create view**

Create Index

Function

The **create index** statement defines an index for a relation.

Syntax

```
create [unique] [ascending|descending] index
index-name on relation-name(field-name-commalist)
```

Options

unique
Disallows duplicate values in the index. The values for the indexed fields must be unique. If you try to store a value that already exists, the assignment operation fails.

ascending | descending
Specifies the order in which an index is built. If neither qualifier is specified, the default order is ascending. Using the qualifier does not replace using the order by clause in the **select** statement.

index-name
Names the index. The index name must be unique among all index names in the database.

relation-name
Identifies the relation for which the index is defined.

field-name-commalist
Specifies a column name or list of field names, separated by commas, that comprise the index.

Examples

The following statements create a non-unique and unique index, respectively:

```
QLI>  create index xxx on states (capital);
QLI>  create unique index xyz on states (state);
```

The following example creates a descending index on the LENGTH field in the RIVERS relation:

```
QLI>  create descending index longriv on rivers
      (length);
```

Troubleshooting

See the discussion of errors and error handling in Chapter 1 of the *Qli Guide*.

See Also See the discussion of defining metadata in the *Qli Guide*.

Create Table

Function

The **create table** statement defines a relation and its constituent fields.

Syntax

```
create table relation-name(field-definition-
 commalist)
 field-definition::= field-name datatype[not
null]
 datatype::= {smallint|integer|date|
char(integer)| varchar(integer)|decimal[scale]|
float|long float}
```

Options

relation-name

Names the relation you want to create. A relation name can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character and be unique among relation names in the database.

field-name-commalist

Specifies the name you want for the field in the relation. The field name must be unique among all field names in the relation. The following table lists the SQL datatype and what InterBase gives you.

Table 3-2. SQL and InterBase Datatypes

SQL Datatype	InterBase Datatype
smallint	short
integer	long
date	date
char	char
varchar	varying
decimal	long

Table 3-2. SQL and InterBase Datatypes

SQL Datatype	InterBase Datatype
float	float
long float	double

`not null`
Disallows the null or missing value as a valid value for this field.

Usage

Using the **create table** statement automatically invokes the SQL security scheme for that table. If you create a table, you are that table's owner and accordingly have all privileges for that table. You also have grant option for those privileges for that table. See the entries in this chapter for **grant** and **revoke** for further information on SQL security.

Note

You cannot assign a security class to relations created with the SQL **create table** command. Instead, you control access to these relations by using SQL **grant** and **revoke** commands.

Example

The following statements define relations:

```
QLI> create table states (
       state char(2) not null,
       state_name varchar(25),
       area integer,
       statehood date,
       capital varchar(25));

QLI> create table populations (
       state char(2) not null,
       census_1950 integer,
       census_1960 integer,
       census_1970 integer,
       census_1980 integer));
```

Troubleshooting

See the discussion of errors and error handling in Chapter 1 of the *Qli Guide*.

See Also

See the discussion on defining metadata in the *Qli Guide*.

Create View

Function

The **create view** statement creates a temporary view of data. When you create a view by using embedded SQL, the view definition is *not* stored on the database. As a result, you cannot access this definition through **qli** or **gdef**.

Syntax

```
create view view-name[(field-name-commalist)]
as select-statement
```

Options

view-name
Names the view you want to create. The view name must be unique among all view names in the database.

field-name-commalist
Optionally names the fields for the view. If you choose not to supply a field name, gpre uses the field name as specified in the select statement that follows. Because the field names map chronologically to the list of selected fields in the select statement, you must specify all view field names or none.

If you supply the field name, it must be unique among all field names in the view.

select-statement
A select statement that specifies the selection criteria for records to be included in the view. Instead of the into clause used in queries, the list of selected fields maps to the list of field names for the view. The order is based on the value of the RDB$FIELD_POSITION field in the RDB$RELATION_FIELDS system relation.

The select statement cannot contain a group by clause or top-level aggregation.

Examples

The following statements define views:

```
QLI> create view half_mile_cities as
CON> select city, state, altitude from cities
CON> where altitude > 2500;

QLI> create view capital_cities as
CON> select c.city, s.state_name, c.altitude
```

```
CON> from cities c, states s where
CON> c.state = s.state and c.city = s.capital;
```

Troubleshooting

See the discussion of errors and error handling in Chapter 1 of the *Qli Guide*.

See Also

See the discussion on defining metadata in the *Qli Guide*.

Declare

Function

The **declare** statement lets you declare local and global variables for use in **qli**. You can use variables in statements, reports, and procedures. To assign a value to a variable, use the **assignment** statement.

Ordinarily, variables are available throughout a **qli** session. They are called "global" variables, and their scope is the entire session. However, 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. 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 it.

Syntax

```
declare variable-name {datatype | based on
[dbhandle.]relation.field}

datatype::={short[scale-clause]|long[scale-
clause]|float|double|char[n]|varying[n]|date}

scale-clause::=scale [-]n
```

Options

variable-name

Names the variable. The variable name must start with an alphabetic character (*a-z*), and can contain numbers, underscores, and dollar signs.

Choose variable names carefully. Variable names that conflict with keywords or field, relation, or database names can cause confusion, resulting in misunderstood queries and improbable answers. See the beginning of this chapter for a list of **qli**'s keywords.

scale-clause
Specifies the power of 10 by which InterBase multiplies the stored integer value for use by **qli**. For example, a negative scale of two means that there should be a decimal point two places to the left of the digits.

The following table lists the datatypes by size and range/precision.

Table 3-3. Datatypes by Size and Range/Precision

Datatype	Size	Range/Precision
short	16 bits	-32768 to 32767
long	32 bits	-2**31 to (2**31)-1
float	32 bits	Approximately 7 decimal digits
double	64 bits	Approximately 15 decimal digits
char[*n*]	n bytes	0 to 32767 characters
varying[*n*]	Varies up to *n* bytes	0 to 32767 characters
date	64 bits	1 January 100 to 11 December 5941
blob	Varies	None

based on
Creates a variable with the same datatype, scale and length as the field specified.

dbhandle
An optional identifier indicating which database the relation is in.

relation
A required qualifier for the specified field.

field
Specifies the field within the relation on which to base a new field.

Examples

The following command declares a variable GLARP, assigns 617, and prints the variable's value:

```
QLI>  declare glarp long
QLI>  glarp = 617
QLI>  print glarp
       GLARP
      =========
        617
```

The following extract uses a prompting expression in the variable assignment:

```
QLI>  declare glarp long
QLI>  glarp = *.'value for glarp'
      Enter value for glarp: 412
```

The following example uses an edit string to print a variable:

```
QLI>  delcare c based on cities.city
CON>  c = "Boston"
CON>  print city of cities with city = c
         CITY
      ====================================
      Boston
```

Troubleshooting

You may encounter the following message when you use the **declare variable** statement:

expected field definition clause, encountered "bad string"

You tried to declare a variable, but the field definition was incorrect. Check the syntax and try again.

See Also

See the entries in this chapter for:

- **assignment**
- **begin-end**

Define Database

Function

The **define database** command creates a database definition file and readies the newly created database for access.

Syntax

```
define database filespec
```

Options

filespec

Specifies the primary file. If the shell you regularly use is case-sensitive, make sure that you always reference the database file exactly as it is spelled in the **define database** statement.

The file specification can contain the full pathname to another node in the network. File specifications for remote databases have the following form:

Table 3-4. File Specifications for Remote Databases

From	To	Syntax
VMS	VMS via DECnet	node-name::filespec
VMS	ULTRIX via DECnet	node-name::filespec
VMS	non-VMS and non-ULTRIX	node-name^filespec
ULTRIX	VMS via DECnet	node-name::filespec
Apollo	Apollo	//node-name/filespec
Everything Else	Whatever is left	node-name:filespec

Example

The following statements define databases:

```
QLI> define database /gds/examples/atlas.gdb
QLI> define database atlas.gdb
```

Troubleshooting

You may encounter a privilege or protection violation from the operating system when you try to create a database. Check the directory in which you are creating the database to make sure that you have the privilege to create files there.

See Also

See the entries in this chapter for:

- **define field**
- **define relation**

Define Field

Function

The **define field** command defines a global field.

Syntax

```
define field field-name datatype [options]
```

Options

field-name
Names the field you want to create. A field name can contain up to 31 characters that can be alphanumeric, dollar signs ($), and underscores (_). However, it must start with an alphabetic character. It also must be unique among all global fields in the database.

datatype
Specifies the field's dataype. The datatype specification must precede other field attributes. See the entry in this chapter for *field-attributes* for more information.

options
Specifies a query name or edit string. See the entry in this chapter for *field-attributes* for more information.

Example

The following example defines a global field:

```
QLI> define field flag char[1]
```

Troubleshooting

You may encounter the following messages when you use the **define field** command:

- *Global field <field-name> already exists*

  The name you chose for the field is already in use in this database for another field. Use a different name and try again.

- *No datatype specified for field <field-name>*

  Unlike some of the clauses and commands that know they are not complete, the **define field** command expects a datatype on the same line or on a continuation line. Use a hyphen to continue a line or the **semicolon** option on the **set** command.

- Expected field definition clause, encountered "string"

  You specified a field name that began with a non-alphabetic character or included an unrecognized attribute in the definition. If the former case is true, change the field name so it

starts with an alphabetic character. If the latter case is true, check the command and make sure that you have included a variable attribute.

See Also

See the entry in this chapter for *field attributes*.

Define Index

Function

The **define index** command defines an index for a relation. You must define a relation before you can create an index for it. Because the index is created as part of the command, response will be slow if the relation is large.

InterBase automatically maintains all indexes. You do not have to reference an index when you access data—the InterBase access method does it automatically.

Syntax

```
define index index-name [for] relation-name
[{unique|duplicate[s]}]
[active|inactive}]
[ascending|descending]
field-name-commalist
```

Options

index-name
Names the index you want to create. An index name can contain up to 31 characters that can be alphanumeric, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

relation-name
Specifies the relation for which you are defining the index. You cannot define an index on an external relation.

unique
Disallows duplicate values in the index. Try to index on fields used as *primary keys*, such as unique identification numbers, part numbers, employee numbers, etc.

You can define a unique index by specifying the optional keyword **unique**. If you do so, the values for *field-name* or combinations of *field-name*s must then be unique. If you try to store a value that already exists, the assignment operation fails. No part of a unique key may be null.

If you create a multi-segment index, you should first consider which of the key fields is likely to have the most unique values. Having done so, you should list the *field-name*s in descending

order by uniqueness. Such ordering improves partial key retrieval.

duplicate
Allows duplicate values in the index. This is the default.

active | inactive
Active specifies that the index should be built immediately. Inactive specifies that the index should be built at a later time. If the definition of an index is marked as inactive, InterBase maintains only the index definition in the database. When you change the state of the index to active with the **modify index** statement, the index is built and becomes available to all users.

ascending | descending
Specifies the order in which an index is built. If neither qualifier is specified, the default order is ascending.

For increased efficiency in returning sorted values, use the qualifier that corresponds to the order you are most likely to specify in an order by or sort by clause. Using the qualifier does not replace using an ordering clause when you retrieve values.

field-name
Specifies one or more fields from *relation-name* that are indexed.

You can create a single or multi-segment index for a relation. A single-segment index consists of a single field, while a multi-segment index consists of two or more fields. In both cases, you should avoid indexing a field that has few unique values. Such indexes provide little performance improvement and can reduce update performance. Because of the nature of the blob datatype, you cannot index a blob field.

Example

The following statements define relations and some indexes for them:

```
QLI define relation states
      ⇓
QLI define relation cities
      ⇓
QLI> define index state_idx1 for states -
CON> unique state;
QLI> define index state_idx2 for states -
CON> inactive unique state, state_name;
```

```
QLI> define index river_idx1 for rivers -
CON> descending river;
QLI> define index rivstat_idx1 for river_states -
CON> duplicate river, state;
```

Troubleshooting

You may encounter the following message when you use the **define index** command:

- *Index<index-name> already exists*

  The name you chose for the index is already in use in this database for another index. Use a different name and try again.

- *No_meta_update - too few key fields found for index <index-name>*

  You did not provide enough fields for an index. This message often means that you listed a field for the index, but it does not exist in the relation. Check the roster of fields in the relation and their spelling and try again.

See Also

See the entry in this chapter for define relation.

Define Procedure

Function

The **define procedure** statement stores a sequence of **qli** operations in the database.

Syntax

```
define procedure [database-handle.]
procedure-name operation...
end_procedure
operation::={qli-command|qli-procedure|
qli-statement|qli-clause|qli-keyword}
```

Options

[database-handle.]procedure-name

Names the procedure. The procedure name can be up to 31 characters and can contain alphabetic characters (A—Z and a—z, all stored as uppercase), numeric characters (0—9), underscores (_), and dollar signs ($). The procedure name must start with an alphabetic character.

The optional database handle specifies the database in which the procedure is stored.

qli-statement

Any of the **qli** statements or any procedure containing statements.You cannot use a command or a procedure containing commands as the action of an **if-else** statement.

There can be only one statement per physical line, unless you separate the statements with a semicolon.

