Ingres® 2006 Release 2

QUEL Reference Guide

INGRES®

February 2007

This documentation and related computer software program (hereinafter referred to as the "Documentation") is for the end user's informational purposes only and is subject to change or withdrawal by Ingres Corporation ("Ingres") at any time.

This Documentation may not be copied, transferred, reproduced, disclosed or duplicated, in whole or in part, without the prior written consent of Ingres. This Documentation is proprietary information of Ingres and protected by the copyright laws of the United States and international treaties.

Notwithstanding the foregoing, licensed users may print a reasonable number of copies of this Documentation for their own internal use, provided that all Ingres copyright notices and legends are affixed to each reproduced copy. Only authorized employees, consultants, or agents of the user who are bound by the confidentiality provisions of the license for the software are permitted to have access to such copies.

This right to print copies is limited to the period during which the license for the product remains in full force and effect. The user consents to Ingres obtaining injunctive relief precluding any unauthorized use of the Documentation. Should the license terminate for any reason, it shall be the user's responsibility to return to Ingres the reproduced copies or to certify to Ingres that same have been destroyed.

To the extent permitted by applicable law, INGRES PROVIDES THIS DOCUMENTATION "AS IS" WITHOUT WARRANTY OF ANY KIND, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IN NO EVENT WILL INGRES BE LIABLE TO THE END USER OR ANY THIRD PARTY FOR ANY LOSS OR DAMAGE, DIRECT OR INDIRECT, FROM THE USE OF THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION, LOST PROFITS, BUSINESS INTERRUPTION, GOODWILL, OR LOST DATA, EVEN IF INGRES IS EXPRESSLY ADVISED OF SUCH LOSS OR DAMAGE.

The use of any product referenced in this Documentation and this Documentation is governed by the end user's applicable license agreement.

The manufacturer of this Documentation is Ingres Corporation.

For government users, the Documentation is delivered with "Restricted Rights" as set forth in 48 C.F.R. Section 12.212, 48 C.F.R. Sections 52.227-19(c)(1) and (2) or DFARS Section 252.227-7013 or applicable successor provisions.

Copyright © 2006 Ingres Corporation.

All Rights Reserved.

Ingres, OpenROAD, and EDBC are registered trademarks of Ingres Corporation. All other trademarks, trade names, service marks, and logos referenced herein belong to their respective companies.

Contents

Chapter 1: Introduction

Chapter 2: Introduction to QUEL

Chapter 3: QUEL Data Types

Chapter 4: Elements of QUEL Statements

Chapter 5: Embedded QUEL

Chapter 6: QUEL and EQUEL Statements

Appendix A: Keywords

Appendix B: Terminal Monitor

Appendix C: Calling Ingres Tools from Embedded QUEL

Index

Chapter 1: Introduction

The *QUEL Reference Guide* provides detailed descriptions of all QUEL statements and provides examples of the correct use of QUEL statements and features.

This guide identifies QUEL as the Ingres proprietary query language and introduces features of QUEL's interactive and embedded releases. It describes QUEL data types and QUEL statements. The guide also discusses:

- Statement syntax
- Program structure
- Host language variables
- Cursors, transaction processing
- Program status information
- Error handling
- Retrieve statement
- Repeat queries

Audience

The *QUEL Reference Guide* is intended for programmers and users of QUEL who have a basic understanding of how relational database systems work. In addition, you must have a basic understanding of the operating system. This guide is also intended as a reference guide for the database system administrator.

Conventions

Query Languages

The industry standard query language, SQL, is used as the standard query language throughout the body of this guide.

Ingres is compliant with ISO Entry SQL92. In addition, numerous vendor extensions are included. For details about the settings required to operate in compliance with ISO Entry SQL92, see the *SQL Reference Guide*.

System-specific Text

The available QUEL statements for UNIX platforms and VMS are described in this guide. Where information differs by system, read the information for your system.

UNIX This text is specific to the UNIX environment.

VMS This text is specific to the VMS environment.

The symbol  indicates the end of the system-specific text.

Embedded QUEL Examples

This guide contains examples of embedded QUEL code. The examples use the following conventions:

Margins	None used
Labels	Appear on a line of their own and are followed by a colon (:)
Host language comments	Indicated by the QUEL comment indicator. For example: /* This is a comment. */
Character strings	Enclosed in double quotes (" ")
Pseudocode	Represents host language statements within embedded QUEL. For example: tablevar = "mytable" descvar = "name = c20, phone = c11" ## create tablevar (descvar)

To determine the correct syntax for your programming language, see the *Embedded QUEL Companion Guide*.

Chapter 2: Introduction to QUEL

QUEL is an Ingres proprietary query language. You use QUEL statements to manipulate and query the information in your database.

There are *interactive* and *embedded* releases of QUEL.

- Interactive QUEL enables you to enter QUEL statements from a terminal and display query results on the terminal screen.
- Embedded QUEL enables you to include QUEL statements in programs written in programming languages such as C or Fortran.

This chapter introduces the features of embedded and interactive QUEL.

Interactive QUEL

There are two ways to use interactive QUEL:

- The forms-based interactive terminal monitor is invoked by the iquel command. Enter QUEL statements into a form and select commands from a menu line.
- The command-based Terminal Monitor is invoked by the quel command. For details, see the appendix "Terminal Monitor."

Embedded QUEL

Embedded QUEL (EQUEL) enables you to include QUEL statements in application programs. This guide refers to the programming language of the application as the *host* language.

For each host language, there is an EQUEL preprocessor. The preprocessor scans your source code for QUEL statements and translates the QUEL statements into the appropriate host language statements. For detailed information about language-dependent topics, see the *Embedded QUEL Companion Guide*.

In addition to the statements available to you in interactive QUEL, embedded QUEL offers the following features:

- Database cursors and transaction processing

  Database cursors enable your application to process database rows that fulfill specified search criteria. Transactions help you to preserve database integrity by grouping QUEL statements; if a transaction fails for any reason, the effects of all the statements in the transaction are undone.

- Dynamic programming

  Your application program can specify portions of many QUEL statements using host variables. The param statement enables database manipulation statements to be built dynamically, in cases where the number and data type of objects to be operated on is not determined until runtime.

- Status information

  QUEL provides inquiry statements that return detailed information about the database and forms being used by your application program.

- Runtime error handling

  In EQUEL applications, you can silence error messages and trap errors using an error handler routine. For more information on handling runtime errors, see the *Embedded QUEL Companion Guide*.

- Repeat queries

  You can reduce the overhead required to run an embedded query that is executed many times by using repeat queries. The first time a repeat query is executed, the DBMS Server encodes the query. On subsequent executions of the query, this encoding can account for significant performance improvements.

Chapter 3: QUEL Data Types

This chapter describes QUEL data types. This chapter points out differences in syntax between embedded and interactive QUEL. When the embedded syntax is dependent on the host language, you are referred to the *Embedded QUEL Companion Guide*.

Object Names

The rules for naming database objects (such as tables, columns, views, and database procedures) created using QUEL are as follows:

- Names can contain only alphanumeric characters, and must begin with an alphabetic character or an underscore (_).
- Names can contain (though not begin with) the following special characters: "0" through "9", "#," "@," and "$".
- Table names cannot begin with "ii". These names are reserved for use by the DBMS Server.
- The maximum length of an object name is 32 characters. Examples of objects managed by Ingres tools (such as VIFRED or Vision) are:
  - Forms
  - JoinDefs
  - QBFNames
  - Graphs
  - Reports
- Avoid assigning reserved words as object names.

For more information about objects managed by Ingres tools, see the *Character-based Querying and Reporting Tools User Guide* or *Forms-based Application Development Tools User Guide*.

Access to Objects Created Through SQL

From QUEL you can freely access objects created using SQL, if the object name is a valid QUEL object name. However, you cannot access objects if the object name is mixed case or contains special characters. These objects are created through SQL using *delimited identifiers*. For example, using SQL you can create a table named "my table" (note the space embedded in the name.) You cannot access this table from QUEL–you must use SQL. For details about delimited identifiers, see the *SQL Reference Guide.*

Comment Delimiters

To indicate comments in interactive QUEL, use "/*" and "*/" (left and right delimiters, respectively). F or example:

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

When you use "/*...*/" to delimit a comment, the comment can continue over more than one line. For example,

```
/*  Everything  from  here...
...to  here  is  a  comment  */
```

To indicate comments in embedded QUEL, precede the comment delimiters with "##". For example:

```
##  /*  This  is  an  EQUEL  comment  */
```

In embedded QUEL you can also use host language comment delimiters. For information about comment delimiters, see the *Embedded QUEL Companion Guide*.

Data Types

There are four classes of data types: character, numeric, abstract and binary. Character strings can be fixed length (c and char) or variable length (text and varchar). Numeric strings can be exact numeric (i4, i2, or i1) or approximate numeric (float4 and float8). The abstract data types are date and money. Binary data can be fixed length (byte) or variable length (byte varying).

Class	Category	Data Type (Synonyms)
Character	Fixed length	c
		char (character)
	Varying length	text
		varchar
Numeric	Exact numeric	integer4 (i4, integer)
		integer2 (i2, smallint)
		Integer8 (i8, bigint)
		Integer1 (i1, tinyint)
		decimal
	Approximate numeric	float (float8, double precision)
		float4 (real)
Abstract	(none)	date
	(none)	money
Binary		byte
		byte varying

Character Data Types

Character data types are strings of ASCII characters. Upper and lower case alphabetic characters are accepted literally. There are two fixed-length character data types, char and c, and two variable-length character data types: varchar and text.

The maximum row length in a table is 2008 bytes. Therefore, the maximum length of a character column is 2008 minus any additional space requirements. Additional space requirements for character columns are as follows:

- varchar columns require two additional bytes to store a length specifier
- nullable char and varchar columns require one additional byte to store a null indicator

Char Data Type

Char strings can contain any printing or non-printing character, and the null character ("\0"). In uncompressed tables, char strings are stored blank-padded to the declared length. (If the column is nullable, char columns require an additional byte of storage.) For example, if you enter "ABC" into a char(5) column, five bytes are stored, as follows:

```
"ABC  "
```

In compressed tables, trailing blanks are removed from char columns. In general, if your application must preserve trailing blanks, use varchar.

Leading and embedded blanks are significant when comparing char strings (unlike c strings). For example, the following char strings are different:

```
"A B C"
"ABC"
```

When retrieving char strings using the question mark (?) wildcard character, you must include any trailing blanks you want to match. For example, to retrieve the following char string:

```
"ABC "
```

the wildcard specification must also contain trailing blanks:

```
"???  "
```

Length is not significant when comparing char strings. For example, the following char strings are equal, even though the second string contains trailing blanks:

```
"ABC" = "ABC "
```

Character is a synonym for char.

C Data Type

The c data type accepts only printing characters. Non-printing characters, such as control characters, are converted into blanks.

The DBMS ignores blanks when comparing c strings. For example, the c string:

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

is treated identically to:

```
"thehouseisaroundthecorner"
```

The c type is supported for backward compatibility, but char is the recommended fixed-length character data type.