A single statement can span more than one physical line. When **qli** encounters a return, it attempts to execute the statement. This attempt can cause problems, especially if your command was not quite complete (for example, *erase cities* followed by a return, rather than *erase cities with state = "VI"*), or if the statement is.

qli-command

Any of the **qli** commands described in this chapter. There can be only one command per physical line, unless you separate the commands with a semicolon.

qli-clause

A clause from a **qli** command or statement.

qli-keyword
A **qli** keyword.

comment
You can include a running commentary in your procedures so that other people can figure out what it does. Comment lines in **qli** begin with a slash and asterisk and end with an asterisk and a slash:

```
QLI> define procedure gl1451
CON> /* This procedure does some useful things */
CON> print states with capital = *.'capital city'
```

Example

The following sequence of commands defines a procedure that finds a record using a prompted-for value:

```
QLI> define procedure capital_info
CON> /* saves typing a common query */
CON> for s in states cross c in cities over
CON> state with s.state = *.'state code' and
CON> s.capital = c.city
CON> print s.capital | ' has a population of ' |
CON> c.population
CON> end_procedure
QLI> :capital_info
      Enter state code: AZ
      Phoenix has a population of 789704
```

Troubleshooting

You may encounter the following messages when you use the **define procedure** statement:

- *Procedure name <name> in use*

  Choose another name.

- *Procedure name over 31 characters*

  Choose a shorter name.

- *Gds_$create_blob failed*

  InterBase could not create the field in which the procedure text is stored. Try again.

  You may get the following errors when you execute a procedure:

- *Procedure <name> is undefined*

  The procedure does not exist as specified. Type *show procedures* for a list of procedures.

- *Procedure <name> not found*

  The procedure does not exist as specified. Type *show procedures* for a list of procedures.

See Also

For a complete discussion of procedures, see the chapter on using procedures in the *Qli Guide*.

For more information on line continuation characters, see the introductory chapter of the *Qli Guide*.

See the entries in this chapter for:

- **copy procedure**
- **edit procedure**
- **delete procedure**
- **rename procedure**

Define Relation

Function

The **define relation** statement creates a relation and several types of fields for the relation, or copies an existing relation. The **based on** clause of the **define relation** statement lets you create relations based on relations. This command copies field names and field characteristics from the old relation to the new one. It does not copy triggers, indexes, or data.

Note

When you specify a new relation be based on a view definition, **qli** defines the new relation as a regular stored relation. It does not define the relation as a view.

Syntax

```
define relation [dbhandle.]relation-name
 field-description-commalist

field-description::=
{included-field | new-field | renamed-field}

included-field::=
field-name[edit-string][query-name]

new-field::=
field-name datatype[edit-string][query-name]

renamed-field::=
field-name based on field-name
[edit-string][query-name]
```

Options

relation-name

Names the relation you want to create or copy. A relation name can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character, and its name must be unique among relation names in the database.

field-description-commalist

Specifies the name(s) you want for the field(s) in the relation. The **define relation** statement supports several types of field definitions, all of which you can use in the same relation definition.

included-field
Included fields are defined in previous **define field** or **define relation** commands.You can define local attributes for included fields or accept the attributes defined previously. Local attributes override global attributes. Field attributes are described in the entry for *field-attributes* in this chapter.

new-field
New fields are defined in the relation. This clause defines a new field within the relation, instead of using existing fields. Because the field is defined from scratch, you must include the field's datatype. **Qli** adds these fields to the list of global fields for that database. You can use those fields in subsequent relation definitions.

renamed-field
This clause renames an existing field for use in the relation being defined. The field retains all characteristics except those you change explicitly. You can include an edit string and query name. If you specify a local attribute that conflicts with the attribute defined for the global field, the local attribute overrides the global attribute.

database_handle
Specifies the database handle you assigned to the database when you readied it. You can use this name to qualify the relation name when you are using multiple databases. If you do not specify a handle, **qli** looks at the most recently opened database for a relation with the name you provided. To avoid confusion when using more than one database, always use database handles to qualify both the new and existing relations.

Example

The following example defines several fields and shows how to use them in the **define relation** statement:

```
QLI>  define field state_char[2]
QLI>  define field state_name varying [25]
QLI>  define field city varying[25]
QLI>  define relation states
CON>  state,
CON>  state_name,
CON>  area long,
CON>  statehood char[4],
CON>  capital based on city
```

The following example uses the **define relation** command with the **based on** clause to copy an existing relation:

```
QLI> ready phone.gdb
QLI> define relation personal_numbers
CON> based on relation phone_numbers
```

Troubleshooting

You may encounter the following messages when you use the **define relation** statement:

- *Relation <relation-name> already exists*

  The name you chose for the relation is already in use in this database for a relation *or* a view. Use a different name and try again.

- *Global field <field-name> already exists*

  The name you chose for the field is already in use in this database for another field. Use a different name and try again.

- *No datatype specified for field <field-name>*

  You failed to provide a datatype for the field. For a list of datatypes, see the entry in this chapter for *field-attributes*.

  However, the problem may be that you hit the carriage return prematurely. The **define relation** command's field definition clause expects a datatype on the same line or on a continuation line. Use a hyphen to continue a line or the **semicolon** option on the **set** command.

- *Expected field definition clause, encountered "string"*

  You specified a field name that began with a non-alphabetic character or you included an unrecognized attribute in the definition. If the former case is true, change the field name so it starts with an alphabetic character. If the latter case is true, check the command and make sure that you have included a valid attribute.

See Also

See the chapter on defining data in the *Qli Guide*.

See also the section on field attributes in this chapter.

Delete

Function

The **delete** statement erases one or more records in a relation.

If you do not provide a search condition, InterBase deletes all records in the specified relation. Be very careful with this statement.

You cannot delete records directly from a view that is a join: you must erase them through each participating relation separately. You can delete records from a view if the view is comprised of a single relation (without a reflexive join) or if the database designer included an *erase* trigger for the view.

Syntax

```
delete from relation-name [alias]
[where predicate]
```

Options

relation-name
Specifies the relation from which a record is to be deleted.

alias
Qualifies field references with an identifier that indicates the source relation. The alias can be useful if predicate references sources with overlapping field names.

The *alias* can contain up 31 characters that can be alphanumeric, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

where *predicate*
Determines the record(s) to be deleted. If you provide a search condition with the optional **where** *predicate* clause, InterBase deletes the record(s) selected from *relation-name*.

Examples

The following statement deletes all records from the CITIES relation with a value for POPULATION of less than 100,000:

```
QLI>  delete from cities -
CON>  where population < 100000;
```

The following statement deletes the entire TERRITORIES relation:

```
QLI>  delete from territories;
```

The following example deletes all qualifying records:

```
QLI>  delete from ski_areas -
CON>  where name = 'Birchwood Slopes';
```

Troubleshooting

You may encounter the following messages when you use the **delete** statement:

- *Can't erase from a join*

  You tried to erase from a join. This is an illegal operation. If you want to erase records in different relations, you must do so in separate statements for each relation.

- *No context for ERASE*

  You did not provide a record selection expression. Try again.

See Also

See the entries in this chapter for:

- predicate
- select expression

Delete Metadata

Function

The **delete metadata** commands erase the specified database entity *and all data associated with that entity*. Therefore, you must be very sure that you want to delete something before you do it.

Although objects can be deleted from an active database, it is generally better to wait until the database is not in use if you want to delete relations and indexes.

Syntax

```
delete {field field-name|index index-name|
relation relation-name}
```

Usage

The following are the usage rules for the **delete metadata** commands:

- You can delete a field that was defined in a **define field** or **define relation** command. However, because fields are included in a relation, you must first delete the field from each relation in which it is included. The following commands delete a field from a relation and then from the database:

```
QLI> modify relation states
CON> delete field statehood
QLI> delete field statehood
```

- You must explicitly delete a field defined in a **define relation** command after dropping it from relations in which it is used. For example, suppose you defined the field BIRTH_DATE in an EMPLOYEES relation, subsequently included it in another relation, and then decided not to keep it.

  The following sequence deletes the field from all its instances and then finally from the database itself:

```
QLI> modify relation employees
CON> drop field birth_date
QLI> modify relation demographics
CON> drop field birth_date
QLI> delete field birth_date
```

Do not drop fields from relations unless you are sure that nothing else depends on their data. Dropping fields causes programs that depend on them to fail. InterBase returns an error if you try to delete a field used in a computed field, view or trigger definition.

- You can delete any index you want. If you try to delete an index in use, your command does not complete until the index can be deleted. The followoing command deletes an index:

```
QLI>  delete index idx4
```

- You can delete any relation you want. If you try to delete a relation inuse, your command does not complete until the relation is free and can be deleted. Programs relying on relations that have been deleted fail when they attempt to reference a deleted relation. Because the **delete relation** command immediately removes a relation and all its records, you should use this command with caution. The following commands delete relations:

```
QLI>  delete relation gudgeons
QLI>  delete relation non_eeoc_approved_data
```

InterBase treats views much like relations. In fact, the syntax for deleting a view uses the word **relation**. When you delete a view, other users should not encounter any problems if they are already running their programs. If they start up a program that references the deleted view, the program fails when it tries to compile the request that mentions that view.

Example

The following commands delete views:

```
QLI> delete relation population_density
QLI> delete relation geo_cities
QLI> delete relation riv_vu
```

Troubleshooting

You will encounter an error message if the database entity you want to delete does not exist. If you receive such a message, check to see if you have spelled the name correctly.

See Also

See the entries in this chapter for:

- **define field**
- **define index**
- **define relation**

Delete Procedure

Function

The **delete procedure** command deletes a stored procedure.

Syntax

```
delete procedure [database-handle.]
procedure-name
```

Options

procedure-name
Specifies the procedure you want to delete.

database-handle
Specifies the database handle you assigned to the database when you readied it. You can use this name to qualify the procedure name when you are using multiple databases. If you do not specify a handle, **qli** looks at the most recently opened database for a procedure with the name you provided. If it can't find the a procedure there, **qli** continues its search backwards through the databases you opened. If you forgot to assign a database handle and want to use one, use the default handle assigned by **qli**. Type *show databases* for a list of handles associated with each database.

Example

The following example deletes a procedure:

```
QLI>  delete procedure sunbelt_cities
```

Troubleshooting

If you happen to acquire a procedure with an illegal name, such as "*" or "3", you cannot use the **delete procedure** command. Instead, you must use the **erase** command to delete it or the **modify** command to change its name.

The procedure is stored in the relation QLI$PROCEDURES, and its name is stored in the QLI$PROCEDURE_NAME field. The following statement deletes a procedure named "*":

```
QLI>  for qli$procedures with -
CON>  qli$procedure_name = '*'
CON>  erase
```

Be sure you qualify the record selection expression so you delete only the offending procedure.

You may encounter the following message when you use the **delete procedure** statement:

Procedure <name> not found

The procedure does not exist as specified. Type *show procedures* for a list of procedures.

See Also

See the *Qli Guide* for a comprehensive discussion of procedures.

See also the entries in this chapter for:

- **copy procedure**
- **define procedure**
- **edit procedure**
- **rename procedure**

Drop Database

Function

The **drop database** statement deletes an entire database.

Syntax

```
drop database filespec
```

Option

filespec
Specifies the database you want to drop.

Example

The following example deletes the entire database:

```
QLI> drop database phones.gdb
```

Troubleshooting

You may encounter the following message when you use the **drop database** command:

No such file or directory

The file does not exist as you specified. Type *show databases* for a list of databases.

See Also

See the discussion on defining metadata in the *Qli Guide*.

Drop Index

Function

The **drop index** command deletes an index.You cannot delete an index in use in an active database. If you do so, InterBase returns an error message.

Syntax

```
drop index index-name
```

Option

index-name
Specifies the index you want to delete.

Example

The following example deletes an index:

```
QLI> drop index statesnames;
```

Troubleshooting

You may encounter the following message when you use the **drop index** command:

Index "index-name"is not defined in database "database-name"

Type *show indexes* for a list of relations and their indexes.

See Also

See the discussion of metadata operations in the *Qli Guide*.

Drop Table

Function

The **drop table** command deletes a relation. You cannot delete a relation in use in an active database. If you do so, InterBase returns an error message. This statement also deletes all indexes for the relation and any views that reference it.

Syntax

```
drop table relation-name
```

Option

relation-name
Specifies the relation to drop.

Example

The following examples deletes a relation:

```
QLI> drop table tourism;
```

Troubleshooting

You may encounter the following messages when you use the **drop table** command:

- *field "field-name" cannot be deleted because "n" view(s) depend on it*

  Type *show views* for a list of views and their fields.

- *exprected relation or view name, encounterd "relation-name"*

  The relation does not exist as you specified. Type *show relations* for a list of relations.

See Also

See the discussion of metadata operations in the *Qli Guide*.

Drop View

Function

The **drop view** command deletes a view. This command also deletes any other views that reference the deleted view. However, the relations that comprise the view are not affected.

Syntax

```
drop view view-name
```

Option

view-name
Specifies the view you want to drop.

Example

The following example deletes a view:

```
QLI> drop view colonies;
```

Troubleshooting

You may encounter the following messages when you use the **drop view** command:

expected relation or view name, encountered "relation-name"

The view does not exist as you specified.Type *show views* for a list of views and the fields they are comprised of.

See Also

See the discussion of metadata operations in the *Qli Guide*.

Edit

Function

The **edit** command calls your default text editor, places the contents of a **qli** command or statement in the editing buffer, and then deposits you in that buffer. You can then revise the **qli** commands or statements.

Use the standard editing commands to change the command line as you want. When you finish éditing, exit from the editor as you normally do. You can also use this command to repeat the previous command. To do so, invoke the editor by issuing the **edit** command and then exit without making any changes. In most cases, **qli** automatically executes the command.

The exception is for systems that use the **vi** editor. **Vi** does not distinguish between an unchanged filed and an aborted eidt. On these systems, you must change something to cause **qli** to re-execute the command from the edit buffer.

Syntax

```
edit [integer|*]
```

Options

integer

Places the last *n* commands in the editing buffer. For example, **edit** 5 puts the last five commands in the buffer.

*

Places all commands issued during your **qli** session into the editing buffer.

Example

The following example corrects a syntax error:

```
QLI>  for cities with state = New York
      ** QLI> error: expected end of statement,
      encountered "YORK"
QLI>  edit
```

Qli calls your default editor. To fix this query, change the state name New York to its abbreviation NY. Once you exit from the editor, **qli** executes the query and displays the records:

```
QLI>

                CITY
============================

Albany
Buffalo
New York
```

Troubleshooting

The only errors you receive from this command are those generated by your editor. Problems can include a lack of disk space or protection violations that prevent the editor from opening a scratch or journal file.

See Also

See the documentation for the text editor you use.

Edit Procedure

Function

The **edit procedure** command lets you change a stored procedure or create a new one.

When you issue the **edit procedure** command, **qli** calls your default editor. If the procedure already exists, **qli** writes the text of the procedure to the editing buffer. The buffer does not include the **define procedure** and **end_procedure** structure. Use the standard editing commands to change the procedure.

If the procedure does not exist, **qli** opens an empty edit window or buffer. Enter **qli** commands or statements. Do not use the **define procedure** and **end_procedure** commands that you would use to define a procedure at the QLI> prompt; **qli** supplies these commands for you.

When you finish editing a procedure or inserting a new one, exit from the editor as you normally do. **Qli** automatically stores the procedure in the database.

Note

When you edit a procedure, **qli** first searches all open databases for a procedure with that name. If it finds one, **qli** puts the text in the edit buffer and allows you to modify that procedure. If **qli** cannot find the procedure you name, it creates a new procedure in the most recently readied database. To avoid confusion, use database handles when working with several procedures.

Syntax

```
edit [database-handle.]procedure-name
```

Option

[database-handle.]procedure-name
Names the procedure you want to edit or create. The optional database handle specifies the database in which the procedure is stored.

Example

The following example calls the default editor and writes the text of the high_cities procedure to the editing buffer:

```
QLI> edit high_cities
```

Troubleshooting

You may get error messages from your editor. Possible problems include a lack of disk space or protection violations that prevent the editor from opening a scratch or journal file. You may also encounter the following messages when you use the **edit procedure** command:

- *Procedure name <name> in use*

  Choose another name.
- *Procedure name over 31 characters*

  Choose a shorter name.
- *Gds_$create_blob failed*

  InterBase could not create the field in which the procedure text is stored. Try again.

See Also

See the *Qli Guide* for a comprehensive discussion of procedures.

See also the entries in this chapter for:

- **copy procedure**
- **define procedure**
- **delete procedure**
- **rename procedure**

Erase

Function

The **erase** statement removes from the database the record(s) specified by the record selection expression.

You cannot erase records directly from a join: you must erase them through each participating relation separately. You can delete records from a view if the view is comprised of a single relation (without a reflexive join) or if the database designer included an *erase* trigger for the view.

Syntax

```
erase[[all of] rse]
```

Options

all of *rse*
Specifies which records specified record selection expression are to be deleted:

- If you do not specify an RSE, you must select the record or records to be deleted in an outer **for** loop.
- If you do specify an RSE, you must provide a complete record selection expression in the **erase** statement.
- Do not erase records whose RSE includes a **reduced to** clause.

Examples

The following example identifies the record to be deleted in an RSE in the **erase** command itself:

```
QLI>  erase all of cities with population < 1000 or
CON>  population missing
```

The following example uses a **for** loop to identify the record you want to erase:

```
QLI>  for cities with population < 100000 or
CON>  population missing
CON>  begin
CON>  print
CON>  if *.'keep it?' containing 'n' erase
CON>  end
```

Troubleshooting

You may encounter the following messages when you use the **erase** statement:

- *Can't erase from a join*

  You tried to erase from a join. This is an illegal operation. If you want to erase records in different relations, you must do so in separate statements.

- *No context for erase*

  You did not provide a record selection expression. Correct the statement using an **edit** command and try again.

See Also

See the entries in this chapter for:

- **for**
- record selection expression

Exit

Function

The **exit** command commits the current transaction, closes all databases, and ends the **qli** session.

Exit and the end-of-file character are exactly equivalent. The end-of-file characters are system-dependent (and user-dependent on some systems):

- *Control-Z* for VAX/VMS, MicroVMS, and Apollo
- *Control-D* for ULTRIX and other UNIX systems

Syntax

```
exit
```

Example

```
QLI> exit
```

Troubleshooting

See the discussion of errors and error handling in Chapter 1 of the *Qli Guide*.

See Also

See the entries in this chapter for:

- **quit**
- **commit**
- **finish**

Field Attributes

Function

The *field attributes* clause describes the characteristics of fields defined or modified by the following statements:

- **define field**
- **modify field**
- **define relation**
- **modify relation**

Syntax

```
field-attributes ::= datatype|edit-string|
query-name}
```

The following sections describe the three optional clauses for the *field-attributes* clause:

- Datatype clause
- Edit-string clause
- Query-name clause

Datatype Clause

Function

The **datatype** clause specifies the datatype of a field.

Syntax

```
datatype ::= {short [scale-clause] |
long [scale-clause] | float | double | char[n] |
varying[n] | date | blob}
   scale-clause ::= scale[-]n
```

The following table lists the datatypes by size and range/precision.

Table 3-5. Datatype Size and Precision

Datatype	Size	Range/Precision
short	16 bits	-32768 to 32767
long	32 bits	-2**31 to (2**31)-1
float	32 bits	Approximately 7 decimal digits
double	64 bits	Approximately 15 decimal digits
char[*n*]	n bytes	0 to 32767 characters
varying[*n*]	Varies up to *n* bytes	0 to 32767 characters
date	64 bits	1 January 100 to 11 December 5941
blob	Varies	None

Qli supplies a segment length of 40 for blobs. If this length is not adequate, use **gdef** or modify the system relation directly.

scale-clause
the power of 10 by which InterBase multiplies the stored integer value for use by **qli**, COBOL, and PL/I.

For example, a negative scale of two means that there should be a decimal point two places to the left of the rightmost digit (that is, in the normal format for dollars and cents).

Example

The following statements define fields with various datatypes:

```
QLI>  define field tolerance long scale -2

QLI>  define relation parts
CON>  item_code char[6],
CON>  item_name char[25],
CON>  manufacturer char[10],
CON>  blurb blob,
CON>  price long,
CON>  tolerance
```

⇓

Edit String Clause

Function

The **edit string** clause specifies an alphabetic, numeric, or date format for a field.

Syntax

```
edit_string "edit-character..."
edit_character::= see Tables 3-6, 3-7, and 3-8
```

edit-character

The following tables list available edit strings by datatype. **Qli** also supports a missing value edit string: see the discussion of Formatting Value Expressions inthe *Qli Guide* for more information.

Table 3-6. Alphabetic and Miscellaneous Edit String Characters

Character	Meaning of Edit String
A (*integer*)	Any alphabetic character. For example, "aaabxxba" yields "HAL 14L" from the value "HAL14L." **Qli** returns an error if the characters are not alphabetic.
x(*integer*)	Any printing character, including special characters. See "A" above for an example.
B(*integer*)	A blank space. See "A" above for an example.
-	A hyphen. For example "aaa-xx-a" yields "HAL-14L" from "HAL14L."
'string' or "string"	Print the quoted string. For example, print "1234" using 9"abc"999 prints out "1abc234."

Table 3-7. Numeric Edit String Characters

Character	Meaning of Edit String Character
9(*integer*)	An ordinary digit. For example, "9999.99" yields "0832.79" for the value "83279" if it has a scale of -2. With an integer value of scale 0, it prints a numeric overflow.
.	Decimal point. See "9" above for an example.
B(*integer*)	Blank space.
,	Comma for thousands, millions, etc. For example, "99,999.99" yields "65,832.79" from the value 6583279.
z(*integer*)	A leading digit or blank if the leading position is zero.
+	Leading plus sign. Prints leading sign for positive and negative numbers. This sign takes up one character space.
-	Leading minus sign. Prints leading sign for negative numbers only. The sign takes up one character space.
$	Leading dollar sign. Multiple dollars sign "floate" so and edit string of "$$$$$$.99" yields "$123.45," "$12.34" for "12.34," and "$1234.56" for "1234.56." This sign takes up one character space.
*	A leading asterisk (for checks). This sign takes up one character space.
H(*integer*)	Hexadecimal representation of a character.
(())	Parentheses to print around negative numbers.
DB	Prints DB for debit after negative numbers.
CR	Prints CR for credit after negative numbers.
""	Quoted strings can be include in a number. For example, *print* 37.95 using $$99.99b"tsk!btsk!" prints "$37.95 tsk! tsk!"

Table 3-8. Date Edit String Characters

Character	Meading of Edit String Character
Y(*integer*)	The year, from right to left. For example, the field value "1987", "y(1)" yields "7" and "y(2)" yields "87."
M(*integer*)	The name of the month. The integer specifies how many of thecharacters in the month name to print. For example, "m(3)"yields "Jan."
N(*integer*)	The numeric month. The best value for the integer is 2. For example, "n(2)" yields "01" for January and "11" for November.
D(*integer*)	The day of the month. The best value for the integer is 2. For example, "d(2)" yields "01" and "11" for the first and eleventhdays of the month.
W(*integer*)	The name of the day of the week. The integer specifies how many of the characters in the day name to print. Theminimum value for the integer should be 2. For example, "w(2)" yields "Mo."
B	A blank space. A "b" in a date edit string enchances readability. For examle, "d(2)bm(3)by(4)" yields dates formattedas "29 May 1956."
J(*integer*)	The Julian day of the year. For example, "01" is 1 January and "32" is 1 February.
T(*integer*)	The time portion of a date field. Unless you append the P editstring discussed below, the time is based on a 24-hour clock.You must supply your own punctuation. For example,"t(2):t(2):t(2)" yields time formatted as "14:23:31."
P[P]	Changes the T time display to a 12-hour clock followed by themeridian value (ante or post). If you specify only one P, itdisplay A or P. Otherwise it displays AM or PM. You mustsupply your own punctuation. For example, tt:tt:bpp" yields time formatted as "2:23:31 PM.
X(*integer*)	The date and time portion of a date field, based on a 24-hour clock. Qli supplies the punctuation. For example, "x(25)" yields date formatted as 23-SEP-1987 15:38:11.9721.

Example

The following commands define a database and three relations, each containing several fields with edit strings:

```
QLI>  define database "stuff.gdb"
QLI>  define relation budgets
CON>  b1 long,
CON>  b2 long edit_string "999,999",
```

```
CON>  b3 long edit_string "((999,999))",
CON>  b4 long edit_string "-ZZZ,ZZZ,ZZ9"

QLI>  define relation employee_stuff
CON>  social_security char [9] edit_string -
CON>  "xxx-xx-xxxx",
CON>  phone_number char [10] edit_string -
CON>  "(xxx)Bxxx-xxxx",
CON>  salary long edit_string -
CON>  "HHHHHHHHHBBB"

QLI>  define relation family_dates
CON>  name varying [10],
CON>  birth date edit_string -
CON> "w(3),bd(2)bm(12)by(4)",
CON>  wedding date edit_string "d(2)bn(2)by(4)",
CON>  awareness date edit_string "y(4)"
```

Query Name Clause

Function

The **query name** clause provides an alternate field name for use in **qli**. You can reference a field by its full name or by the query name.

You may find that the longer the name, the more likely interactive users are to mistype it. However, for reasons of internal documentation, you might want to keep the name as descriptive as possible. Therefore, you can use a query name to rename the field to something easier to type.

Syntax

```
query_name[is] alternate-name
```

alternate-name
A query name can contain up to 31 characters that can be alphanumeric, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

Examples

The following statement defines a field with a query name:

```
QLI>  define field longitude_degrees char[2] -
CON>  query_name longd
```

The following statement defines a relation and assigns a query name to six fields:

```
QLI> define relation cities
CON> city,
CON> state,
CON> population,
CON> latitude_degrees char[2] -
CON> query_name latd,
CON> latitude_minutes char[3] -
CON> query_name latm,
CON> latitude_compass char[1] -
CON> query_name latd,
CON> longitude_degrees char[2] -
CON> query_name longd,
CON> longitude_minutes char[2] -
CON> query_name longm,
CON> longitude_compass char[2] -
CON> query_name longc
```

Troubleshooting

See the discussion of errors and error handling in Chapter 1 of the *Qli Guide*.