Varchar Data Type

Varchar strings are variable-length strings, stored as a 2-byte (I2) length specifier followed by data. In uncompressed tables, varchar columns occupy their declared length. (If the column is nullable, varchar columns require an additional byte of storage.) For example, if you enter "ABC" into a varchar(5) column, the stored result is:

```
"03ABCxx"
```

where "03" is a 2-byte length specifier, "ABC" is three bytes of data, and "xx" represents two bytes containing unknown (and irrelevant) data.

In compressed tables, varchar columns are stripped of trailing data. For example, if you enter "ABC" into a varchar(5) column in a compressed table, the stored result is:

```
"03ABC"
```

The varchar data type can contain any character, including non-printing characters and the ASCII null character ("\0").

Blanks are significant in the varchar data type. For example, the following two varchar strings are not considered equal:

```
"the store is closed"
```

and

```
"thestoreisclosed"
```

If the strings being compared are unequal in length, the shorter string is padded with trailing blanks until it equals the length of the longer string.

For example, consider the following two strings:

```
"abcd\001"
```

where "\001" represents one ASCII character (Control-A) and

```
"abcd"
```

If they are compared as varchar data types, then

```
"abcd" > "abcd\001"
```

because the blank character added to "abcd" to make the strings the same length has a higher value than **Control-A** ("\040" is greater than "\001").

Text Data Type

All ASCII characters except the null character ("\0") are allowed within text strings; null characters are converted to blanks.

Blanks are not ignored when you compare text strings. Unlike varchar, if the strings are unequal in length, blanks are not added to the shorter string. For example, assume that you are comparing the text strings

"abcd"

and

"abcd "

The string "abcd " is greater than the string "abcd" because it is longer.

Text is supported for backward compatibility, but varchar is the preferred varying length character type.

Numeric Data Types

There are two categories of numeric data types: *exact* and *approximate*. The exact data types are the integer data types. The approximate data types are the floating point data types.

Integer Data Types

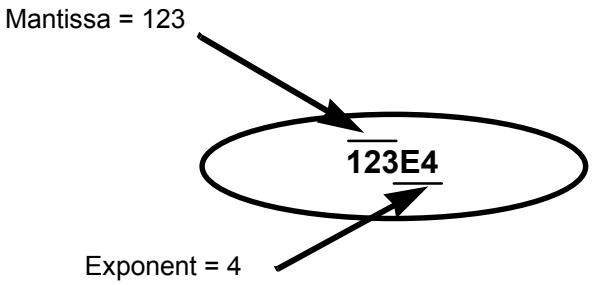
There are three integer data types: I1 (one-byte), I2 (two-byte), and I4 (four-byte). Integer2 is a synonym for I2 and integer4 is a synonym for I4.

The following table lists the ranges of values for each integer data type:

Integer Data Type	Lowest Possible Value	Highest Possible Value
I4 (integer4)	-2,147,483,648	+2,147,483,647
I2 (integer2)	-32,768	+32,767
I1	-128	127

Floating Point Data Types

A floating-point value is represented either as whole plus fractional digits (for example, 123.45) or as a mantissa plus an exponent. The following figure illustrates the mantissa and exponent parts of floating point values.



There are two floating point data types: float4 (4-byte) and float (8-byte). (Real is a synonym for float4, and float8 and double precision are synonyms for float.)

Floating point numbers are double-precision quantities stored in four or eight bytes. The range of float values is processor-dependent, and the precision is approximately 16 significant digits.

For information about the correct notation for a floating-point numeric literal, see Numeric Literals.

Date Data Type

The date data type is an abstract data type. Date values can be either absolute dates and times or time intervals.

Absolute Date Input Formats

Dates are specified as quoted character strings. You can specify a date by itself or together with a time value. If you enter a date without specifying the time, no time is displayed on output. For more information about date and time display, see Date and Time Display Formats.

The legal formats for absolute date values are determined by the setting of II_DATE_FORMAT, summarized in the following table. If II_DATE_FORMAT is not set, the US formats are the default input formats. II_DATE_FORMAT can be set on a session basis; for information on setting II_DATE_FORMAT, see the *System Administrator Guide.*

II_DATE_FORMAT Setting	**Valid Input Formats**	**Output**
US (default format)	*mm/dd/yyyy* *dd-mmm-yyyy* *mm-dd-yyyy* *yyyy.mm.dd* *yyyy_mm_dd* *mmddyy* *mm-dd* *mm/dd*	*dd-mmm-yyyy*
MULTINATIONAL	*dd/mm/yyyy* and all US formats except *mm/dd/yyyy*	*dd/mm/yy*
ISO	*yymmdd* *ymmdd* *yyyymmdd* *mmdd* *mdd* and all US input formats except *mmddyy*	*yymmdd*
SWEDEN/FINLAND	yyyy-mm-dd all US input formats except *mm-dd-yyyy*	*yyyy-mm-dd*
GERMAN	*dd.mm.yyyy* *ddmmyy* *dmmyy* *dmmyyyy* *ddmmyyyy* and all US input formats except *yyyy.mm.dd* and *mmddyy*	dd.mm.yyyy
YMD	*mm/dd* *yyyy-mm-dd* *mmdd* *yymdd* *yymmdd* *yyyymdd* *yyyymmdd* *yyyy-mmm-dd*	*yyyy-mmm-dd*

II_DATE_FORMAT Setting	Valid Input Formats	Output
DMY	*dd/mm* *dd-mm-yyyy* *ddmm* *ddmyy* *ddmmyy* *ddmyyyy* *ddmmyyyy* *dd-mmm-yyyy*	*dd-mmm-yyyy*
MDY	*mm/dd* *mm-dd-yyyy* *mmdd* *mddyy* *mmddyy* *mddyyyy* *mmddyyyy* *mmm-dd-yyyy*	*mmm-dd-yyyy*

Year defaults to the current year. In formats that include delimiters (such as forward slashes or dashes), you can specify the last two digits of the year. The first two digits default to the current century (2000). For example, if you enter

```
"03/21/03"
```

using the format *mm/dd/yyyy*, the DBMS assumes that you are referring to March 21, 2003.

In three-character month formats, for example, *dd-mmm-yy*, you must specify three-letter abbreviations (for example, mar, apr, may).

To specify the current system date, use date(today). For example:

```
retrieve (tdate=date("today"))
```

(BODY_TEXT__C)To specify the current system time, use date(now).

Absolute Time Input Formats

The legal format for inputting an absolute time is:

hh:*mm*[:*ss*] [**am**|**pm**] [**gmt**]

Input formats for absolute times are assumed to be on a 24-hour clock. If you enter a time with an am or pm designation, the DBMS Server automatically converts the time to a 24-hour internal representation.

If you omit gmt (Greenwich Mean Time), the local time zone designation is assumed. Times are stored and displayed using the time zone adjustment specified by II_TIMEZONE_NAME. If you enter an absolute time without a date, the current system date is assumed.

Combined Date and Time Input

Any valid absolute date input format can be paired with a valid absolute time input format to form a valid date and time entry. Some examples are shown in following table, using the US absolute date input formats:

Format	Example
"mm/dd/yy hh:mm:ss"	"11/15/03 10:30:00"
"dd-mmm-yy hh:mm:ss"	"15-nov-03 10:30:00"
"mm/dd/yy hh:mm:ss"	"11/15/03 10:30:00"
"dd-mmm-yy hh:mm:ss **gmt**"	"15-nov-03 10:30:00 gmt"
"dd-mmm-yy hh:mm:ss [**am**|**pm**]"	"15-nov-03 10:30:00 am"
"mm/dd/yy hh:mm"	"11/15/03 10:30"
"dd-mmm-yy hh:mm"	"15-nov-03 10:30"
"mm/dd/yy hh:mm"	"11/15/03 10:30"
"dd-mmm-yy hh:mm"	"15-nov-03 10:30"

Date Interval Formats

Date intervals, like absolute date values, are entered as quoted character strings. You can specify date intervals in terms of years, months, days, or combinations of these. You can abbreviate years and months to yrs and mos, respectively. For example:

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

The following table lists valid ranges for date intervals:

Date Interval	Range
Years	-9999 to +9999
Months	-119977 to +119977
Days	-3652047 to +3652047

Time Interval Formats

You can express time intervals as hours, minutes, seconds, or combinations of these units. (You can abbreviate time intervals to hrs, mins, or secs.) For example:

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

All values in an interval must be in the range -2,147,483,639 to +2,147,483,639. The DBMS Server adjusts time units as appropriate, as illustrated in the following table:

Value entered	Value displayed
3601 seconds	1 hrs 1 secs
61 minutes	1 hrs 1 mins
26 hours	1 day 2 hours

Date and Time Display Formats

Date values are displayed as strings of 25 characters with trailing blanks inserted. To specify the output format of an absolute date and time, you must set II_DATE_FORMAT. For a list of II_DATE_FORMAT settings and associated formats, see Absolute Date Input Formats. The display format for absolute time is:

hh:mm:ss

The DBMS Server displays 24-hour times for the current time zone, which is determined when Ingres is installed. Dates are stored in Greenwich Mean Time and adjusted for your time zone when they are displayed.

If you do not enter seconds when you enter a time, zeros are displayed in the seconds' place when that value is retrieved and displayed.

For a time interval, the DBMS Server displays the most significant portions of the interval that fit in the 25-character string. If necessary, trailing blanks are appended to fill out the string. The format appears as

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

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

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

the entry is displayed as:

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

truncating the minutes and seconds, the least significant portion of the time, to fit the result into 25 characters.

Money Data Type

The money data type is an abstract data type. Money values are stored significant to two decimal places. Money values are rounded to dollars and cents on input and output, and arithmetic operations on the money data type retain two-decimal-place precision.

The range of money values is:

$-999,999,999,999.99 to $999,999,999,999.99

You can specify a money value as either:A character string literal

The format for character string input of a money value is "$*sddddddddddddd.dd*". The dollar sign is optional and the algebraic sign (*s*) defaults to + if not specified. You do not need to specify a cents value of zero (.00).A number

Any valid integer or floating point number is acceptable. The DBMS Server converts the number to the money data type automatically.

On output, money values are displayed as strings of 20 characters with a default precision of two decimal places. The display format is:

$[-]*ddddddddddddd.dd*

where

$ is the default currency symbol
d is a digit from 0 to 9

The following settings affect the display of money data. For more details, see the *System Administrator Guide.*

Variable	Description
II_MONEY_FORMAT	Specifies the character displayed as the currency symbol. The default currency sign is the dollar sign ($).
II_MONEY_PREC	Specifies the number of digits displayed after the decimal point. Valid settings are 0, 1, and 2.
II_DECIMAL	Specifies the character displayed as the decimal point. The default decimal point character is a period (.).

Binary Data Types

There are three binary data types:

- Byte
- Byte varying

Binary columns can contain data such as graphic images, which cannot easily be stored using character or numeric data types. The binary data types are described in the following sections.

Byte Data Type

The byte data type is a fixed length binary data type. If the length of the data assigned to a byte column is less than the declared length of the column, the value is padded with zeros to the declared length when it is stored in a table. The minimum length of a byte column is 1 byte, and the maximum length is limited by the maximum row width configured but not exceeding 32,000.