See Also

See the entries in this chapter for:

- **define field**
- **define relation**
- **define view**
- **modify field**
- **modify relation**

Finish

Function

The **finish** command explicitly closes a database.

If you close a database and want to access it later, you must ready it again.

Syntax

```
finish [database-handle-commalist]
```

database-handle
Specifies the database to close. If you do *not* specify a database handle, the **finish** command commits all default transactions and closes *all* open databases.

If you close a specific database, InterBase commits the default transaction for that database.

If you neglected to declare a database handle when you opened the database, you can use the default handle declared by **qli**. Use the **show databases** command to display the name of the handle that **qli** declared for the database.

Examples

The following command closes all open databases:

```
QLI> finish
```

The following example readies two databases, performs some data manipulation, and closes one of the databases:

```
QLI> ready /usr/case/databases/atlas.gdb as atlas
QLI> ready maps.gdb as map
⇓
QLI> finish atlas
```

Troubleshooting

You may encounter the following message when you use the **finish** command:

Expected database handle, encountered <string>

You need a database handle. You may have mistyped the handle. Type *show databases* to check the database handle.

See Also

See the entries in this chapter for:

- **ready**
- **commit**
- **rollback**

For

Function

The **for** statement evaluates a record selection expression and executes a substatement for each qualifying record.

You can nest **for** loops to display a hierarchy of records or to join relations across databases.

Syntax

```
for rse qli-statement
```

Options

rse
Provides the record selection criteria to form a record stream.

qli-statement
Any of the **qli** statements or any procedure containing statements.You cannot use a command or a procedure containing commands as the action of an **if-else** statement.

Examples

The following example creates a record stream **for** loop and displays records from that stream:

```
QLI>  for states sorted by state
CON>  print capital, state, statehood
```

The following example joins two relations:

```
QLI>  for states cross cities over state sorted -
CON>  by city
CON>  print city, state, altitude, population
```

The following example uses a **for** loop to select records to be erased and then erases them:

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

The following example picks up a value from an outer loop, prints it, and then prints associated values from another relation in an inner loop:

```
QLI>  for r in rivers sorted by river
CON>  begin
CON>  print river
CON>  for rs in river_states with -
```

```
CON>  r.river = rs.river
CON>  print state
CON>  end
```

The following example is equivalent to a join operation across databases:

```
QLI>  ready apollo:/usr/data/mapper.gdb as mapper
QLI>  ready atlas.gdb as atlas
QLI>  for s in atlas.states sorted by s.state
CON>  begin
CON>  for c in mapper.cities with
CON>  s.state = c.state
CON>  print s.state_name, c.city, c.population
CON>  end
```

Note

You cannot reference relations from more than one database in a record selection expression. Use **for** loops to combine relations across databases.

Troubleshooting

You may encounter the following message when you use the **for** statement:

- *Relations from multiple databases in single rse*
- *Can't mix databases within RSE*

  You tried to access more than one database in the same record selection expression. Use nested **for** statements to do that.

See Also

See the entry in Chapter 2 for record selection expression.

For Form

Function

The **for form** statement specifies that a form be used to accept or display data.

Syntax

```
for form [context-variable in] form-name
 qli-statement
```

Options

context-variable
The context variable qualifies references to the form fields to distinguish them from database fields or program variables.

form-name
Specifies the form to bind. The form name must be the name of a form already defined in a database. If you include a database handle, the form must be in that database. Otherwise, **qli** searches databases referenced by the program, beginning with the most recently declared database.

qli-statement
Any of the **qli** statements or any procedure containing statements.You cannot use a command or a procedure containing commands as the action of an **if-else** statement.

The *qli-statement* can be any **qli** statement, but most often is a begin-end block. It usually contains one or more **accept** statements or form field assignments.

Example

The following example displays a form to accept the input of a state code, and then displays a form to data from cities in that state:

```
QLI> for form f in cities
CON> begin
CON> accept ("Enter state code,
CON>  then <enter>") state
CON>  for c in cities with c.state = f.state
CON>  begin
CON>  f.state = c.state
CON>  f.city = c.city
CON>  f.altitude = c.altitude
CON>  f.latitude = c.latitude
```

```
CON>  f.longitude = c.longitude
CON>  accept ("Hit <enter> to continue
CON>  or <f1> to stop")
CON>  end
CON> end
```

Troubleshooting

You may encounter the following messages when you use the **for form** statement:

Form <form-name> is not defined in database "file-spec"

The form name you specified does not exist in the database. Check the spelling of the form name and try again.

See Also

See the chapter on using forms with GDML in the *Forms Guide*.

See also the entries in this chapter for:

- **accept**
- **for form**
- **for menu**

Grant

Function

The SQL **grant** command defines user privileges for designated tables and views. It can also grant a user the ability to pass along privileges. A relation's owner is the only user to have automatic grant authority for that relation. To pass the ability to grant privileges to a user, the **grant** command must contain the **with grant option** clause.

Syntax

```
grant privilege-comma-list on relation-name|view-
name to user [with grant option]
privilege::= {all [privileges]|select|delete|
insert|update (column-list)}
user::=public|userid-comma-list
```

Options

privilege-comma-list
Allows user to specify the following operations:

Privilege	Authority
All	Selects, deletes, inserts, updates
Select	Retrieves records from table or view
Delete	Eliminates records from table or view
Insert	Stores new records in table or view
Update	Changes value of one or more fields in existing table or view

relation-name
Specifies the relation to which you assign privileges

view-name
Specifies the view to which you assign privileges. Because a view is only a window into one or more base relations, you can never grant a user more privileges on a view than that user has to the base relation or relations.

public|*userid*
Specifes which authorized users have access to privileges for a table or view. **Public** incorporates all authorized user ids.

with grant option
Passes grant authority along to the user(s) specified in the **grant** statement. This is valid for only those privileges authorized in the grant statement.

Usage

Once you have secured a table using SQL, you should use only SQL to further secure it. Do not use the InterBase security class system in combination with SQL security.

Examples

The following example grants select and delete privileges to a user and gives that user the authority to grant other users select and delete privileges:

```
QLI>  grant select, delete on cities to julie with
CON>  grant option;
```

The following example grants update privileges to a user for specific fields in a relation:

```
QLI>  grant update state_name, capital on states to
CON>  john;
```

Troubleshooting

You may encounter the following errors when you use the **grant** command:

- *expected relation name, encountered "string"*

  You typed the relation name incorrectly.

- *expected on, encountered "string"*

  You typed a privilege incorrectly.

- *QLI error from database "filename"*
  unsuccessful metadata update
  -STORE RED$USER_PRIVILEGES failed on gran
  -action cancelled by trigger (1) to preserve data integrity

  You do not have privilege to grant the privilege or privileges you tried to grant.

See Also

See the entry for revoke in this chapter.

Help

Function

The **help** command provides assistance on **qli** commands and statements. If you ask for help without specifying a command or statement, **qli** displays a listing of what help is available. If you ask for help on a subject for which there is no assistance, **qli** tells you that no help is available for that subject.

InterBase's help is structured hierarchically, so you may not find what you are looking for on the first try. For example, suppose you want to look at the help entry for the *arithmetic-expression* value expression. There is no entry for this topic, but there is one for *value_expression*. When you ask for help on value expressions, the text notes that there is additional help available for *arithmetic_expression*.

Note

The first time you ask for help, there may be a slight delay as InterBase readies the database containing the help topics.

Syntax

```
help [qli command | qli-statement]
```

Options

qli-command
Specifies the **qli** command for which you want help.

qli-statement
Specifies the **qli** statement for which you want help.

Usage

You can edit the help files, add new topics, or delete existing ones. For example, you may want to replace frivolous examples with ones more closely tied to your application, or document procedures that you want everyone to use.

To edit the help files, add new topics, or delete existing topics, invoke **qli** and ready the help database:

Operating System	Pathname
VMS	sys$help:help.gdb
Apollo	/interbase/help/help.gdb

Operating System	Pathname
UNIX	/usr/interbase/help/help.gdb

Use the **show** commands to see the record structure of the help database:

```
QLI> show fields
      Database "help.gdb"
      TOPICS
      TOPIC        text, length 31
      PARENT       text, length 31
      FACILITY     text, length 6
      SYSTEM_FLAG  text, length 1
      TEXT         blob, segment length 80
```

The database help.gdb contains all the help topics. Now that you have readied the help database, you can manipulate it as you would any database.

To modify a topic, select the record with an RSE and change field values. For example:

```
QLI> modify text of topics with topic = STORE
```

Qli calls your default editor. Make the changes you want, then exit from the editor in the normal manner.

To store a new topic, use the **store** or **insert** statement. For example:

```
QLI> store topics
Enter TOPIC: LJUBJANKA
Enter SYSTEM_FLAG: X
Enter FACILITY: NKVD
```

Standard InterBase help topics have a system flag of "S"; your messages should use some other flag value. You should back up the help library before installing new versions of InterBase, so that you can reintegrate your changes.

Use the **erase** statement to remove unwanted topics:

```
QLI> for topics with topic = SELECT
CON> erase
```

As you enter new topics and modify existing ones, you should pay attention to the form of the TOPIC name. Whenever someone asks for help on a particular subject, the help facility in **qli** searches through TOPICS for a match on the TOPIC field. multiple word topics cause problems in matching, so use underscores between them.

You can structure a hierarchy for your entries. All **qli** statements and many commands have a value of "QLI" for the PARENT field. The expressions (value, record selection, selection, and scalar) also have a value of "QLI." However, the actual expressions, such as arithmetic expression, have a value of "QLI VALUE_EXPRESSION" for the PARENT field. If there were a subordinate entry for arithmetic expression, its parent would be "QLI VALUE_EXPRESSION ARITHMETIC_EXPRESSION." When you add your own entries, or update the standard ones, plan its parenthood well.

Example

The following command displays the general help listing:

```
QLI> help
```

The following command displays help about the **store** statement:

```
QLI> help store
```

Troubleshooting

You may encounter the following message when you use the **help** command:

No help is available for "subject."

The subject for which you requested help does not exist. Type help for a list of topics.

See Also

See the entries in this chapter for:

- **modify**
- **store**
- **erase**

If-Else

Function

The **if-else** statement provides an if-then-else structure in **qli**. The **then** in the syntax is optional.

Syntax

```
if boolean-expression [then] qli-statement
[else qli-statement]
```

Options

qli-statement

Any of the **qli** statements or any procedure containing statements.You cannot use a command or a procedure containing commands as the action of an **if-else** statement.

If there is more than one action you want to execute as a result of the **if**, put the statements into a **begin-end** statement.

If there is an **else**, **qli** requires that you place a hyphen after the **end** or that you place the **else** on the same line as the **end**. For example:

```
if expression
   begin
   qli-statement
   qli-statement
end else
begin
   qli-statement
   qli-statement
end
```

Because the **else** clause is optional, a statement of the form **if** *boolean-expression* [**then**] *qli-statement* is complete. The **else** *qli-statement* that follows is read as a new statement.

boolean-expression

Specifies a condition that must be true in order for the **if** to be executed. If the condition is not true, the **else** branch is executed.

Examples

The following statement is a simple case of an **if-then** construct:

```
QLI> for rivers
CON> begin
CON> print
CON> if length > average length of rivers
CON> print col 5, "At " | length |
CON> " miles it's longer than your average river."
CON> end
```

The following statement uses nested for loops to walk through the states relation looking for cities with fewer than a million residents. Nested **if** statements cause it to produce an appropriate message if there are 0, 1, or more small cities. For each small city, another set of nested **if**s print different messages for small, very small, very small indeed, and totally negligible cities:

```
QLI> for states sorted by state_name
CON> begin
CON> declare counter long;
CON> counter = count of cities over state with
population < 1000000
CON> if counter = 0
CON> print skip,
CON> state_name | " has no small cities." else
CON> if counter = 1
CON> print skip,
CON> col 1, state_name
CON> | " has one pretty small city." else
CON> print skip, col 1,
CON> state_name | " has some pretty small cities.
CON> For example: "
CON> for cities with population < 1000000 and
CON> state = states.state sorted by population
CON> if population < 10000 then
CON> print col 5,
CON> city | " is particularly dinky." else
CON> if population < 100000 then
CON> print col 5,
CON> city | " is pretty small." else
CON> if population < 500000 then
CON> print col 5,
CON> city | " is a biggish small city." else
CON> print col 5,
```

```
CON> city | " is quite a big small city."
CON> end
```

The output follows:

```
Alabama has some pretty small cities. For example:
  Montgomery is a biggish small city.
  Birmingham is a biggish small city.

Alaska has one pretty small city.
  Juneau is particularly dinky.

Arizona has one pretty small city.
  Phoenix is quite a big small city.

Arkansas has one pretty small city.
  Little Rock is a biggish small city.

California has some pretty small cities. For
example:
  Fresno is a biggish small city.
  Sacramento is a biggish small city.
  San Francisco is quite a big small city.
  San Diego is quite a big small city.

Colorado has one pretty small city.
  Denver is a biggish small city.

Connecticut has no small cities.

Delaware has no small cities.
   ⇓
```

The following statement cleans 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. The **repeat** loop gives the user a second (through fiftieth) chance when the choice is out of bounds:

```
QLI> begin
CON> declare updchar [1];
CON> print skip, col 1, "Cleanup our cities."
CON> print skip, col 1,
CON> "Type D to delete the city, M to change its
CON> population,", skip,
CON> "or L to leave it alone"
CON> for cities with population missing
```

```
CON> begin
CON> print skip
CON> upd = "X"
CON> repeat 50
CON> begin
CON> if upd not in ("D", "M", "L") then
CON> begin
CON> print col 1,
CON> "What do you want to do to " | city | " " |
CON> state | "?"
CON> upd = *."D[elete] / M[odify] / L[eave it be]"
CON> if upd = "D"
CON>        begin
CON> print col 1, city | " " | state | " is gone"
CON> erase
CON> end else
CON> if upd = "M"
CON> modify using
CON> population = *."new value for population" else
CON> if upd = "L"
CON> print col 1, "ok by me" else
CON> print "Bad guess. Try again"
CON> end
CON> end
CON> end
CON> end
```

The output follows:

```
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]: l
     Bad guess. Try again
What do you want to do to Springfield IL?
⇓
```

Troubleshooting

You may encounter the following message when you use the **if-else** statement:

- *Expected statement, encountered "command"*

  You cannot use a command with the **if-else** statement.