Byte Varying Data Type

The byte varying data type is a variable length data type. The actual length of the binary data is stored with the binary data, and, unlike the byte data type, the data is not padded to its declared length. The minimum length of a byte varying column is 1 byte, and the maximum length is limited by the maximum row width configured, but not exceeding 32,000.

Storage Formats of Data Types

The following table lists storage formats for QUEL data types:

Notation	Type	Range
char(1) - char(*n*)	character	A string of 1 to *n* characters; *n* represents the lesser of the maximum configured row size and 32,000.
c1 - c*n*	character	A string of 1 to *n* characters; *n* represents the lesser of the maximum configured row size and 32,000.
varchar(1) - varchar(*n*)	character	A string of 1 to *n* characters; *n* represents the lesser of the maximum configured row size and 32,000.
text(1) - text(*n*)	character	A string of 1 to *n* characters; *n* represents the lesser of the maximum configured row size and 32,000.
i1	1-byte integer	-128 to +127
i2	2-byte integer	-32,768 to +32,767
i4	4-byte integer	-2,147,483,648 to +2,147,483,647
float4	4-byte floating	-1.0e+38 to +1.0e+38 (7 digit precision)
float	8-byte floating	-1.0e+38 to +1.0e+38 (16 digit precision)
date	date (12 bytes)	1-jan-0001 to 30-dec-9999 (for absolute dates) and -9999 years to 9999 years (for time intervals)
money	money (8 bytes)	$-999,999,999,999.99 to $999,999,999,999.99
byte	binary	Fixed length binary data, 1 to maximum configured row size.
byte varying	binary	Variable lengthe binary data, 1 to maximum configured row size.

Note: If your hardware supports the IEEE standard for floating point numbers, the float type is accurate to 14 decimal precision ($-*ddddddddddddd.dd* to $+*ddddddddddddd.dd*) and ranges from -10**308 to +10**308, and the money type is accurate to 14 decimal precision.

Literals

A literal is an explicit representation of a value. There are two types of literals: string and numeric.

String Literals

String literals are specified by one or more characters enclosed in double quotes. The default data type for string literals is varchar, but you can assign a string literal to any character data type or to money or date data type without using a data type conversion function.

To include a double quote inside a string literal, you must precede it with a backslash; for example:

```
"The following letter is quoted: \"A\"."
```

which evaluates to

```
The following letter is quoted: "A".
```

Numeric Literals

Numeric literals specify numeric values. There are two types of numeric literals: integer and floating point.

You can assign a numeric literal to any of the numeric data types or the money data type without using an explicit conversion function. The DBMS Server automatically converts the literal to the appropriate data type, if necessary.

By default, the period (.) indicates the decimal point. You can change this default by setting II_DECIMAL. For information about setting II_DECIMAL, see the *System Administrator Guide.*

Integer Literals

Integer literals are specified by a sequence of up to 10 digits and an optional sign, in the following format:

[**+**|**-**] *digit* {*digit*} [e *digit*]

Integer literals are represented internally as either an i4 or a i2, depending on the value of the literal. If the literal is within the range -32,768 to +32,767, it is represented as a i2. If its value is within the range -2,147,483,648 to +2,147,483,647 but outside the range of a i2, it is represented as an i4.

You can specify integers using a simplified scientific notation, similar to the way floating point values are specified. To specify an exponent, follow the integer value with the letter "e" and the value of the exponent. This notation is useful for specifying large values; for example, to specify 100,000 you can use exponential notation as follows:

```
10e5
```

Floating Point Literals

A floating point literal must be specified using scientific notation. The format is:

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

For example:

```
2.3 e-02
```

You must specify at least one digit, either before or after the decimal point.

QUEL Constants

The following constants can be used in queries:

Special Constant	Description	Used in
now	The current date and time. You must specify this constant in quotes	The date() function
null	Indicates a missing or unknown value in a table.	Queries and expressions
today	The current date. You must specify this constant in quotes.	The date() function
user	The session's effective user.	Queries and expressions

These constants can be used in queries and expressions. For example:

```
/* Display the current date and time */
retrieve (dcolumn=date("now"))
/* Add a row to a sales order table, recording the
current user as the sales clerk, and a billing date
calulated as one week from today */

append to sales_order
(item_number="123", clerk=user,
 billing_date=date("today")+date("7 days"));
```

To specify the effective user, use the Ingres -u flag (for operating system commands).

Nulls

A null represents an undefined or unknown value and is specified by the keyword null. A null is not the same as a zero, a blank, or an empty string. You can assign a null to any nullable column when no other value is specifically assigned. For more information about defining nullable columns, see Create in the "QUEL and EQUEL Statements" chapter.

The ifnull function and the is null predicate enable you to handle nulls in queries. For details, see Ifnull and Is Null Comparison in the "Elements of QUEL Statements" chapter.

Nulls and Comparisons

Because a null is not a value, it cannot be compared to any other value (including another null value). For example, the following where clause evaluates to "false" if one or both of the columns is null:

```
where  columna  =  columnb
```

Similarly, the where clause

```
where  columna  <  10  or  columna  >=  10
```

is true for all numeric values of "columna", but false if "columna" is null.

Nulls and Aggregate Functions

If you execute an aggregate function against a column that contains nulls, the function ignores the nulls. This prevents unknown or inapplicable values from affecting the result of the aggregate. For example, if you apply the aggregate function avg() to a column that holds the ages of your employees, you want to be sure that any ages that have not been entered in the table are not treated as zeros by the function. This distorts the true average age. If a null is assigned to any missing ages, the aggregate returns a correct result: the average of all known employee ages.

Aggregate functions, with the exception of count(), return null for an aggregate over an empty set, even when the aggregate includes columns which are not nullable (count() returns 0).

In the following example, the retrieve returns null, because there are no rows in "test."

```
create  table  test  (col1=integer  not  null)
retrieve  (x=max(test.col1))
```

In the above example, you can use the ifnull function to return a zero (0) instead of a null:

```
retrieve (ifnull(max(test.coll),0))
```

For more information, see Ifnull in the "Elements of QUEL Statements" chapter.

Nulls and Integrity Constraints

When you create a table with nullable columns and subsequently create integrities on those columns, the constraint must include the or...is null clause to ensure that nulls are allowed in that column.

For example, if the following define statement is issued:

```
define test (a=int, b=int not null)
/* "a" is nullable */
```

and the following integrity constraint is defined on the "test" table:

```
define integrity on test is a > 10
```

the comparison "a >10" is not true whenever "a" is null. For this reason, the table does not allow nulls in column "a", even though the column is defined as a nullable column. Similarly, the following append statements fails:

```
append to test (b=5)
```

```
append to test (a=null, b=5)
```

Both of these append statements are acceptable if the integrity has not been defined on column "a". To allow nulls in column "a", you must define the integrity as

```
define integrity on test is a > 10 or a is null
```

Note: If you try to create an integrity on a nullable column without specifying the or...is null clause and the column already contains nulls, the attempt fails.

Chapter 4: Elements of QUEL Statements

This chapter describes the following elements of QUEL statements:

- Functions, operators, and predicates
- Arithmetic operations, assignments, and other basic operations
- Expressions and search conditions in queries

This chapter points out differences in syntax between embedded and interactive QUEL. If the embedded syntax is dependent on the host language, see the *Embedded QUEL Companion Guide*.

Operators

There are three types of operators in QUEL: *arithmetic*, *comparison*, and *logical*, described in the following sections.

Arithmetic

Arithmetic operators are used to combine numeric expressions arithmetically to form other numeric expressions. Valid arithmetic operators are (in descending order of precedence):

Operator	Description
+ and -	plus and minus (unary)
**	exponentiation (binary)
* and /	multiplication and division (binary)
+ and -	addition and subtraction (binary)

Unary operators group from right to left and binary operators group from left to right. You can use the unary minus (-) to reverse the algebraic sign of a value.

To force a desired order of evaluation, use parentheses; for example:

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

is an expression in which the parentheses force the addition operator (+) to take precedence over the multiplication operator (*).

Comparison

Comparison operators allow you to compare two expressions. Valid comparison operators are listed in the following table:

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

All comparison operators are of equal precedence.

The equal sign (=) also serves as the assignment operator in assignment operations. For a discussion of assignment operations, see Assignment.

Logical

QUEL has three logical operators: and, or, and not. Not has the highest precedence, followed by and, and or has the least precedence. You can use parentheses to change this behavior. For example, the following expression:

exprA or *exprB* and *exprC*

is evaluated as:

exprA or (*exprB* and *exprC*)

To change the order of evaluation you must use parentheses:

(*exprA* or *exprB*) and *exprC*

When parenthesized as shown, the DBMS evaluates (*exprA* or *exprB*) first, then ands the result with *exprC*.

You can also use parentheses to change the default evaluation order of a series of expressions combined with the same logical operator. For example, the following expression:

exprA and *exprB* and e*xprC*

is evaluated as:

(*exprA* and *exprB*) and *exprC*

To change this default left-to-right grouping, use parentheses as follows:

exprA and (*exprB* and *exprC*)

The parentheses direct the DBMS Server to and *exprB* and *exprC* and then ands that result with *exprA*.

Note: There is a per-query limit of 127 or expressions. Because the limit is checked after the query is optimized, it is not obvious that your query has exceeded the limit. The query optimizer converts all expressions to expressions combined using the and logical operator. The following example illustrates this effect of query optimization:

Before optimization:
expressionA or (*expressionB* and *expressionC*)

After optimization:
(*expressionA* or *expressionB*) and (*expressionA* or *expressionC*)

As a result of optimization, the number of ors in the query has doubled. To avoid exceeding the limit, be aware of this side-effect of query optimization.

Operations

This section describes the basic operations that you can perform: string concatenation, assignments, arithmetic operations, and date operations.

String Concatenation

To concatenate strings, use the **+** operator: for example:

```
"This " + "is " + "a " + "test."
```

gives the value

```
"This is a test."
```

You can also concatenate strings using the concat function; see String.

Assignment

An assignment operation is an operation which places a value in a column or variable. Assignment operations occur during the execution of append, replace, and retrieve statements.

When an assignment operation occurs, the data types of the assigned value and the receiving column or variable must either be the same or comparable. If they are not the same, the DBMS Server performs a default type conversion if the data types are comparable. If they are not comparable, you must convert the assignment value into a type which is the same or comparable with the receiving column or variable. For information about the type conversion functions, see Data Type Conversion Functions.

All character string types (char, varchar, c, and text) are comparable with one another. Dates are comparable with string types if the format of the value in the string corresponds to a valid date input format. For information about valid date input formats, see Absolute Date Input Formats.

All numeric types are comparable with one another. Money is comparable with all of the numeric and string types. For example, assuming that the following table is created:

```
create emp
(name=char(20),
 salary=money not null,
 hiredate=date not null);
```

this append statement

```
append to emp (name="John Smith", salary=40000,
hiredate="10/12/93")
```

assigns the varchar string literal, "John Smith", to the char column "name", the i4 literal 40000 to the money column "salary", and the varchar string literal "10/12/93" to the date column "hiredate".

The following assignment replaces an existing value in a table:

```
replace emp (name = "Mary Smith")
where name = "Mary Jones"
```

In the following embedded QUEL example, the value in the "name" column is assigned to the variable "name_var" for each row that fulfills the where clause.

```
retrieve (:name_var=emp.name)
where empno = 125
```

The following sections present guidelines for assigning values (including nulls) to each of the general data types. If you are assigning to a host language variable, see the *Embedded QUEL Companion Guide* for information about which host language data types are comparable with QUEL data types.

Character String

All character types are comparable with one another; you can assign any character string to any column or variable of character data type. The result of the assignment depends on the types of the assignment string and the receiving column or variable.

Assigned String	Receiving Column or Variable	Description
Fixed-length (c or char)	Fixed-length	The assigned string is truncated or padded with spaces if the receiving column or variable is not the same length as the fixed length string.
Fixed-length	Variable-length (varchar or text)	Trailing blanks are trimmed. If the receiving column or variable is shorter than the fixed length string, the fixed length string is truncated from the right side.
Variable-length (varchar or text)	Fixed-length	The variable length string is truncated or padded, as necessary, if the receiving column or variable is not the same length as the variable length string.
Variable-length	Variable-length	The variable length string is truncated if the receiving column or variable is not long enough.

Numeric

You can assign any numeric data type to any other numeric data type. You can assign a money value to any numeric data type and a numeric value to the money data type. Numeric assignments have the following characteristics:

- The DBMS Server can truncate leading zeros or all or part of the fractional part of a number if necessary. If the non-fractional part of a value (other than leading zeros) is truncated, an overflow error results. (These errors are reported only if the -numeric_overflow flag is set. For information about the -numeric_overflow flag, see the quel command description in the *Command Reference Guide*.)
- When a float, float4, or money value is assigned to an integer column or variable, the fractional part is truncated.

Date

You can assign absolute date or interval column values to a date column. In addition, you can assign a string literal, a character string host variable, or a character string column value to a date column if its value conforms to the valid input formats for dates.

When you assign a date value to a character string, the DBMS Server converts the date to the display format. For more information about date display formats, see Date and Time Display Formats in the "QUEL Data Types" chapter.

Null

You can assign a null to a column of any data type if the column was defined as a nullable column. You can also assign a null to a host language variable if there is an indicator variable associated with the host variable. (For more information about indicator variables, see Indicator Variables in the "Embedded QUEL" chapter.)

To ensure that a null is not assigned to a column, you can use the ifnull function, described in this chapter.

Arithmetic

An arithmetic operation combines two or more numeric expressions using the arithmetic operators to form a resulting numeric expression. For details about arithmetic operators, see Arithmetic.

Before performing any arithmetic operation, the DBMS Server converts the participating expressions to identical data types. After the arithmetic operation is performed, the resulting expression has that storage format also. For details, see Default Type Conversion.

Default Type Conversion

When two numeric expressions are combined, the DBMS Server converts as necessary to make the data types of the expressions identical and assigns that data type to the result. The expression having the data type of lower precedence to that of the higher is converted. The order of precedence among the numeric data types is, in highest-to-lowest order:

money
float4
float
i4
i2
i1

For example, in an operation that combines an integer and a floating point number, the integer is converted to a floating point number. If the DBMS Server operates on two integers of different sizes, the smaller is converted to the size of the larger. The conversions are done before the operation is performed.

The following table lists the data types that result from combining numeric data types in expressions:

	i1	i2	i4	float	float4	money
i1	i4	i4	i4	float	float4	money
i2	i4	i4	i4	float	float4	money
i4	i4	i4	i4	float	float4	money
float	float	float	float	float	float4	money
float4	float4	float4	float4	float4	float4	money
money	money	money	money	money	money	money

For example, for the expression

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

the first operator (+) combines a float4 expression (job.lowsal) with a i2 constant (1000). The result is float4. The second operator (*) combines the float4 expression with a i2 constant (12), resulting in a float4 expression.

To convert one data type to another you must use data type conversion functions. For details, see Data Type Conversion Functions.

Arithmetic Errors

To specify error handling for numeric overflow, underflow and division by zero, use the connect statement -numeric_overflow=*option* flag. Error-handling options are:

ignore	No error is issued.
warn	A warning message is issued.
fail (default setting)	An error message is issued and the statement that caused the error is aborted.

This flag can also be specified on the command line for Ingres operating system commands that accept QUEL option flags. For details about QUEL option flags, see the quel command description in the *Command Reference Guide*.

Arithmetic Operations on Dates

QUEL supports the following arithmetic operations for the date data type:

Addition				**Result**
interval	+	interval	=	interval
interval	+	absolute	=	absolute
Subtraction				**Result**
interval		interval	=	interval
absolute		absolute	=	interval
absolute		interval	=	absolute

You cannot multiply or divide date values.

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

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

yields "6 days 5 hours," while

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

yields "4 years 6 months 1 hour 40 minutes."

In the preceding example, 20 minutes and 80 minutes are added and the result is "1 hour 40 minutes." 20 minutes plus 80 minutes is 100 minutes; however, because there are only 60 minutes in one hour, this result is considered to have overflowed the minute time unit. With one exception, whenever you add intervals, the DBMS Server propagates all overflows upward. In the above example, the result is returned as "1 hour 40 minutes." However, days are not propagated to months. For example, if you add 25 days to 23 days, the result is 48 days.

When you subtract intervals or absolute dates, the result is returned in appropriate time units. For example, if you perform the following subtraction:

```
date("2 days") - date("4 hours")
```

the result is "1 day 20 hours."

You can convert date constants into numbers of days relative to an absolute date. For example, to convert today's date to the number of days since January 1, 1900:

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

To convert the interval back to a date:

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

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

In comparisons, a blank (default) date is less than any interval date. All interval dates are less than all absolute dates. Intervals are converted to comparable units before they are compared. For instance, before comparing date("5 hours") and date("200 minutes"), the DBMS Server converts both the hours and minutes to milliseconds internally before comparing the values. Dates are stored in Greenwich Mean Time (GMT). For this reason, "5:00 pm" Pacific Standard Time is equal to "8:00 pm" Eastern Standard Time.

Adding a month to a date always yields the same date in the next month. For example:

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

yields March 1.

If the result month has fewer days, the resulting date is the last day of the next month. For instance, adding a month to May 31 yields June 30, instead of June 31, which does not exist. Similar rules hold for subtracting a month and for adding and subtracting years.

Functions

This section describes the QUEL functions. *Scalar* functions take single-valued expressions as their argument. *Aggregate* functions take a set of values (for example, the contents of a column in a table) as their argument.

Scalar

There are four types of scalar functions:

- data type conversion
- numeric
- string
- date

The scalar functions require either one or two single-value arguments. Scalar functions can be nested to any level.

Data Type Conversion Functions

The following table lists the data type conversion functions:

Name	Operand Type	Result Type	Description
c(*expr* [, *len*])	any	c	Converts argument to c string. If you specify the optional length argument, the function returns the leftmost *len* characters. *Len* must be a positive integer value that does not exceed the length of the *expr* string.
char(*expr* [, *len]*)	any	char	Converts argument to char string. If you specify the optional length argument, the function returns the leftmost *len* characters. *Len* must be a positive integer value that does not exceed the length of the *expr* string.
date(*expr*)	c, text, char, varchar	date	Converts a c, char, varchar or text string to internal date representation.
dow(*expr*)	date	c	Converts an absolute date into its day of week (for example, "Mon," "Tue"). The result length is 3.

Name	Operand Type	Result Type	Description
float4(*expr*)	c, char, varchar, text, float, money, and the integer data types	float4	Converts the specified expression to float4.
float8(*expr*)	c, char, varchar, text, float, money, and the integer data types	float	Converts the specified expression to float.
hex(*expr*)	varchar, char, c, text	varchar	Returns the hexadecimal representation of the argument string. The length of the result is twice the length of the argument, because the hexadecimal equivalent of each character requires two bytes. For example, hex("A") returns "61" (ASCII) or "C1" (EBCDIC).
int1(*expr*)	c, char, varchar, text, float, money, and the integer data types	i1	Converts the specified expression to i1. Floating point values are truncated. Numeric overflow occurs if the integer portion of a floating point value is too large to be returned in the requested format.
int2(*expr*)	c, char, varchar, text, float, money, and the integer data types	i2	Converts the specified expression to i2. Floating point values are truncated. Numeric overflow occurs if the integer portion of a floating point value is too large to be returned in the requested format.
int4(*expr*)	c, char, varchar, text, float, money, and the integer data types	i4	Converts the specified expression to i4. Floating point values are truncated. Numeric overflow occurs if the integer portion of a floating point value is too large to be returned in the requested format.
money(*expr*)	c, char, varchar, text, float, and the integer data types	money	Converts the specified expression to internal money representation. Rounds floating point values, if necessary.
text(*expr* [, *len*])	any	text	Converts argument to text string. If you specify the optional length argument, the function returns the leftmost *len* characters. *Len* must be a positive integer value that does not exceed the length of the *expr* string.

Name	Operand Type	Result Type	Description
varchar(*expr* [, *len*])	any	varchar	Converts argument to varchar string. If you specify the optional length argument, the function returns the leftmost *len* characters. *Len* must be a positive integer value that does not exceed the length of the *expr* string.

If you omit the optional length parameter, the length of the result returned by data type conversion functions c(), char(), varchar(), and text() are as follows:

Data Type of Argument	Result Length
c	Length of operand
char	Length of operand
date	25 characters
float & float4	11 characters; 12 characters on IEEE computers
integer1 (smallint)	6 characters
integer	6 characters
integer4	13 characters
long varchar	Length of operand
money	20 characters
text	Length of operand
varchar	Length of operand

Numeric

The numeric functions are listed in the following table:

Name	Operand Type	Result Type	Description
abs(n)	all numeric types and money	same as *n*	Absolute value of *n*
atan(n)	all numeric types and money	float	Arctangent of *n*; returns a value from (-pi/2) to pi/2

Name	Operand Type	Result Type	Description
cos(n)	all numeric types and money	float	Cosine of *n*; returns a value from -1 to 1
exp(n)	all numeric types and money	float	Exponential of *n*
log(n)	all numeric types and money	float	Natural logarithm of *n*
mod(n,b)	i4, i2, i1	same as *b*	*n* modulo *b*. The result is the same data type as *b*
sin(n)	all numeric types and money	float	Sine of *n*; returns a value from -1 to 1
sqrt(n)	all numeric types and money	float	Square root of *n*

For trigonometric functions (atan(), cos(), and sin()), you must specify arguments in radians. To convert degrees to radians, use the following formula:

```
radians = degrees/360 * 2 * pi
```

To obtain a tangent, you must divide sin() by cos().

String

String functions perform a variety of operations on character data**.** String functions can be nested; for example,

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

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

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

concatenates "x.lastname" with a comma and concatenates "x.firstname" with the first concatenation result. You can also use the **+** operator to concatenate strings:

```
x.lastname + ", " + x.firstname
```

The following table lists the string functions supported by QUEL. The expressions *c1* and *c2*, representing the arguments, can be any of the string types, except where noted. The expressions *len* and *nshift* represent integer arguments.