See Also

Any of the qli statements in this chapter.

Insert

Function

The **insert** statement that stores a new record into a relation.

If you are storing a record that contains a blob field, you cannot use the **insert** statement to assign a value other than null to a blob field. To store records with blob fields, use the GDML **store** statement.

Syntax

```
insert into relation-name [(database-field-
commalist)]
{values constant-commalist|select-statement}
```

Options

relation-name

Specifies the relation into which you want to store a new record.

You can insert into a view comprised of a single relation, or into a view with a *store* trigger.

database-field

Lists the field in *relation-name* for which you are providing a value. If you do not provide an insertion list, **qli** assumes that you want to store all fields in the record and stores them in their default order. The default order is based on the value of the RDB$FIELD_POSITION field in the RDB$RELATION_-FIELDS system relation. See the *Data Definition Guide* for more information about system relations.

If you want to store the missing value for a field, do not reference that field in the **insert** statement.

If the database field is a blob, you can only assign the **null** value.

constant

Provides a value for *database-field*. You can assign field values by inserting quoted strings, and quoted or unquoted numbers.

select-statement

Specifies that the values for the new record are to come from the record identified by a **select** statement.

Examples

The following statement inserts quoted values:

```
QLI> insert into ski_areas (name, type, city,
CON> state)
CON> values ('Radar Acres', 'N', 'Dunstable',
CON> 'MA');
```

The following example stores a new record into CITIES, using most of the values from an existing record:

```
QLI> insert into cities-
CON> (city, state, latitude_degrees,
CON> latitude_minutes,
CON> latitude_compass, longitude_degrees,
CON> longitude_minutes-
CON> longitude_compass)
CON> select 'Troy', state, latitude_degrees,
CON> latitude_minutes,
CON> latitude_compass, longitude_degrees,
CON> longitude_minutes,
CON> longitude_compass-
CON> from cities where city = 'Albany' and
CON> state = 'NY'
```

The following statement stores a new record into CITIES, implicitly assigning the missing value to all unreferenced fields:

```
QLI> insert into cities-
CON> (city, state)-
CON> values ('Lowell', 'MA');
```

The following statement stores a new record into TOURISM, but does not reference the blob fields OFFICE or GUIDEBOOK, thereby assigning the missing value to those fields:

```
QLI> insert into tourism-
CON> (state, zip, city)-
CON> values ('NY', '10022', 'New York');
```

Troubleshooting

You may encounter the followint error when you use the **insert** statement:

***QLI error: the number of values did not match the number of fields*

Your value list or **select** did not have the same number of entries as the field list.

See the error list in the entry for the *assignment* statement.

See Also

See the entries in this chapter for:

- **select**
- **store**

List

Function

The **list** statement displays fields from records in a record stream. Unlike the **print** statement, it displays the field values in a vertical format.

Syntax

Standalone format

```
list value-expression-commalist of rse
[on 'filespec' | to shell-command]
```

Loop Syntax:

```
for rse
list value-expression-commalist
```

Options

value-expression **of** *rse*
Specifies a list of fields or other values from the record stream created by the record selection expression.

on *'filespec'*
Sends the output to the named, quoted file, rather than writing it to your monitor.

to *shell-command*
Sends the output to standard input of the shell or command interpreter command, rather than writing it to your screen. These commands typically send the output to a printer, as in **print**, **lpr**, **lpt**, **prf**, and **'prf -npag'**. Note that if you include a switch on the shell command, you must quote the entire command.

Examples

The following query lists all records in STATES:

```
QLI> list states
```

The following query includes a **for** loop that selects records:

```
QLI> for states with area lt 10000
CON> list state_name, area
```

The following query writes field values from STATES to the file *state_data.dat*:

```
QLI> list state, capital, area of states on
CON> 'state_data.dat'
```

Troubleshooting

You may encounter the following messages when you use the **list** statement:

- *No items in print list*

  You must provide a record selection expression or value expression.

- *Can't open output file.*

  Qli cannot open an output file for a *print on filespec* command.

See Also

See the entry in this chapter for **print**.

Modify

Function

The **modify** statement updates a field or fields in a record or records.

You can modify records from a view if the view is comprised of a single relation (without a reflexive join) or if the database designer included a *modify* trigger.

Syntax

Standalone format:

```
modify dbfield-expression-commalist of rse
```

Substatement format:

```
modify{dbfield-expression-commalist|using
statement}[of rse]
```

Form format:

```
modify using form[form-name]
```

Options

dbfield-expression
Specifies the field you want to update. **Qli** prompts you for a field value. If you choose the using option, you must supply assignment statements.

statement
Ordinarily an assignment to a field or a begin-end block containing assignments. However, a begin-end block in this position can contain any type of statement.

If you want to modify a blob field, use either the edit option of the assignment statement or **qli**'s prompting feature.

rse
Specifies record selection criteria.

Unless you use the standalone format, you must enclose the modify command in a for loop that contains a record selection expression.

Do not modify records whose record selection expression includes a reduced to clause.

using form [*form-name*]
Tells **qli** to use a form for assignments instead of statements. This form of the modify statement can be used only inside a for statement. If you do not supply a form name, **qli** looks for a form with the same name as the relation. Failing to find that, it returns an error.

Examples

The following statement modifies river lengths:

```
QLI> for rivers
CON> modify using begin
CON> print river
CON> length = *.length
CON> end
```

The following statements change the same record using the **modify** statement in different ways:

```
QLI> /* rse in modify statement */
QLI> modify population of cities with
CON> city = 'New York'
Enter POPULATION: 10000000

QLI> /* rse in for statement */
QLI> for cities with city = 'New York'
CON> modify using population = 10000000

QLI> modify population of cities with
CON> city = 'New York'
Enter POPULATION: 10000000
```

Troubleshooting

See the discussion of errors and error handling in Chapter 1 of the *Qli Guide*.

See Also

See the entries in this chapter for:

- **assignment**
- **for**
- **begin-end**

Modify Field

Function

The **modify field** command changes the attributes of a global field.

Syntax

```
modify field field-name field-attributes
```

Options

field-name
Specifies the global field you want to change.

field-attributes
Specifies the field's datatype, query name, or edit string. The datatype specification must precede other field attributes. See the entry in this chapter for *field attributes* for more information.

Example

The following example modifies a global field:

```
QLI> modify field party_affiliation query_name
CON> "party"
```

Troubleshooting

You may encounter the following message when you use the **modify field** command:

Expected field definition clause, encountered "string"

You included an unrecognized attribute in the definition. If the former case is true, change the field name so it starts with an alphabetic character. If the latter case is true, check the command and make sure that you have included a valid attribute.

If you modify the datatype of a field to or from a blob, you will receive a conversion error when you try to print the field.

See Also

See the entry in this chapter for field attributes.

Modify Index

Function

The **modify index** command changes the uniqueness, activeness, or order of an index. If you want to add or drop fields from an index, you must delete the index and then redefine it.

Syntax

```
modify index index-name
[unique|duplicate]
[active|inactive]
[ascending|descending]
```

Options

index-name
Specifies the index you want to modify.

unique
Changes an index that allows duplicate index values to one that does not.

If you make the index unique, you will receive errors during index creation if there are duplicate keys. Before defining a unique index, or modifying an index to be unique, find duplicate keys. You can use a statement such as the following to do so:

```
QLI> declare x based_on relation.name.key
CON> x = 0
CON> for relation.name sorted by key
CON> begin
CON> if key = x print key
CON> key = x
CON> end
```

For large relations, creating an index with duplicates will significantly reduce the time this winnowing process takes.

duplicate
Changes an index that disallows duplicates to one that allows them.

active
Active changes an inactive index to an active one.

inactive
Inactive changes an active index to an inactive one.

Because InterBase automatically maintains all indexes, you may want to change an index to inactive if you are going to store many records at one time. Once you have stored all your records, reactivate the index.

ascending
Changes an index to ascending.

descending
Changes an index to descending.

Examples

The following statements modify indexes:

```
QLI> modify index state_idx1 duplicate inactive

QLI> modify index river_idx_1 ascending
```

Troubleshooting

You may encounter the following message when you use the **modify index** command:

Expected index state option, encountered "string"

The **modify index** command lets you change two characteristics of an index. You specified something other than the supported options. Check your command and try again.

Because the **modify index** command rebuilds the index, its execution may take a few minutes. If another user is running a request that relies on the index, you cannot rebuild the index until that request completes.

See Also

See the entry in this chapter for **define index**.

Modify Relation

Function

The **modify relation** command can change a relation's complement of fields and local field characteristics.

Syntax

```
modify relation relation-name operation-commalist
operation::={add field field-name
[field-attributes]|drop field field-name|
modify field field-name [field-attributes]
```

Options

relation-name
Identifies the relation you want to modify.

add field *field-name*
Adds a field to the relation:

- If the field has already been defined in the database, *field-attributes* can specify an edit string or a query name.
- If the field does not exist elsewhere, you must specify a datatype. You can also specify an edit string or query name.

The addition of fields to a relation is identical to the inclusion of fields when you define the relation. See the entry for **define relation** in this chapter for more information.

drop field *field-name*
Removes the named field from the relation. When you delete a field from a relation, other users should not encounter any problems if they are already running their programs. However, if they start up a program that references the deleted field, the program fails when it tries to compile the request that mentions that field.

You cannot delete fields that are used in views based on this relation without first deleting the field from those views.

modify field *field-name*
Identifies the field whose relation-specific characteristics you want to change. You can change only the edit string and query name. You cannot change the datatype.

Example

The following example modifies a relation by adding fields, dropping fields, and modifying fields:

```
QLI> ready test_atlas.gdb
QLI> modify relation cities
CON> add field year_incorporated char[4]
CON> query_name inc,
CON> add field type_of_government char[1]
CON> query_name gov,
CON> drop field population
```

Troubleshooting

You may encounter the following message when you use the **modify relation** command:

Expected field definition clause, encountered "string"

You specified a field name began with a non-alphabetic character or included an unrecognized attribute in the definition. If the former case is true, change the field name so it starts with an alphabetic character. If the latter case is true, check the command and make sure that you have included a valid attributed.

See Also

See the entry in this chapter for **define relation**.

Prepare

Function

The **prepare** command signals your intention to commit the default transaction. InterBase automatically issues a **prepare** for **qli** sessions that involve multiple databases.

The **prepare** command is particularly useful for sessions that access multiple databases or require coordination with external events. It executes the first phase of a two-phase commit. The InterBase access method polls all participants and waits for replies from each. It checks to see that no other database activity can affect the transaction. If the statement completes successfully, InterBase guarantees that a **commit** statement will execute successfully if the disk is still intact.

Syntax

```
prepare [database-handle-commalist]
```

Options

database-handle

Specifies a name you assign to a database when you ready it. Use the handle to qualify database reference when you are using multiple databases.

A **prepare** command without the optional database handle prepares all open databases. If you assign a database handle when you ready the database, you can use the handle to limit the scope of the **prepare** to specific databases.

When you access more than one database in **qli**, InterBase automatically starts up separate subtransactions for each database. However, these appear to be a single transaction. The optional database handle lets you control these subtransactions explicitly by letting you prepare them by database.

Qli automatically assigns a default handle if you forgot to assign a database handle when you ready the database.

Type the following to find out the default database handle assigned by **qli**:

```
QLI> show databases
Database "atlas.gdb" readied as QLI_0
```

Qli displays the names of all available entities, including databases and handles. The default handles are in the form "QLI_n," where *n* is a numeric integer. You can supply this handle as an argument to the **prepare** command:

```
QLI> prepare qli_1
```

Example

The following statements ready several databases, perform some unspecified data manipulation, prepare to commit the transaction, and then commit the transaction:

```
QLI> ready remote_database_1.gdb
QLI> ready local_database.gdb
QLI> ready remote_database_2.gdb
       ⇓
QLI> prepare
QLI> commit
```

Troubleshooting

You may encounter the following message when you use the **prepare** statement:

Expected database handle, encountered <string>

You need a database handle. You may have mistyped the handle. Type *show databases* to check the database handle.