Name	Result Type	Description
concat(*c1,c2*)	Any character data type	Concatenates one string to another. The result size is the sum of the sizes of the two arguments. If the result is a c or char string, it is padded with blanks to achieve the proper length. To determine the data type results of concatenating strings, see the following table, which shows the results of string concatenation.
left(*c1,len*)	Any character data type	Returns the leftmost *len* characters of *c1*. If the result is a fixed-length c or char string, it is the same length as *c1*, padded with blanks. The result format is the same as *c1*.
length(*c1*)	i2	If *c1* is a fixed-length c or char string, returns the length of *c1* without trailing blanks. If *c1* is a variable-length string, returns the number of characters actually in *c1.*
locate(*c1,c2*)	i2	Returns the location of the first occurrence of *c2* within *c1,* including trailing blanks from *c2*. The location is in the range 1 to size(*c1*). If *c2* is not found, the function returns size(*c1*) + 1. (The function size() is described below, in this table.) If *c1* and *c2* are different string data types, *c2* is coerced into *c1*'s datatype.
lowercase(*c1*)	Any character data type	Converts all upper case characters in *c1* to lower case.
pad(*c1*)	text or varchar	Returns *c1* with trailing blanks appended to *c1*; for instance, if *c1* is a varchar string that holds fifty characters but only has two characters, "pad(*c1*)" appends 48 trailing blanks to c1 to form the result.
right(*c1,len*)	Any character data type	Returns the rightmost *len* characters of *c1*. Trailing blanks are not removed first. If *c1* is a fixed-length character string, the result is padded to the same length as *c1*. If *c1* is a variable-length character string, no padding occurs. The result format is the same as *c1*.
shift(*c1,nshift*)	Any character data type	Shifts the string *nshift* places to the right if *nshift* > 0 and to the left if *nshift* < 0. If *c1* is a fixed-length character string, the result is padded with blanks to the length of *c1*. If *c1* is a variable-length character string, no padding occurs. The result format is the same as *c1.*
size(*c1*)	i2	Returns the *declared size* of *c1* without removal of trailing blanks.

Name	Result Type	Description
squeeze(*c1*)	text or varchar	Compresses white space. White space is defined as any sequence of blanks, null characters, newlines (line feeds), carriage returns, horizontal tabs and form feeds (vertical tabs). Trims white space from the beginning and end of the string, and replaces all other white space with single blanks. This function is useful for comparisons. The value for *c1* must be a string of variable-length character string data type (not fixed-length character data type). The result is the same length as the argument.
trim(*c1*)	text or varchar	Returns *c1* without trailing blanks. The result has the same length as *c1.*
notrim(*c1*)	Any character string variable	Retains trailing blanks when placing a value in a varchar column. You can only use this function in an embedded QUEL program. For more information, see the *Embedded QUEL Companion Guide*.
uppercase(*c1*)	any character data type	Converts all lower case characters in *c1* to upper case.
charextract(*c1,n*)	char	Returns the *nth* byte of *c1.* If n is larger than the length of the string, the result is a blank character.
soundex	any character data type	Returns a four-character field that can be used to find similar sounding strings. For example, SMITH and SMYTHE produce the same soundex code. If there are less than three characters, the result is padded by trailing zero(s). If there are more than three characters, the result is achieved by dropping the rightmost digit(s). This function is useful for finding like-sounding strings quickly. A list of similar sounding strings can be shown in a search list rather than just the next strings in the index.

The following table shows the results of concatenating expressions of various character data types:

1st String	2nd String	Trim Blanks		Result Type
		from 1st?	from 2nd?	
C	c	Yes	–	c
	text	Yes	–	c
	char	Yes	–	c
	varchar	Yes	–	c

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

When concatenating more than two operands, the DBMS Server evaluates expressions from left to right. For example: varchar + char + varchar is evaluated as (varchar+char)+varchar. To control concatenation results for strings with trailing blanks, use the trim, notrim, and pad functions.

Date

QUEL supports functions that derive values from absolute dates and from interval dates. These functions operate on columns that contain date values. Some date functions require you to specify a unit parameter; unit parameters must be specified using a quoted string. The following table lists valid unit parameters:

Unit	How Specified
Second	second, seconds, sec, secs
Minute	minute, minutes, min, mins
Hour	hour, hours, hr, hrs

Day	day, days
Week	week, weeks, wk, wks
Month	month, months, mo, mos
Quarter	quarter, quarters, qtr, qtrs
Year	year, years, yr, yrs

The following table lists the date functions:

Name	**Format (Result)**	**Description**
date_trunc(*unit,date*)	date	Returns a date value truncated to the specified *unit.*
date_part(*unit,date*)	integer	Returns an integer containing the specified (*unit*) component of the input date.
date_gmt(*date*)	Any character data type	Converts an absolute date into the Greenwich Mean Time character equivalent with the format *yyyy_mm_dd hh:mm:ss* GMT. If the absolute date does not include a time, blanks are returned for the time portion of the result. For example, the query retrieve (dcolumn=date_gmt("1-1-93 10:13 PM PST")) returns the following value: 1998_01_01 06:13:00 GMT
gmt_timestamp(*s*)	Any character data type	Converts *s* (where *s* is an integer that represents the number of seconds since January 1, 1970 GMT) into the GMT character equivalent with the format *yyyy_mm_dd hh:mm:ss* GMT. For example, the query retrieve (dcolumn = gmt_timestamp(123456)) returns the following value: 1970_01_02 10:17:36 GMT

Name	Format (Result)	Description
interval (*unit, date_interval*)	float	Converts a date interval into a floating-point constant expressed in the unit of measurement specified by unit. The interval function assumes that there are 30.436875 days per month and 365.2425 days per year when using the mos, qtrs, and yrs specifications. For example, the query retrieve (icolumn = interval("days", "5 years")) returns the following value: 1826.213
_date(*s*)	Any character data type	Returns a 9-character string giving the date *s* seconds after January 1, 1970 GMT. The output format is *"dd-mmm-yy".* For example, the query retrieve (dcolumn = _date(123456)) returns the following value: 2-jan-1970
_time(*s*)	Any character data type	Returns a 5-character string giving the time *s* seconds after January 1, 1970 GMT. The output format is "*hh:mm*" (seconds are truncated). For example, the query retrieve (tcolumn = _time(123456)) returns the following value: 02:17

Using the Date_trunc

You can use the date_trunc function to group all the dates within the same month or year, and so forth. For example:

```
date_trunc("month",date("23-oct-1998 12:33"))
```

returns "1-oct-1998", and

```
date_trunc("year",date("23-oct-1998"))
```

returns "1-jan-1998".

Truncation takes place in terms of calendar years and quarters ("1-jan," "1-apr," "1-jun" and "1-oct").

To truncate in terms of a fiscal year, you must offset the calendar date by the number of months between the beginning of your fiscal year and the beginning of the next calendar year ("6 mos" for a fiscal year beginning July 1, or "4 mos" for a fiscal year beginning September 1):

```
date_trunc("year",date+"4 mos") - "4 mos"
```

Weeks start on Monday. The beginning of a week for an early January date falls into the previous year.

Using the Date_part

This function is useful in set functions and in assuring correct ordering in complex date manipulation. For example, if date_field contains the value 23-oct-1998,

```
date_part("month",date(date_field))
```

returns a value of "10" (representing October), and

```
date_part("day",date(date_field))
```

returns a value of "23".

Months are numbered 1 to 12, starting with January. Hours returned according to the 24-hour clock. Quarters are numbered 1 through 4. Week 1 begins on the first Monday of the year. Dates before the first Monday of the year are considered to be in week 0.

Aggregate

An aggregate function returns a single value based on the contents of a column. Aggregate functions are also called "set" functions. Aggregate functions can be nested.

The syntax for QUEL aggregate functions is as follows:

*afunct***(***expr* [**by** *expr*{**,** *expr*}] **[[only] where** *qual*]**)**

where *afunct* denotes an aggregate function, *expr* an expression representing a column or host variable, and *qual* a qualification. (Qualifications are explained below).

The following table lists aggregate functions:

Function	Data Type of Result	Value Returned
count()	i4	Number of entries in column.
countu()	i4	Number of unique entries in column.
sum()	i4, float8, money	Sum of values in column.
sumu()	i4, float8, money	Sum of unique values in column.
avg()	float8, money	Average of values in column.
avgu()	float8, money	Average of unique values in column.
max()	All types	Maximum value in column.
min()	All types	Minimum value in column.
any()	i2	Returns 1 if any rows satisfy the condition expressed by the argument; 0 if no rows satisfy the condition.

Aggregate Functions Using the Where and By Clauses

Aggregate functions typically evaluate a column and return a single value (for example, avg(e.age) returns the average of all values in the "age" column of table "e"). This section describes how you can use the where and by clauses to modify the results returned by aggregate functions.

The where clause enables you to qualify (filter) the set of values used to determine the result of the aggregate function. For example,

```
sum(j.salary where j.salary > 1500
```

returns the sum of all salaries from table j that exceed 1500.

The by clause causes the function to return a *set* of results, as opposed to a single result. One result is returned for each grouping specified by the by clause. Think of by as meaning "*for each*." For example,

```
avg(e.age by e.dept)
```

returns an average age *for each* department in table e.

You can combine the by and where clauses:

```
avg(e.age by e.dept where e.job=1023)
```

returns the average age, by department, for employees who have a job code of 1023.

You can use the only where format to skip zero results. For example,

```
count(emp.salary by emp.dept where emp.salary > 10000)
```

returns a value for *every* department, but

```
count(emp.salary by emp.dept
only where emp.salary > 10000)
```

returns a value only when there are departments containing employees earning more than 10000.

If you use a by clause on a column that contains nulls, the DBMS Server returns a single result for the rows that contain null in the column specified in the by clause–in other words, nulls are grouped.

The result of the only where clause is affected by the set aggregate project|noproject statement. For more information, see Set in the "QUEL and EQUEL Statements" chapter.

When an aggregate is applied to a nullable column, any nulls are disregarded in computing the aggregate. For example, for the following table "temp":

x
0
1
1
2
null
null

The statement

```
retrieve (c = countu(temp.x))
```

yields

c
3

Several variables can appear within a single aggregate function. For example,

```
avg(j.salary by e.dept where e.job=j.jid)
```

Ifnull

The ifnull function enables you to specify a value other than a null that is returned to your application when a null is encountered. The ifnull function is specified as follows:

ifnull(*v1*,*v2*)

If the value of the first argument is not null, ifnull returns the value of the first argument. If the first argument evaluates to a null, ifnull returns the second argument.

The sum, avg, max, and min aggregate functions return 0 if the argument to the function evaluates to an empty set. To receive a specified value when the function evaluates to an empty set, use the ifnull function, as in this example:

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

Ifnull returns the value of the expression "sum(employee.salary)/25" unless that expression is null. If that expression is null, the ifnull function returns –1.

If the arguments are of the same data type, the result is of that data type. If the two arguments are of different data types, they must be of comparable data types. For a description of comparable data types, see Assignment.

When the arguments are of different but comparable data types, the DBMS Server uses the following rules to determine the data type of the result:

- The result type is always the higher of the two data types; the order of precedence of the data types is as follows:

  ```
  date > money > float4 > float > i4 > i2 > i1
  ```

  and

  ```
  c > text > char > varchar
  ```

- The result length is taken from the longest value. For example,

  ```
  ifnull (varchar (5), c10)
  ```

  results in **c10**.