See Also

See the entry in this chapter for **commit**.

See the *OSRI Guide* for information on the internal operations of the **prepare** command.

Print

Function

The **print** statement displays fields from records in a record stream. You can create the record stream in the **print** statement itself or in an outer **for** statement. You can also display records using forms.

Syntax

Print format:

```
print print-list of rse[on 'filespec'|to shell-
command]
print-list::=[distinct]print-element-commalist
print-element::= {format-token|qualified-value}
format-token::={space [integer]|skip [integer]|
tab [integer]|col integer|new_page}
qualified-value::=value-expression [(query-
header|-)][using edit-string]
query-header::=quoted-string[/quoted-string]
```

For ... print format:

```
for rse print print-list
 [on 'filespec'|to shell-command]
```

Forms **print** statement format:

```
print rse using form [form-name]
```

Forms **for** ... **print** statement format:

```
for rse print using form [form-name]
```

Options

distinct
Prints only unique values or combinations of values specified in the print list. The unique values are created through a project relational operation. For information about the project operation, see the description of the reduce clause in the entry for RSE.

space [*integer*]
Inserts one blank horizontal space in the output file. If you specify the optional integer, **qli** inserts that many blank spaces.

skip [*integer*]
Inserts a carriage return and line feed. If you specify the optional integer, **qli** inserts that many blank lines.

tab [*integer*]
Inserts one horizontal tab into the output line. If you specify the optional integer, **qli** inserts that many tabs.

col *integer*
Begins the next print element in the specified column. If the number of columns you specify is less than the current column number, it inserts a carriage return and line feed.

new_page
Inserts a page break into the output file.

value-expression
Specifies the field, variable, quoted string, arithmetic expression, statistical expression, or other value to be printed. If the value expression is or includes a database field name, you must supply a record selection expression when you use the **print** statement , and **qli** must be able to resolve the field in the context of the *rse*. For more information about value expressions, see the entry in this chapter for *value-expression*.

The value expression can be of the form *context-variable*.*, so that you can request all the fields from a particular relation without having to list them all.

Qli accepts the word **and** as a substitute for a comma in a list of value expressions. Therefore, *print city, state, population of cities*, *print city, state and population of cities*, and *print city and state and population of cities* are equivalent.

query-header
Specifies a column header. By default, **qli** uses the database field name as a column header when it returns values. If you include a value expression that is not a field, or if you want to give the column a different header for its output, you may specify the query header in the print list. To create a multi-line query header, separate each line in the header with slashes.

For example, "Name" / "of" / "Field" stacks the words "Name," "of," and "Field" on three separate lines at the top of the column. To suppress default column header, specify an unquoted hyphen.

edit-string
Specifies an output format for the value expression. **Qli** forces the value expression into the specified format if possible. For example, *print "today"* prints the string "today," but *print "today" using w(9)* prints the day of the week.

See Tables 3-4 through 3-6 in the *edit-string* section of the *field-attributes* entry. These three tables present alphabetic, numeric, and date edit strings. In all cases where the token *integer* appears, you can substitute a repetition of the edit string character. For example, *hhhh* and *h(4)* are equivalent.

using form [*form-name*]
Tells **qli** to use a form to display records from a relation. If you do not provide a form name, **qli** looks for a form with the same name as the relation in the *rse*. If it can't find one, **qli** returns an error.

Examples

The following query prints all records in the STATES relation:

```
QLI> /* simple print format */
QLI> print states sorted by state_name
```

The following query prints a literal value expression:

```
QLI> print "Jean, approach ramming speed."
Jean, approach ramming speed.
```

The following query prints the number of populated cities and uses a query header:

```
QLI> print count of cities with
CON> population > 0 ("Populated"/"Cities"
```

The following query prints whatever you ask it to print:

```
QLI> print *."whatever your heart desires"
Enter whatever your heart desires: chocolate
chocolate
```

The following query writes field values from the STATES and SKI_AREAS to the file shush_boom.dat:

```
QLI> print state_name, name, city of states cross
CON> ski_areas over state on 'shush_boom.dat'
```

The following query prints field values from records in a stream created by a **for** command:

```
QLI> for states cross ski_areas over state
CON> print state_name, name, city
```

The following query prints the hexadecimal representation of Albany's altitude:

```
QLI> print altitude using hhhh of cities with
CON> city = 'Albany'

 ALTITUDE
===========
   3b
```

The following query prints today's date using an edit string:

```
QLI> print "today" using w(8)" the "dd"th
CON> of "m(12)" in the year "y(4)
Thursday the 15th of May in the year 1986
```

The following query prints the POPULATION field from CITIES using an edit string to format the number:

```
QLI> print city, population using z,zzz,zz9 of CON>
cities

            CITY                      POPULATION
============================      ==========

Juneau                                 7,000
Indianapolis                         700,000
Montgomery                           177,807
   ⇓                                    ⇓
```

Troubleshooting

You may encounter the following messages when you use the **print** statement:

- *No items in print list*

  You must provide a record selection expression or value expression.

- *Can't open output file*

  Qli cannot open an output file for a *print on filespec* command.

See Also

See the entries in this chapter for:

- **for**
- **list**

Quit

Function

The **quit** command prompts you to either roll back or commit any updates to databases when you leave **qli**.

If you have made any updates since your last **commit** or **rollback** command, **qli** asks if you want to roll back your changes. If you answer "y" (yes), **qli** undoes any changes you made since your last **commit** and then exits. If you answer "n" (no), it commits the changes and then exits.

If you have not many any changes, **qli** exits without prompting you.

If you use a **quit** command inside a command file and do not include a "y" or "n" following the **quit**, **qli** rolls back your changes.

Syntax

```
quit
```

Example

The following **qli** session updates the database, quits, and rolls back the changes:

```
% qli
Welcome to QLI
Query Language Interpreter
QLI> ready atlas.gdb
QLI> store river_states
Enter STATE: MA
Enter RIVER: Connecticut
QLI> quit
Do you want to rollback your updates? y
```

The following **qli** session does not make any changes, so quitting results in an exit without prompting:

```
% qli
Welcome to QLI
Query Language Interpreter
QLI> ready atlas.gdb
QLI> print average length of rivers
```

```
 LENGTH
===========
   1081
QLI> quit
```

Troubleshooting

See the discussion of errors and error handling in Chapter 1 of the *Qli Guide*.

See Also

See the entries in this chapter for:

- **exit**
- **commit**
- **finish**

Ready

Function

The **ready** command attaches a database and opens it for access. This command must precede other database access in **qli**.

The **ready** command automatically starts a transaction that is not terminated until you commit it or roll it back. **Qli** automatically starts a new transaction with the next data manipulation statement that follows the **commit** or **rollback** command.

The **ready** command also opens the database for metadata update.

Syntax

```
ready filespec [as database-handle]
```

Options

filespec

Specifies the name of the file that contains the database. The file specification can contain the full pathname, including the name of the node on which the database is stored.

If the command language interpreter or shell from which you invoked **qli** is case-sensitive, make sure that you type the name of the database file exactly as it appears when you list the directory.

If you are in a directory other than the one that contains the database file, filespec must include the pathname. If the database is on another node, the filespec must include the node name and pathname. You can also define a link or logical name for the database file, and then reference it through either of those names.

File specifications for remote databases have the formats shown in the following table.

Table 3-9. Remote Database Access

From	To	Syntax
VMS	VMS via DECnet	node-name::filespec
VMS	ULTRIX via DECnet	node-name::filespec

Table 3-9. Remote Database Access

From	To	Syntax
VMS	non-VMS and non-ULTRIX	node-name^filespec
ULTRIX	VMS via DECnet	node-name::filespec
Apollo	Apollo	//node-name/filespec
Everything Else	Whatever is left	node-name:filespec

For example, the following command readies a database in the directory [*public.data*] on the VMS system *pariah*:

```
QLI> ready pariah:[public.data]phones.gdb
```

Make sure that what follows the colon is a valid file specification on the target system; use brackets, slashes, and spaces as appropriate.

`database-handle`

Specifies a name that can be used to qualify database references when you are using multiple databases. If you do not provide a handle, **qli** automatically assigns one of the form *qli_n*, where *n* represents a positive integer.

The optional *database-handle* lets you work with multiple databases, accessing each when you need it and closing each with a **finish** statement as appropriate. This approach saves system resources.

Usage

The database you access may be on another computer in the network. Such a database is called a *remote database*, and the computer where it is stored is called the *remote node*. The node you are using is called the *local node*. If the database you access is on the same node as you are, then it is a *local database*. To access a remote database, use the full network pathname of the database file or establish a logical link to it. Once you have readied the remote database, you can read and write records in the database as if it were local.

Example

The following example readies a database for access:

```
QLI> ready atlas.gdb
```

The following example readies two databases for access, states the explicit path to the database file for one database, and provides a database handle for each. The final line finishes one of the databases:

```
QLI> ready /usr/igor/datafiles/atlas.gdb as atlas
QLI> ready mailing_list.gdb as mailing
       ⇓
QLI> finish atlas
```

The following example readies a local and a remote database:

```
QLI> ready pariah::[doncikov.datafiles]atlas.gdb
QLI> ready mailing_list.gdb
```

Troubleshooting

InterBase may not be able to find the database file you think you want to ready. The database file might not exist anymore, might not have the name you specified, or might not be where you thought it was. In any of these cases, check the database file name and location.

InterBase may not be able to ready a remote database due to a communication problem with the remote node. If that is the problem, make sure the remote servers are running.

To use an InterBase database you need read and write access to the files. You may not have the right kind of access to the file. See your system administrator.

You may encounter the following message when you use the **ready** command:

- Operating system directive failed
  -no active servers (library/MBX manager)
  -communication error with journal "
  journal_directory_name"

  This message means that journaling has been enabled for the database you tried to ready, but no one has started the journal. Use **journal** to start the journal.

You may encounter the following message on an APOLLO:

- *Database error: I/O error during "ms_$mapl" operation for file "dbfile" -name not found (OS/naming server).*

The database you tried to ready does not exist where you thought it did, is unavailable for some reason, or does not exist at all. Check the pathname and try again.

See Also

See the entry in this chapter for **finish**.

Rename Procedure

Function

The **rename procedure** command changes the name of an existing procedure.

Syntax

```
rename procedure [database-handle.]old-name
[to] [database-handle.]new-name
```

Options

[database-handle.]old-name
Specifies the name of the procedure you want to change. If you specify the optional database handle, **qli** renames the procedure from that database.

[database-handle.]new-name
Specifies the new name of the procedure. The procedure name can be up to 31 characters and can contain alphabetic characters (A—Z and a—z, all stored as uppercase), numeric characters (0—9), underscores (_), and dollar signs ($). The procedure name must start with an alphabetic character.

Unless you supply a database handle, **qli** creates the new procedure in the most recently readied database. If a procedure of that name exists in that databases **qli** reports an error.

Example

The following command renames a procedure:

```
QLI> rename procedure capital_info to capital_city
QLI>
```

Troubleshooting

You may encounter the following messages when you use the **rename procedure** statement:

- Procedure name <name> is in use

  Choose another name.
- Procedure name over 31 characters

  Choose a shorter name.
- *gds_$create_blob failed*

  InterBase could not create the field in which the procedure text is stored. Try again.

You may get the following errors when you execute a procedure:

- *Procedure <name> is undefined*

  The procedure does not exist as specified. Type **show procedures** for a list of procedures.

- *Procedure <name> not found*

  The procedure does not exist as specified. Type **show procedures** for a list of procedures.

See Also

See the entries in this chapter for:

- **copy procedure**
- **define procedure**
- **edit procedure**
- **delete procedure**

Repeat

Function

The **repeat** statement lets you execute a **qli** statement multiple times.

Syntax

```
repeat integer-expression qli-statement
```

Options

integer-expression
Specifies the number of repetitions. If *integer-expression* is not an integer, **qli** truncates the fractional part. The token *integer-expression* does not have to be a literal; instead, it can be a prompting expression, an arithmetic expression, or even a field name.

qli-statement
Any of the **qli** statements or any procedure containing statements. You cannot use a command or a procedure containing commands as the action of an **if-else** statement.

You can intermix the GDML and SQL variants of **qli** in a **repeat** statement.

Usage

If you want to include a procedure in a **repeat** statement, enclose it in a **begin-end** statement. Otherwise, only the first statement in the procedure repeats.

If, at any time during the repeated operations, you decide to stop, type the end-of-file character (system-dependent). **Qli** then stops whatever it is doing and displays the following message:

```
Error: execution terminated by signal
```

Qli does not complete the operation that was interrupted. For example, suppose you decide to store five new SKI_AREAS. After storing two records, you begin the third. However, you make a mistake while typing the value of the second field for the third record. You type the end-of-file character. **Qli** stores the first two records, but does not store the third record.