The result is nullable if either argument is nullable. The first argument is not required to be nullable, though in most applications it is nullable.

Qualifications

The term *qualification* refers to a condition in a query that is applied to the rows of a table to extract the desired subset of rows.

Comparison Operators

A comparison operator is a binary operator that takes two expressions as operands. The expressions must both be numeric, character (any of the four character types), money or date types. The following operators are recognized in QUEL:

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

All comparison operators are of equal precedence. When comparisons are made between **c** strings or between a **c** string and a string of any of the other types, all blanks are ignored. When comparisons are made between text, char, or varchar strings, all blanks are significant.

Nullable and non-nullable data types can be compared. If one or both of the values is null, the comparison returns the value unknown.

Partial Match Specification

QUEL supports special characters for use with comparison operators (in particular, the equals operator) to indicate partial matches of character string (c, char, varchar and text) data. These characters allow the following partial match specifications:

*	matches any string of zero or more characters
?	matches any single character
[..]	matches any of the characters in the brackets

QUEL allows any of these special characters singly or in combination to specify partial match criteria, as the following examples illustrate:

```
e.ename="*"
```

matches any value in "e.ename". If e.ename is nullable, * does not match NULL values.

```
e.ename="E*"
```

matches any value beginning with "E".

```
e.ename="*ein"
```

matches any value ending with "ein".

```
e.ename="*[aeiou]*"
```

matches any value with at least one vowel.

```
e.ename="Br???"
```

matches any five-character value beginning with "Br".

```
e.ename="[A-J]*"
```

matches any value beginning with A, B, C, ..., J.

```
e.ename="[N-Z]???"
```

matches any four-character value beginning with N, O, P, ..., Z.

Blanks must not be embedded in bracketed expressions such as "[A-J]*" or "[N-Z]???."

The special meaning of these characters can be disabled in a clause by preceding them with a backslash character (\). Thus, "*" refers to the asterisk character. However, in an assignment (as opposed to a clause), the special characters do not perform a partial match specification, as in the following:

```
jtitle  =  "**accountant**"
```

Because the fragment above assigns a value "**accountant**" to the column "jtitle," the asterisks need no escape treatment with the backslash. However, to retrieve the value so assigned requires the following syntax:

```
j.jtitle="\*\*accountant\*\*"
```

Is Null Comparison

The is null predicate has the following syntax:

is [**not**] **null**

The is null predicate explicitly tests for a null value. A null is not greater than or less than anything and is not equal to anything, even another null value. For example, the predicate

```
where  column1=column2
```

does not evaluate to "true" even if both columns are null. To explicitly test a column for a null value, you must use the **is null** predicate. Similarly, an explicit test can be made for the absence of a null value by specifying the **is not null** predicate:

```
column1  is  not  null
```

Clauses

A clause has the form

expr comp_op expr

where *comp_op* is a comparison operator. A clause can be enclosed in parentheses without affecting its interpretation, as in the following examples:

```
(e.age < 50)
((j.salary*12)  >=  20000)
```

A clause returns the truth value true, false or unknown.

Logical Operators

The following Boolean logical operators are recognized in QUEL:

not (negation)

and (conjunction)

or (disjunction)

These operators take and return truth functions (true, false or unknown).

Not has the highest precedence of the three operators; **and** and **or** have equal precedence. Parentheses can be used to override the default order of processing; by default logical operators are processed from left to right.

General Qualification

You can use the following constructions to form a where clause:

- not qual
- qual or qual
- qual and qual
- (qual)

where *qual* is a condition that qualifies a query. For example:

```
where e.age <= 50
where (e.age <=50) and (j.salary >= 40000) and
(e.job=j.jid)
```

These examples apply boolean operators to the results of each predicate. If boolean operators are not specified, the result of the *qual* condition is the result of the predicate. Not(true) is false, not(false) is true, not(unknown) is unknown. AND and OR are defined by the tables that follow.

The following table shows the results of the AND Logical Operator:

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

The following table shows the results of the OR Logical Operator:

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

Chapter 5: Embedded QUEL

This chapter discusses the following topics:

- Statement syntax
- Program structure
- Host language variables
- Cursors
- Transaction processing
- Program status information
- Error handling
- The retrieve statement
- Repeat queries

When the syntax of an EQUEL statement depends on the host language, you are referred to the *Embedded QUEL Companion Guide*.

The examples in this section indicate host language statements by using language-independent pseudocode. Pseudocode statements are italicized and enclosed in curly braces as shown below.

```
{
host language statement
}
```

General Syntax of EQUEL Statements

All EQUEL statements must be preceded by a pair of number signs:

```
##EQUEL_statement
```

The number signs must be the leftmost characters on the line, except in languages that require line numbers. (If you are programming in a language that uses line numbers, see the *Embedded QUEL Companion Guide* for the correct format.) The EQUEL preprocessor ignores lines that do not begin with "##".

EQUEL statements can be continued across multiple lines; each continuation line must begin with "##". For example:

```
##retrieve (sal = e.salary, ename = e.empname)
##where e.empnum = 23
```

To continue a string literal to the next line, precede the continuation line with a backslash () and omit the "##". For example:

```
##append to employee (empname = "john jones")
```

EQUEL does not require statement terminators. However, if your host language uses a statement terminator, you can use it to terminate EQUEL statements.

EQUEL comments can appear only on lines that begin with "##". EQUEL comments are delimited by "/*" and "*/". You can also use the host language format to place comments in an EQUEL program.

String literals in EQUEL statements must follow the rules of the host language.

Basic Structure of EQUEL Programs

A typical EQUEL application performs the following steps:

- Connect with a database.
- Execute queries against the database.
- Disconnect from the database.

In general, you can mix EQUEL and host language statements. Specific restrictions are discussed in this chapter. The following example shows a simple EQUEL program that retrieves and prints the salary and name for employee 23:

```
begin program
##   ename      character_string (26)
##  sal         float
##  /* connect... */
##  ingres "personnel"
##   /* execute queries... */
##  range of e is employee
##  retrieve (sal = e.salary, ename = e.empname)
##       where e.empnum = 23
##  {
                print ename, sal
##  }
##   /* disconnect */
##  exit
end program
```

Host Language Variables

The following section discusses the use of host language variables in an EQUEL application.

Variable Declaration

EQUEL statements use host language variables to transfer data between a database and an application program. In addition, you can use host variables to specify the names of cursors, tables, views, and columns. You must declare host variables to EQUEL before you can use them in EQUEL statements. (If a variable is not used in an EQUEL statement, it does not need to be declared to EQUEL.) To declare a variable to EQUEL, precede the variable declaration with ##. For example,

```
## char employee, street, city, zipcode
```

The *Embedded QUEL Companion Guide* lists the data types acceptable to EQUEL and discusses conversion between host language and QUEL data types. EQUEL restricts you to these data types for variables you use in EQUEL statements.

EQUEL automatically converts between host and QUEL data of the same type (numeric or character). However, EQUEL does not convert across data types. For example, you cannot ask EQUEL to return a numeric value in a host character variable. To convert data types, use the QUEL data type conversion functions. For details, see Data Type Conversion Functions in the "Elements of QUEL Statements" chapter.

Dereferencing Column Names

If a host variable declared to EQUEL has the same name as a column, table, or form object in a table, you must precede the column name with a number sign (#) (*dereference* it). Dereferencing tells the EQUEL preprocessor to treat the flagged item as a column (or table or form object) name, not a host variable.

For example, if table "employee" has a column named "salary", and the your application has a variable also named "salary", you must use the following retrieve statement to read data from the column into the host variable of the same name:

```
##retrieve (salary = e.#salary)
```

Scope of Variables

EQUEL obeys host language conventions for the scope of variables. The scope of an EQUEL-declared variable opens at its declaration. The variable is visible to the preprocessor from that point to the end of the file, unless an EQUEL statement closes the scope of the variable. For information about statements that open and close the scope of variables, see the *Embedded QUEL Companion Guide*.

Include Statement

The include statement allows you to include external files in your source code. The syntax of the include statement is

```
##include filename
```

For example, you can use include to incorporate a file of EQUEL variable declarations:

```
begin program
## include "myvars.dec"
## /*
## ** the equel program can reference the data items
## ** declared in myvars.dec
## */
end program
```

For information about the naming conventions for include files, see the *Embedded QUEL Companion Guide*.

Indicator Variables

An *indicator variable* can be associated with a host variable for the following purposes:

- To indicate if a null was retrieved from a column
- To assign a null to a column.
- To indicate if a string retrieved from a column was truncated.

Use the following syntax to associate an indicator variable with a host variable:

host_variable:indicator_variable

If your application program retrieves a null into a host variable, and an indicator variable is not associated with the host variable, the DBMS Server issues a runtime error.

Retrieving Data Using Null Indicators

After you retrieve data into a host variable that is associated with an indicator variable, the indicator variable contains one of the following values:

-1	Value was null. The contents of the host variable are unchanged.
0	Value was not null. The host variable contains the retrieved value.

The following example illustrates the use of an indicator variable. In this example the indicator value is used to detect missing phone numbers, which are listed in a roster as "n/a":

```
##retrieve cursor emp_cursor (name, phone:phone_null, id)
if (phone_null = -1) then
  update_roster(name, "n/a", id)
else
  update_roster(name, phone, id)
end if
```

The following EQUEL statements can include indicator variables in their output target lists:

- retrieve
- retrieve cursor

Setting Values Using Null Indicators

To assign null to a database column, set the indicator variable (associated with the host variable you are writing) to -1 and execute the assignment statement. You can also assign null using the keyword null.

You can use the following statements in conjunction with indicator variables to assign null values:

- append
- replace
- replace cursor

If you attempt to assign a null to an object that is not nullable, the DBMS Server issues a runtime error.

A null indicator variable can accompany a variable used in the where clause of the retrieve, append and replace statements, if you are comparing with nullable columns or expressions.

The following example demonstrates the use of both an indicator variable and the null constant: an indicator variable is used to set "phone" to null (if no phone number was entered), and the null constant is used to set the "comment" field before it is written to the new employee database.

```
read ename, eno, phone from terminal
if phone = "" then
  phone_null = -1
else
  phone_null = 0
end if
##append to newemp (empname = ename,
##  #phone = phone:phone_null,
##  empnum = eno, comment = null)
```

Detecting String Truncation Using Indicator Variables

If you application retrieves a character string into a host variable that is too small to hold the string, the DBMS Server truncates the string to fit into the host variable. If you specify an indicator variable with the host variable, the indicator variable is set to the original length of the data. You can detect truncation by comparing the value of the indicator variable with the length of the string that was retrieved: if the indicator variable is greater than the length, the string was truncated.

Variable Usage and Dynamic Operation of EQUEL Statements

EQUEL allows you to use host language variable to specify many and various parts of EQUEL statements. This powerful feature enables you to write applications that have a great deal of runtime flexibility.

Of course, the data type and use of the host language variable must make sense in the context of the EQUEL statement.