Examples

The following example specifies that the **store** statement is to be repeated five times, thereby causing **qli** to prompt for field values for five records:

```
QLI> repeat 5 store ski_areas
      ⇓
```

The following statement prompts for the number of repetitions:

```
QLI> repeat*.'number of items'
CON> store ski_areas
```

The following statement repeats a procedure five times:

```
QLI> repeat 5 begin :procedure end
      ⇓
```

Troubleshooting

See the discussion of errors in of the *Qli Guide*.

See Also

Any of the qli statements discussed in this chapter.

Report

Function

The **report** statement invokes **qli**'s report writer.

Syntax

```
report rse [on 'filespec'|to shell-command]
    [set report_name = query-header]
    [set columns = n]
    [set lines = n]
    [at top of report [print] print-list]
    [at bottom of report [print] print-list]
    [at top of page [print] print-list]
    [at bottom of page [print] print-list]
    [at top of database-field [print] print-list]
    [at bottom of database-field [print] print-
list]
end_report [on 'filespec'|to shell-command]
 print-list::=print-element-commalist

print-element::={format-token|qualified-value}

format-token::={space [integer]|skip
[integer]|tab [integer]|col integer|
new_page|column_header|report_header}

qualified-value::=value-expression [(query-
header|-)][using edit-string]

query-header::=quoted-string[/quoted-string]
```

Options

rse
Creates the record stream to be reported.

on *'filespec'*
Sends the output to the named file, rather than writing it to your monitor. This clause can appear immediately after the **report** statement or after the **end_report** statement.

to *shell-command*
Sends the output to standard input of the shell or command interpreter command, rather than writing it to your screen. These commands typically send the output to a printer, for

example **print**, **lpr**, **lpt**, **prf**, and **'prf -npag'**. Note that if you include a switch on the shell command, you must quote the entire command. This clause can appear immediately after the **report** statement or after the **end_report** statement.

set report_name = *query-header*
Names the report. The query header is one or more quoted strings separated by slashes.

set columns = *n*
Specifies the width in mono-spaced characters for the output device. Reports printed on standard U.S. (8-1/2 by 11 inch) or European (A4) paper should not exceed 75 columns.

set lines = *n*
Specifies the length in lines of the report. Reports printed on standard U.S. (8-1/2 by 11 inch) or European (A4) paper should not exceed 60 lines in length.

at top of report print *print-list*
Specifies a title to be printed at the beginning of the report. If you omit this statement, **qli** prints the column headers at the top of the report along with the report name if you specified one.

at top of page print *print-list*
Specifies a title to be printed at the top of every page. If you omit this statement, **qli** prints column headers at the top of each page. If you use this statement, and you want default headers or user-specified headers to print, include the **column_header** format token.

at bottom of page print *print-list*
Specifies a title to be printed at the bottom of every page.

at bottom of report print *print-list*
Specifies a title to be printed at the end of the report.

at top of *database-field* **print** *print-list*
Provides a control break and the title to print for that break. The *rse* at the beginning of the report must include the *database-field*, and the *database-field* must be a sort field.

at bottom of *database-field* **print** *print-list*
Provides a summary of a control group and an expression to print for that break. Typically, the **at bottom** matches an **at**

top and calculates a total or aggregate expression for the control group.

`print-statement`
Provides a detail line which prints once for every record.

space [*integer*]
Inserts one blank horizontal space in the output line. If you specify the optional integer, **qli** inserts that many blank spaces.

skip [*integer*]
Inserts one blank vertical line. If you specify the optional integer, **qli** inserts that many blank lines.

tab [*integer*]
Inserts one horizontal tab into the output line. If you specify the optional integer, **qli** inserts that many tabs.

col *integer*
Begins the next print element in the specified column.

new_page
Inserts a page break into the output.

column_header
Specifies that the lines of the query header, either default or user-specified, should appear at the top of each column. By default, column headers appear at the top of each page. If you format the top of page yourself using the **at top of page** statement, and you want your headers to appear, you should include the **column_header** format token.

report_header
Specifies the report name you specified. If you want to print the report name at the top of each page, include the **report_header** format token in the **at top of page** statement. If you format the first page of the report yourself, you should also specify the report header.

value-expression
Specifies the field, variable, quoted string, arithmetic expression, statistical expression, or other value to be printed. If the value expression is a database field name or includes one, you must supply a record selection expression in the **print** statement (do not use a **for...print** format), and **qli** must be able to

resolve the field in the context of the *rse*. For more information about value expressions, see the entry in this chapter for *value-expression*.

query-header
Specifies a column header. By default, **qli** uses the database field name as a column header. If you include a value expression that is not a field, or if you want to use a non-standard column header, you may specify a query header in the print list. To create a multi-line query header, separate the lines of the header with slashes. For example, "Name" / "of" / "Field" stacks the words "Name," "of," and "Field" on three lines at the top of the column. To suppress the creation of a default column header, specify an unquoted hyphen.

edit-string
Specifies an output format for the value expression. **Qli** forces the value expression into the specified format if possible. For example, *print "today"* prints the string "today," but *print "today" using w(9)* prints the last day of the week.

See the tables in the *edit-string* section of the *field-attributes* entry. These tables present alphabetic, numeric, and date edit strings. In all cases where the token *integer* appears, you can substitute a repetition of the edit string character. For example, *hhhh* and *h(4)* are equivalent.

Examples

The following statements report on records in the CITIES relation with control breaks by state:

```
QLI> report cities with population not missing
CON> sorted by state
CON> set columns = 75
CON> set lines = 55
CON> set report_name =
CON> 'C I T I E S  B Y S T A T E'
CON> at top of state print state
CON> print city, population, altitude, latitude,
CON> longitude
CON> end_report
```

The following report joins records from the CITIES and STATES relation to display data from cities with populations exceeding 1,000,000:

```
QLI> report c1 in cities cross s in states over
CON> state cross
CON> c2 in cities over state with
CON> c2.population > 1000000 sorted by s.state
CON> set report_name = "A Report of Cities" /
CON> "From States With Very Large Cities" /
CON> "In the United States"
CON> at top of report print col 54, "today" using
CON> dd-mmm-yyyy, skip, report_header
CON> at top of page print col 57, "Page", col 62,
CON> running count (-) using Z(9),
CON> skip, column_header
CON> at top of state print s.state_name
CON> print c1.city, c1.population
CON> at bottom of state print total c1.population,
CON> new_page
CON> at bottom of report print total c1.population
CON> end_report on "cities.out"
```

Troubleshooting

See the discussion of errors and error handling in Chapter 1 of the *Qli Guide.*

See Also

See the chapter on the report writer in the *Qli Guide.*

See also the entry in this chapter for **print**.

Restructure

Function

The **restructure** statement lets you copy data from one relation to another or from one database to another. **Qli** automatically matches up fields and copies values from one relation to another.

Syntax

```
[database-handle.]relation name = rse
```

Options

database_handle.relation_name
Specifies the relation to which you want to assign values. The optional database handle is useful if you are using multiple databases with overlapping relation names.

rse
Creates a record stream that serves as the source of values for *relation-name*.

Examples

The following example assumes that you have defined a new relation, CITY_STATES, into which you want to store cities with populations greater than 500,000. The relation definition follows, with fields from both the CITIES and STATES relations:

```
QLI> define relation city_states
CON> city, state, population
CON> define relation city.states
CON> city, population, state_name
```

The following statement loads the new relation:

```
QLI> city_states = cities cross states over state
CON> with population > 500000
```

Troubleshooting

See the discussion of errors and error handling in Chapter 1 of the *Qli Guide*.

See Also

See the entry in this chapter for assignment.

See also the discussion of restructuring in the chapter on defining metadata in the *Qli Guide*.

Revoke

Function

The SQL **revoke** command takes privileges away from a user for a designated table or view. Only the user who grants a privilege can revoke that privilege. A revoke statement does not effect privileges a user may have received from other grant statements. The revoke statement has a cascading effect on any privileges that were passed on through the with grant option clause in the grant statement.

Syntax

```
revoke privilege-comma-list on relation-
name|view-name from userid-comma-list
```

Options

privilege-comma-list
Specifies the following operations:

Privilege	Authority
All	Selects, deletes, inserts, updates
Select	Retrieves records from table or view
Delete	Eliminates records from table or view
Insert	Stores new records in table or view
Update	Changes value of one or more fields in existing table or view

relation-name
Specifies the relation from which you revoke privileges

view-name
Specifies the view from which you revoke privileges.

userid
Specifies which authorized users have access to privileges for a table or view.

Examples

The following example takes the select privilege away from a user for the CITIES relation:

```
QLI> revoke select on cities from julie;
```

In the following example, John grants Julie select and delete privileges on a relation that he created, and he gives her the ability to pass the grant privilege to other users:

```
QLI> grant select, delete on rivers to julie with
CON> grant option;
```

Julie can now pass the select privilege for the RIVERS relation on to Dana:

```
QLI> grant select on rivers to dana;
```

If John decides to revoke Julie's select privilege for the RIVERS relation, the revoke cascades through Julie's grant statement and also takes away Dana's select privilege:

```
QLI> revoke select on rivers from julie;
```

Troubleshooting

You may encounter the following errors when you use the **grant** command:

- *expected relation name, encountered "string"*

  You typed the relation name incorrectly.

- *expected on, encountered "string"*

  You typed a privilege incorrectly.

- *QLI error from database "filename"*
  unsuccessful metadata update
  -STORE RED$USER_PRIVILEGES failed on grant
  -action cancelled by trigger (1) to preserve data integrity

  You do not have privilege to grant the privilege or privileges you tried to grant.

See Also

See the entry in this chapter for grant.

Rollback

Function

The **rollback** command ends a transaction and undoes all changes made to the database since the most recent transaction started.

The **rollback** command does not affect the **define**, **delete**, **drop**, and **modify** metadata commands.

Syntax

```
rollback [database-handle-commalist]
```

Options

database-handle-commalist

Specifies the database to roll back. A **rollback** command without the optional database handle affects all open databases. It causes InterBase to undo all changes to data.
Rollback also flushes out all modified buffers and closes any record streams that are open.

If you assign a database handle when you ready the database, you can use the handle to limit the effect of the **rollback** to specific databases. When you access more than one database in **qli**, InterBase automatically starts up separate subtransactions for each database. However, these appear to be a single transaction. The optional database handle lets you control these subtransactions explicitly by letting you commit or roll back transactions by database.

If you forgot to assign a database handle when you readied the database but later have need for one, you can use the default handle. **Qli** assigns a default handle if you have not specified one. To find out the default database handle assigned by **qli** type the following :

```
QLI> show databases
  Database "atlas.gdb" readied as QLI_0
  Page size is 1024 bytes. Current allocation
  is 156 pages.
```

Qli displays the names of all available entities, including databases and handles. The default handles are of the form "QLI_n," where *n* is a numeric integer.

Supply this handle as an argument to the **rollback** command:

```
QLI> rollback qli_0
```

Example

The following example performs some unspecified data manipulation activities and then undoes the changes. Consequently the changes are not written to the database:

```
QLI> ready atlas.gdb
⇓
QLI> rollback
```

Troubleshooting

A **rollback** cannot fail.

See Also

See the entries in this chapter for:

- **commit**
- **finish**
- **prepare**
- **quit**

Select

Function

The **select** statement finds the record(s) of the relations specified in the **from** clause that satisfy the given search condition.

Syntax

```
select-expression [ordering-clause]

ordering-clause ::= order by sort-key-commalist

sort-key ::= [asc|desc] [exactcase|anycase]
```

Options

select-expression
Specifies the selection criteria. See the manual page for *select-expression*.

ordering-clause
Returns the record stream sorted by the values of one or more *database-fields*.

You can sort a record stream alphabetically, numerically, by date, and by any combination of these. The *ordering-clause* lets you have up to 40 sort keys.

Each sort key can specify whether the sorting order of the sort key is **asc** (the default order for the first sort key) or **desc**. Unlike the GDML version of **qli**, the sorting order is not "sticky."

sort-key
Specifies the field or fields on which you want to sort. You can sort a record stream alphabetically, numerically, by date, and by any combination of these. The *sort-clause* lets you have as many sort keys as you want.

ascending | descending
Each sort key can specify whether the sorting order is **ascending** (the default order for the first sort key) or **descending**.

The sorting order is "sticky"; that is, if you do not specify whether a particular sort key is **ascending** or **descending**, InterBase 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, InterBase sorts all keys in descending order.

exactcase | anycase
The sort key can specify whether a sort is case sensitive or not. A case sensitive sort (**exactcase**) sorts capital letters before lowercase letters. A case insensitive sort (**anycase**) does not. The default is **exactcase**.

The sorting order is "sticky"; that is, if you do not specify whether a particular sort key is **exactcase** or **anycase**, InterBase assumes that you want the order specified for the most recent key. Therefore, if you list several sort keys, but only include the keyword **exactcase** for the first key, InterBase sorts all keys by exactcase.

Examples

The following query returns cities in Massachusetts:

```
QLI> select city, state, population
CON> from cities where state = 'MA'
```

The following query includes an ordering clause with two sort keys:

```
QLI> select city, state, population -
CON> from cities where state = 'MA' order by -
CON> city, state
```

The following example joins the relations CITIES and STATES on the basis of the equality of values in STATE:

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

The following query selects cities that might be on rivers:

```
QLI> select city, state from cities where
CON> state in (select state from river_states)
```

This query selects cities that might be on a longer than average river:

```
QLI> select city, state from cities where
CON> state in (select state from river_states where
CON> river in (select river from rivers where
CON> length > (select avg (length) from rivers)))
```

Troubleshooting

You may encounter the following message when you use the **select** statement:

No items in print list

You must provide something to print. *Select* * or *select [alias].** selects all fields.

See Also

See the entry in the previous chapter for **select** expression.

Set

Function

The **set** command lets you change various environmental features of **qli**.

Syntax

```
set [no] {blr|columns integer|
continuation "string"|echo|form|lines integer
|prompt "string"|semicolon|statistics|
matching_language "string"}
```

Options

`blr`
Displays the *binary language representation*, or BLR, of the query before displaying the results of the query.

You can use the blr option to develop programs that use the call interface. For example, you can develop queries using **qli**, take the generated requests, and modify them as needed by your application. However, **qli** first parses the query for syntactic accuracy before sending off the request. If there is an error in your query, **qli** displays the appropriate error message and does not generate any BLR. •

`column` *integer*
Sets the maximum width of a print line.

`continuation` "*string*"
Replaces the CON> continuation prompt with one of your own. This option does not change the QLI> prompt; use the set prompt command to change that prompt.

`echo`
Displays procedure commands and statements as a procedure is executed.

`form`
Tells **qli** to use the form defined for a relation whenever it prints, modifies, or stores records in that relation.

`lines` *integer*
Lets the number of lines that **qli** assumes will appear on the output device.

prompt *"string"*
Replaces the QLI> prompt with one of your own. This option does not change the CON> continuation prompt.. Use the **set continuation** command to change that prompt.

semicolon
Changes **qli**'s line continuation behavior. Without the **semicolon** option set, you must break a command in the middle of a clause or at a comma, or use a hyphen. With the **semicolon** option set, **qli** does not execute a command until it encounters a semicolon. When you turn off this option, be sure you type the semicolon at the end of the command:

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

statistics
Displays the following system statistics after:

- Number of read requests
- Number of write requests
- Number of requests for data which may be serviced in cache
- Number of requests for updates which may be serviced in cache
- Elapsed time
- CPU time
- Memory usage
- Database page size
- Database buffers used

matching_language *"string"*
Defines the default pattern language for the **matching** operator.

Examples

The following commands set the **blr** switch and execute a query:

```
QLI> set blr
QLI> print states
0000 blr_version4,
< BLR for query is printed >
< display of STATES records >
```

The following commands set the **statistics** switch and execute a query:

```
QLI> set statistics
QLI> print city, state of first 2 cities
< display of first 2 CITIES records >
Statistics for database "atlas.gdb"
   reads = 2 writes = 0 fetches = 7 marks = 0
      elapsed = 0.06 cpu = 0.05 mem = 55296
```

The following example changes the behavior of matching:

```
QLI> print city of cities with city matching "A?"
QLI> set matching_language "-s(A=[A-z]0 = [0-9] -
CON> a = [a-z]?=?*"
QLI> print city of cities with city matching "A"
```

Troubleshooting

You may encounter the following message when you use the **set** command:

Expected set option, encountered "invalid-option"

Qli did not recognize the **set** option you chose. Check the Syntax section above for the supported option.

See Also

See the entry in this chapter for **show**.

See the section on matching in the entry for Boolean expression in Chapter 2.

Shell

Function

The **shell** command lets you execute shell commands from the **qli** environment. This command is supported only for UNIX and Apollo environments. Use the spawn command on VMS systems.

Syntax

```
shell ["shell-command"]
```

Options

shell-command
A shell command enclosed in single (') or double (") quotation marks.

If you do not issue a shell command, **qli** puts you in a shell. Type the end-of-file character to escape from the shell back to **qli**.

Examples

The following command escapes from **qli** and deposits you in a shell:

```
QLI> shell 'sh'
%
```

Type the end-of-file character to return to **qli**.

The following command checks the time from within **qli**:

```
QLI> shell 'date'
Tue Apr 23 13:54:22 EDT 1986
```

Troubleshooting

You may encounter the following message when you use the **shell** command:

?(sh) "string" - name not found (OS/naming server)

The string you typed was not a command understood by the shell.

See also the discussion of errors in Introduction to the *Qli Guide*.

See Also

See the entry in this chapter for **spawn**.

Show

Function

The **show** command displays information about data definitions, procedures, statistics and other information from the database.

On all **show** commands except system relations, **qli** displays only user metadata; that is, metadata that has a value other than 1 or 2 for the RDB$SYSTEM_FLAG in the appropriate system relation. See the *Data Definition Guide* for more information about the **system_flag** option on most data definition statements.

If the **show** command references anything other than one of the listed options, **qli** assumes that you mean a procedure and looks for a procedure with that name. If it cannot find a procedure with that name, **qli** returns a message that the procedure was not found.

Syntax

```
show {all |database-handle | databases | database
database-handle | field relation-name.field-name
| fields [for relation relation-name] |
filters ["string"] |
forms [for [database] database-handle |
function[s] | global field field-name |
global fields [for database database-handle]|
indexes [for [database-handle.]relation-name]
|matching_language|procedure procedure-name |
procedures | ready | relation-name | relations |
security_classes | system [relations] |
triggers [for database database-handle] |
triggers for [relation] relation-name| variables |
version |views}
```

Options

all
Displays the file specification and handle for all readied databases, relation name, and field names and datatype for each relation.

database-handle
Displays everything about the named database, including a database description if one exists. The handle must have been

assigned in the **ready** statement, or may have been assigned automatically by **qli**.

databases
Displays the file specification, handle, page size, allocation, and database description of all readied databases.

database database-*handle*
Displays the file specification and handle for all the database identified by the handle, relation names, and field names and datatype for each relation.

field *relation-name.field-name*
Displays the attributes of the field as it occurs in the named relation.

fields [**for relation** *relation-name*]
Displays all fields and datatypes for each relation in a readied database. If you specify a *relation-name*, it displays the fields and datatypes for the named relation.

filters [*"string"*]
Lists all blob filters. If you include a string naming the filter, it shows the named filter.

forms [**for** [**database**] *database-handle*
Displays the names of forms for each readied database. If you include the optional database handle clause, it only displays the forms for the named database.

function
Lists all information about a particular function.

functions
List the function names in all the open databases.

global field *field-name*
Lists the description of the named global field.

global fields [**for database** *database-handle*]
Lists the global fields for all readied databases, or, if you specify the optional database handle, only the global fields for that database

indexes [**for** [*database-handle.*]*relation-name*]
Displays for each relation the name of any index that has been defined, the fields that comprise the index, and whether or not

it is a unique index, or a message indicating that no relation was defined. If you specify the relation name, it lists the indexes only for that relation.

Qli also accepts the correct form of the plural for "index," "indices."

matching_language
Displays the current matching language.

procedure *procedure-name*
Displays the file specification of the database where *procedure-name* is stored and the text of the procedure.

This option is essentially the default. If you ask **qli** to show you something it does not understand, **qli** assumes that the desired item is a procedure.

procedures
Displays the names of procedures for all readied databases.

ready
Displays the file specification and handle of all readied databases.

relation-name
Displays the field names and datatypes for the specified relation. You can also qualify the relation name with a database handle.

relations
Displays the names of relations for each readied database.

security_classes
Displays security classes defined for the database and objects with which they are associated.

system [**relations**]
Displays the names of the system relations for each readied database.

triggers [**for database** *database-handle*]
Displays the triggers for all relations in all readied databases, or, if you specify the optional database handle, only for the relations in the named database.

triggers for [relation] *relation-name*
Displays the triggers for the specified relation.

variables
Displays the names of declared variables.

version
Displays the software release number for **qli**, and the version numbers of the access method being used. For network connections, **qli** lists all participating versions of InterBase.

views
Displays the names of views for each readied database.

Examples

The following commands ready a database and then ask for information about all entities in the database:

```
QLI> ready atlas.gdb
QLI> show all
< display of all metadata information for readied
database >
```

The following command asks for version information:

```
QLI> show version
   QLI, version "S3-I3.0L"
   Version(s) for database "atlas.gdb"
      InterBase/sun4 (access method, version "S4-
T3.0K"
      InterBase/sun4 (remote server), version "S4-
T3.0K/tcp (yuppie)"
      InterBase /sun (remote interface, version
"S3-I3.0/tcp (pisces)"

QLI>
```

The following command asks for information about readied databases:

```
QLI> show ready
Database "/usr/castor/databases/atlas.gdb" readied
as QLI_0
   Page size is 1024 bytes. Current allocation is
   0 pages.
      Database description:
```

```
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.
```

Troubleshooting

You may encounter the following messages when you use the **show** command:

- *No databases are currently ready*

  Qli cannot display anything because there is nothing to display. Ready a database and try the command again.

- *Procedure <procedure-name> not found*

  Qli could not find a procedure with the name you typed. Type **show procedures** for a list of procedures. Likewise, if you reference a relation from a database other than the one(s) you have readied, **qli** assumes that the relation name is a procedure name. If it cannot find a procedure with that name, **qli** returns a message that the procedure was not found.

See Also

See also the discussion of the **show** command in the section on help in the introductory chapter of the *Qli Guide*.

Spawn

Function

The **spawn** command lets you "escape" from **qli** to a VMS subprocess. When you are finished with DCL commands, log out of the subprocess to return to **qli**.

Syntax

```
spawn
```

Example

The following command escapes from **qli** and deposits you at DCL level:

```
QLI> spawn
$
```

Logout to return to **qli**.

Troubleshooting

See the discussion of errors in Chapter 1 of the *Qli Guide*.

See Also

See the entry in this chapter for **shell**.

Store

Function

The **store** statement inserts a new record into a relation.

Syntax

Standard form:

```
store relation-name [using statement]
```

Form format:

```
store relation-name using form [form-name]
```

Options

relation-name
Specifies the relation into which you want to store a new record. If you specify only *relation-name* without an assignment, **qli** prompts you for field values.

statement
A **qli** statement.

If you specify *relation-name* and **using**, you make assignments to fields using **qli** statement. In this case, you need the **begin-end** command if there is more than one field to which you must assign a value.

using form [*form-name*]
Tells **qli** to use a form for assignments instead of statements. If you do not supply a form name, **qli** looks for a form with the same name as the relation. Failing to find that, it returns an error, or if you want to include more than one statement.

Examples

The following example stores a record using **qli**'s automatic prompting:

```
QLI> store ski_areas
Enter NAME: Reedy Run
Enter TYPE: N
Enter CITY: Groton
Enter STATE: MA
```

The following example stores a record, but uses a **begin-end** statement to structure a compound statement for assigning values to each field:

```
QLI> store ski_areas
CON> begin
CON> name = 'Moose Pond'
CON> type = 'N'
CON> city = 'Dixville Notch
CON> state = 'NH'
CON> end
```

Troubleshooting

See the entry in this chapter for the **assignment** statement.

See Also

See the entry in this chapter for:

- assignment
- **begin-end**

Then

Function

The **then** statement lets you sequence **qli** statements.

Syntax

```
qli-statement then qli-statement
```

Options

qli-statement
Any of the **qli** statements or any procedure containing statements. You cannot use a command or a procedure containing commands as the action of an **if-else** statement.

Example

The following example modifies a field value in several records, but displays each record before prompting for a new field value:

```
QLI> for cross_country with state = 'VT'
CON> print area_name, food then modify food
```

Troubleshooting

You may encounter the following message when you use the **then** statement:

Expected statement, encountered "command"

You cannot use a command with the **then** statement.

See Also

See also the discussion of errors and error handling in Chapter 1 of the *Qli Guide*.

Update

Function

The **update** statement changes the values of one or more fields in a record in a relation.

You cannot modify records directly from a join: you must modify them through each participating relation separately. You can modify records from a view if the view is comprised of a single relation (without a reflexive join) or if the database designer included a *modify* trigger.

Syntax

```
update relation-name set assignment-commalist
[where predicate]

assignment ::= database-field=scalar-expression
```

Options

relation-name
Specifies the relation that contains the record you want to update.

assignment
Assigns the *scalar-expression* to *database-field*.

predicate
Selects the record to modify. If you provide a search condition with the optional *where-clause* of the *predicate*, InterBase updates the listed fields in the record(s) selected from *relation-name*. If you do not provide a search condition, InterBase updates all records in *relation-name*.

Example

The following statement modifies the altitude of all cities:

```
QLI> update cities set altitude = altitude - 10
```

The following statement modifies the altitude of all cities in California and Washington:

```
QLI> update cities -
CON> set altitude = altitude - 100 -
CON> where state in ("CA", "WA");
```

The following statement elevates all cities in New York and changes the state code to "NA" (New Amsterdam):

```
QLI> update cities -
CON> set altitude = altitude * 1.1, state = 'NA' -
CON> where state = 'NY'
```

Troubleshooting

See the entry for the **assignment** statement.

See Also

See the entries in this chapter for:

- predicate
- **select**

E

F

G

Q

R

S

T

U

V

W