The following are general rules and guidelines:

- Host variables can be used to receive values from tables and status information obtained from the DBMS Server.
- Host variables can be used to specify the following portions of EQUEL statements:

Portion of Statement	Description
Values of constants within expressions	A variable can contain a value to be matched in a database qualification (where clause), a value to be stored in a database column or an operand in a complex expression. The variable must contain a single value of an appropriate data type, and must not be a string containing multiple operands or operators.
Qualifications	A string variable can be used to specify an entire qualification (where clause), including names of range variables and columns, values to be matched, and QUEL functions. This string variable must not contain names of host language functions or other host language variables which are not understood by the database management system. This feature allows considerable flexibility in programs, permitting applications to construct a "where clause" from parameters that the user specifies at runtime.
Names of database objects	The general rule when using variable substitution for database object names, (such as range variables, tables, and columns) is that one variable can substitute for one name in a statement: for example, you cannot assign a string variable a value such as "e.salary"; you must specify the range variable e and the column salary using separate host variables. When using a variable to specify a database name, you can use a single string variable to specify both the network node and database name.
Miscellaneous arguments	In general, constant values for statement arguments can be specified using host variables of the appropriate data type. For example, the components of a with clause on the index and copy statements, and the items in a define permit or a save statement, can be represented by host variables. In the sort clause of a retrieve statement, the sort keys can be specified individually using string variables containing the name of a result column. A string variable can be used to specify the entire *target_list* on the create, copy, define view and declare cursor statements.

- Host variables of short integer types can be used as null indicator variables as described in this chapter.

Host variables cannot be substituted for keywords in EQUEL statements. For details about the parameters that can be specified using host variables, see the "QUEL and EQUEL Statements" chapter.

In the following example, the retrieve statement makes use of host variables. The two host variables are "name" and "sal".

```
## retrieve (name = e.empname, sal = e.salary)
##          where e.empnum = 23
```

In the following example, the variable "eno" is used as an expression in the where clause.

```
## retrieve (ename = e.empname, sal = e.salary)
##          where e.empnum = eno
```

If an embedded retrieve statement returns no rows, the contents of the host variables are not be modified.

Param Statements

The EQUEL param statement allows you to create lists of host variables at runtime, for retrieve and append (and other) operations, rather than hard-coding variables into such statements.

The param feature is not supported for all host languages; it is described in detail in the *Embedded QUEL Companion Guide* for the languages that do support it.

Data Manipulation with Cursors

Cursors return a series of rows to an embedded application, one row at a time, as the result of a retrieve statement. To use cursors, perform the following steps:

- Declare a cursor; when you declare a cursor, you assign it a name and associate the cursor with a retrieve statement.
- Open the cursor.
- Retrieve columns from the next row. The columns you specified in the declare statement are retrieved into the host variables you specify in the retrieve statement.

- If required by your application, replace selected columns from the current row with the contents of the host variables you specify, or delete the current row.
- Close the cursor to terminate processing of the table.

During processing, the row to which the cursor is pointing is referred to as the *current row*. The cursor is advanced by issuing a cursor retrieve statement. The current row is updated by issuing a cursor replace statement.

Example of Cursor Processing

The following example uses a cursor to print the names and salaries of all the employees in the table and set any salaries under $10,000 to $10,000.

```
begin program
## name    character_string(15)
## salary  float
## ingres personnel
## range of e is employee
## declare cursor c1 for
## retrieve
## (e.empname, e.#salary)
## for update of (#salary)
## open cursor c1
loop while more rows
## retrieve cursor c1 (name, salary)
print name, salary
  if salary less than 10000 then
## replace cursor c1 (#salary = 10000)
  end if
/* use the inquire_ingres statement to check endquery
status for end-of-table*/
 end loop
## close cursor c1
## exit
end program
```

Declaring a Cursor

To declare a cursor, you associate a cursor name with a retrieve statement. You must declare a cursor before you can use it. In your source code, the declare statement must appear before the first use of the cursor; the declare statement is used by the preprocessor and does not generate executable code.

The syntax for declaring a cursor is

**declare cursor** *cursor_name* **for**
retrieve_statement
for [**deferred** | **direct**] **update of** *column* {**,** *column*}

The *cursor_name* can be either a literal or a host language character string variable assigned a valid cursor name at runtime. Cursor names must obey the naming conventions described in the "QUEL Data Types" chapter. The retrieve clause used in a declare statement must observe the correct QUEL syntax.

The for update clause allows you to specify the manner in which the DBMS Server updates the tables that are referenced by the cursor. If you intend only to delete rows, you do not need to declare the cursor for update. The default mode for the for update clause is deferred. In deferred mode, the updates you make using the cursor are not written until you close the cursor. Only one cursor can be open for deferred update at any time. In direct mode, the updates you make using the cursor are written immediately. If you write a change that affects the sequence of rows (for example, you modify a key field), the next retrieve statement returns the next row in the new sequence.

No data is retrieved as a result of the declare cursor statement. Data is retrieved when you open the cursor and issue a retrieve cursor statement.

Opening and Closing Cursors

You must open a cursor before you can use it to read, write, or delete data:

open cursor *cursor_name* [**for readonly**]

When you open a cursor, it is positioned before the first row; the first retrieve cursor statement you issue advances the cursor to the first row and return its data. More than one cursor can be open at the same time.

You can use the for readonly clause if you do not intend to write or delete data; for readonly is valid even if the cursor was defined for direct|deferred update. Specifying for readonly can speed up processing. If you attempt to write data using a cursor that was opened readonly, the DBMS Server issues a runtime error.

The close statement terminates processing of a cursor:

close cursor *cursor_name*

A cursor can be opened and closed any number of times; it must be closed, however, before it can be reopened. Closing and reopening a cursor repositions it to the top of the table.

Cursors cannot remain open across transactions; a cursor must be opened and closed within a single transaction.

Open Cursors and Transaction Processing

The only way you can have more than one cursor open at a time is using multi-query transactions (MQT). (Cursors opened for update must be opened in direct mode.) An MQT also allows your program to issue other queries while there are open cursors. No work is committed (written to the database) until the end transaction statement is executed. At this point, all queries since the last begin transaction statement are committed and any open cursors are closed.

The following table summarizes the interaction of EQUEL transaction statements and cursors:

Statement	Effect
begin transaction	Denotes the beginning of a transaction. More than one cursor can be open within a transaction.
end transaction	Commits all cursor updates specified within the transaction, and closes all open cursors.
abort	Undoes all cursor updates within the transaction and closes open cursors.
savepoint *savepoint_name*	Not allowed if cursors are open: cursors must be opened and closed between savepoints.
abort to *savepoint_name*	Undoes all cursor updates performed after the specified savepoint. Closes open cursors.

Retrieving the Data

The retrieve cursor statement reads the next row of data (as specified in the declare statement) into the specified host variables. The syntax is:

**retrieve cursor** *cursor_name* (*variable* {**,** *variable*})

To detect the end of a table, use the inquire_ingres statement to determine the endquery status. For details about inquire_ingres, see Inquire_ingres in the "QUEL and EQUEL Statements" chapter.

The retrieve cursor statement is typically used within a program loop to processes a series of rows; using cursors you can only move forward through rows (or reposition to the top of the table by closing and reopening the cursor).

Fetching Rows Inserted by Other Queries

While a cursor is open, your application can append rows using non-cursor append statements. If these newly inserted rows are inserted after the current cursor position, the rows are or are not be visible to the cursor, depending on the following criteria:

- Updatable cursors

  The newly inserted rows are visible to the cursor.

- Non-updatable cursors

  If the cursor retrieve statement retrieves rows directly from the base table, the newly inserted rows are visible to the cursor. If the retrieve statement manipulates the retrieved rows (for example, includes an sort by clause), the cursor retrieves rows from an intermediate buffer, and cannot detect the newly inserted rows.

Using Cursors to Update Data

To update fields that were retrieved using a cursor, use the replace cursor statement:

replace cursor *cursor_name* (*column* **=** *expression*
{, *column* = *expression*})

The replace cursor statement causes no change in the position of the cursor. A retrieve cursor is required to move the cursor forward one row. If you try to replace the same row twice (without advancing the cursor) and the cursor was opened in deferred mode, the DBMS Server issues a runtime error.

The update affects only the current row, and you can only update columns that were declared in the for...update clause of the declare cursor statement. For details, see Declaring a Cursor.

Using Cursors to Delete Data

To delete a row from a table, use the delete cursor statement:

delete cursor *cursor_name*

This statement deletes the current row. The cursor does not have to be declared for update to use a delete cursor. The cursor must have been positioned to the row using retrieve cursor. Once the row is deleted, a retrieve cursor must be issued to advance the cursor to the next row.

The following example illustrates the use of a cursor to update and delete rows:

```
## name character_string(15)
## salary float
## ingres personnel
## declare cursor c1 for
## retrieve (employee.empname, employee.#salary)
## for update of (#salary)
## open cursor c1
loop while more rows
## retrieve cursor c1 (name, salary)
   print name, salary
  /* Increase salaries of all employees earning less
** than 60,000. */
if salary < 60,000 then
  print "updating", name
## replace cursor c1 (#salary = salary * 1.1)
/* Fire all employees earning more than 300,000. */
else if salary > 300,000 then
  print "terminating ", name
## delete cursor c1
end if
end loop
## close cursor c1
## exit
```

Summary of Cursor Positioning

The following list summarizes the effects of cursor statements on cursor positioning:

Statement	Effect on Cursor
open cursor	Cursor positioned before first row in set.
retrieve cursor	Cursor moves to next row in set. If already on last row, cursor moves beyond the set and its position becomes undefined.
replace cursor	Cursor remains on current row.
delete cursor	Cursor moves to a position after the deleted row (but before the following row).
close cursor	Cursor and set of rows become undefined.

For more information about cursors, see the *Embedded QUEL Companion Guide*.

Dynamically Specified Cursor Names

The following example illustrates the use of host variables to dynamically declare cursor names, and the use of a recursive routine to scan a table that contains a "tree" structure (in this example, an organization chart).

In this example, the table "orgchart" contains three columns: employee name, title, and the name of the employee's manager. The program uses a subroutine that displays the employees that report to a manager. If an employee is also a manager, the subroutine calls itself to list the employees he or she manages.

The subroutine declares a cursor for each level it scans. The cursor name is defined as "C" plus the number of the level being scanned (C1, C2, and so on).

```
## character-string ename(25)
## integer level
## ingres "mydatabase"
/* First, print the president's name */
## retrieve (ename=orgchart.employee)
## where orgchart.title="president"
print "the president is ", ename
/* initialize level for recursive calls */
level=0
## begin transaction
    printorg(level, ename)
##  end transaction

## /*****************************
## * display employees **
## * for each manager **
## *************************/
printorg(alevel, amanager)
## character amanager(25), cursorname(2), cname(25),
## character title(25), cmanager(25)
## integer alevel, end_of_query, ecount
## /* is this employee a manager? */
## retrieve (ecount=count(orgchart.manager
## where orgchart.manager=amanager))
## /* no, return */
if ecount=0
  return
endif
cursorname = "c" + char(alevel+1)
## declare cursor cursorname for retrieve
## (orgchart.employee,
## orgchart.title,
## orgchart.manager)
## where orgchart.manager=amanager
## open cursor cursorname
/* cursor loop reads all employees for manager */
end_of_query=0
loop while end_of_query = 0
## retrieve cursor cursorname (cname, ctitle, cmanager)
## inquire_ingres(end_of_query=endquery)
if end_of_query = 0
    indent to appropriate level, print cname, ctitle
    /* see if this employee is a manager */
    call printorg(alevel+1, cname)
    end if
  end while loop
##  close cursor cursorname
return
```

Cursors and Retrieve Loops Compared

Use cursors:

- When a program needs to scan a table to update or delete rows
- When a program requires access to other tables (or cursors) while processing rows
- When more than one table needs to be scanned simultaneously ("parallel query")
- When more than one table needs to be scanned in a nested fashion, for example, in a master-detail application

Use retrieve loops if the program is scanning the rows to:

- Generate a report, or
- Accumulate general statistics

For straightforward reading operations, the retrieve loop runs faster than a cursor. However, you cannot execute other queries inside a retrieve loop.

The following example shows the use of a retrieve loop and a cursor to scan a table:

```
begin program
## ename           character_string(21)
## salary          float
## eno, thatsall   integer
## ingres "personnel"
## range of e is employee
## /* retrieve loop */
## retrieve (ename = e.empname, eno = e.empnum,
## salary = e.#salary)
## sort by #ename
## {
     print ename, eno, salary
## }

## /* cursor retrieve */
## declare cursor c1 for
##   retrieve (e.empname, e.empnum, e.#salary)
##   sort by empname
## open cursor c1
   loop until thatsall = 1
##    retrieve cursor c1 (ename, eno, salary)
##    inquire_ingres (thatsall = endquery)
      if thatsall = 0 then
        print ename, eno, salary
      end if
     end loop
## close cursor c1
## exit
end program
```

Transactions

A *transaction* is one or more QUEL statements that are processed as a single database action. The effects of a transaction on the database become permanent and visible to other users when the transaction is committed.

Your application program can abort (reverse the effects) of some or all of the statements within a multi-query transaction (MQT). The ability to execute groups of statements as a single transaction, and to selectively abort transactions, enables you to ensure that your applications preserve the consistency of the data in the database.

The DBMS Server insures that simultaneously executing transactions do not interfere with each other–this is called "concurrency control." For more information about concurrency issues, see Deadlock: Detection, Avoidance, and Handling.

Transaction Statements

EQUEL's transaction-controlling statements are:

- abort

  Terminates an MQT without committing (updating the database)

- abort to savepoint_name

  Rolls back all statements executed after the specified savepoint

- begin transaction

  Begins an MQT

- end transaction

  Ends an MQT and commits the transaction's effects to the database.

- savepoint savepoint_name

  Declares a savepoint

For details about these statements, see the "QUEL and EQUEL Statements" chapter.

Defining Transactions

One or more QUEL statements enclosed within a begin transaction-end transaction block constitutes a multi-query transaction (MQT). Any QUEL statement not within a begin transaction-end transaction block is a single-query transaction (SQT).

MQTs guarantee the atomic execution of a group of QUEL statements. Within MQTs you can declaring savepoints, which enable you to partially undo the effects of a transaction without aborting the transaction.

The following QUEL statements must not appear within an MQT:

- begin transaction
- end transaction
- set lockmode

Committing Transactions

When a transaction is committed, its effects on the database are made permanent and visible to other users. Before a transaction is committed, none of its updates to the database are visible to other users, and the transaction can be aborted. An SQT is committed upon execution (barring errors). An MQT is committed when the end transaction statement is executed.

Note: Under certain circumstances, the effects of an uncommitted transaction are visible to other users. For details, see Set Lockmode Option in the "QUEL and EQUEL Statements" chapter.

Aborting Transactions

At any time before an end transaction statement commits an MQT, the transaction can be aborted, either by the application program (using an abort statement) or by the DBMS Server (under specific circumstances). When a transaction is aborted, all effects of the transaction on the database are rolled back. No other transactions in progress are adversely affected.

MQTs can be aborted in the following ways:

- Program abort

  The QUEL statement abort terminates an MQT.

- Log file too full

  When the log file becomes too full (80% is the default), the DBMS Server begins to abort the oldest transactions to free up space in the log file. (To avoid forced aborts, allocate sufficient space for the log file.)

- Deadlock

  When the DBMS Server detects deadlock, it aborts one transaction to end the deadlock. For an explanation of deadlock, see Deadlock: Detection, Avoidance, and Handling.

- Exit statement

  Exiting the database with an EQUEL exit statement aborts any in-progress MQT.

Savepoints and Partial Transaction Aborts

The savepoint statement establishes a point within an MQT to which the transaction can be aborted. This enables your application to partially undo the effects of a transaction instead of aborting the entire transaction. All database changes performed by the transaction after the savepoint are rolled back. All changes preceding the savepoint remain.

If the same savepoint name is used in multiple savepoint statements within an MQT, the most recently executed savepoint is always in effect. There is no limit to the number of savepoint declarations allowed within a transaction.

Interrupt and Timeout Handling in Transactions

Any user action which aborts an EQUEL program also causes the DBMS Server to abort any transaction in progress. Termination of EQUEL programs in this manner is strongly discouraged.

If an application times out while waiting for a lock, the DBMS Server displays an error message and aborts any statement in progress. A timeout error during an MQT does not abort the transaction. For details about timeout, see Set Lockmode Option in the "QUEL and EQUEL Statements" chapter.

Deadlock: Detection, Avoidance, and Handling

Deadlock occurs when each of two transactions has locked some portion of a database that the other transaction requires. Neither transaction releases the part of the database it has until it gets the other part. This standoff brings processing to a halt.

The DBMS Server detects deadlock, aborts one of the transactions, and returns an error message to the process whose transaction was aborted.

You cannot guarantee deadlock-free processing. However, you can include appropriate handling of deadlock within your program. (For example, if the application detects deadlock, it restarts the transaction.)

The following example is an EQUEL program that handles general errors in a collection of single statements. All detected errors suspend program execution with the exception of deadlock, which resumes execution at the statement that caused deadlock.

The following is a simple deadlock handling example:

```
begin program
## /*
## ** an equel program that performs a series of appends
## ** and handles ingres errors, including deadlock,
## ** within a single-query transaction.
## */
## ingerr, inum  integer
## ingres "personnel"
## create item (number = i4)
inum = 0
loop until inum = 9
    inum = inum + 1
##  append to item (number = inum)

## /*
## ** if an ingres error occurred, then report the error
## ** and break out of the loop if the error was
## ** something
## ** other than deadlock. if the error was deadlock
## ** then resume with the append that encountered the
## ** deadlock.
## **
## ** the error number for deadlock is 4700.
## */
##   inquire_ingres (ingerr = errorno)
    if ingerr != 0 then
         if ingerr != 4700 then
            print "error number ", ingerr, "on append ", inum
            break loop;
        else
##          /*
##          ** deadlock - try again without incrementing
##          ** the counter
##          */
            inum = inum + 1
        end if
    else
            print "append ", inum, "succeeded"
    end if
 end loop
## exit
```

Another approach to handling deadlock is to suppress the error message and restart the transaction without notifying the user. This approach requires the use of an error handler declared with iiseterr(). For an example of this approach, see the *Embedded QUEL Companion Guide*.

Program Status Information

The following features enable you to obtain QUEL status information:

- The inquire_ingres statement

  Returns runtime information about the status of programs and the results of queries.

- The dbmsinfo() function

  Returns runtime information about the current database session.

For a detailed description of the inquire_ingres statement, see Inquire_ingres in the "QUEL and EQUEL Statements" chapter. For details about the dbmsinfo function, see The Dbmsinfo() Function in this chapter.

The Inquire_ingres Statement

An example of the use of the inquire_ingres statement follows:

```
begin program
## rcount, errno   integer
## errmsg          character_string(256)
## ingres   "personnel"
## append to employee (empnum = 12,
##  empname = "john smith", salary = 10000)
## /* find out if an error occurred while appending */
## inquire_ingres (rcount = rowcount, errno = errorno,
## errmsg = errortext)
## /* if error occurred, print its number and message */
  if errno > 0 then      print "ingres error", errorno, "occurred"
    print errmsg
## /* tell the user whether or not a row was added */
 else if rcount > 0 then
    print "row successfully appended"
 else
    print "integrity violation or duplicate record"
 end if
## exit
end program
```

The rowcount value is useful for detecting integrity violations.

The Dbmsinfo() Function

Dbmsinfo() is a function that returns a string containing information about the current session. You can use this statement in the Terminal Monitor or in an embedded QUEL application. The dbmsinfo() statement has the following syntax:

dbmsinfo("*request_name*")

For example, to find out which release of Ingres you are using, enter:

```
retrieve (x=dbmsinfo("_version"))
```

The following table lists valid *request_name*:

Request Name	Description
autocommit_state	Returns 1 if autocommit is on; 0 if autocommit is off.
_bintim	Returns the current time and date in an internal format, represented as the number of seconds since January 1, 1970 00:00:00 GMT.
_bio_cnt	Returns the number of I/Os to and from the front-end client (application) that created your session.
collation	Returns the collating sequence defined for the database associated with the current session. This returns blanks if the database is using the collating sequence of the machine's native character set, such as ASCII or EBCDIC.
_cpu_ms	Returns the CPU time for your session, in milliseconds.
cursor_deferred_update	Returns "Y" if the default cursor mode is deferred; "N" otherwise. The default cursor mode is specified when the DBMS Server is started.
cursor_direct_update	Returns "Y" if the default cursor mode is direct; "N" otherwise. The default cursor mode is specified when the DBMS Server is started.
database	Returns the database name.
dba	Returns the user name of the database owner.
dbms_bio	Returns the number of buffered I/O requests for all connected sessions.
dbms_cpu	Returns the cumulative CPU time for the DBMS Server, in milliseconds, for all connected sessions.
dbms_dio	Returns the number of direct I/O requests for all connected sessions.
db_delimited_case	Returns "LOWER" if delimited identifiers are translated to lower case, "UPPER" if delimited identifiers are translated to upper case, or "MIXED" if the case of delimited identifiers is not translated. For details about delimited identifiers, see the *SQL Reference Guide*.
db_name_case	Returns "LOWER" if regular identifiers are translated to lower case, or "UPPER" if regular identifiers are translated to upper case.
_dio_cnt	Returns the number of disk I/O blocks for your session.
_et_sec	Returns the elapsed time for session, in seconds.
flatten_aggregate	Returns "Y" if the DBMS Server is configured to flatten queries involving aggregate subselects; "N" otherwise. (Query flattening options are specified when the DBMS Server is started.)
flatten_none	Returns "Y" if query flattening is disabled. (Query flattening options are specified when the DBMS Server is started.)

Request Name	Description
flatten_optimize	Returns "Y" if the DBMS Server is configured to flatten queries wherever possible; "N" otherwise. (Query flattening options are specified when the DBMS Server is started.)
flatten_singleton	Returns "Y" if the DBMS Server is configured to flatten queries involving singleton subselects; "N" otherwise. (Query flattening options are specified when the DBMS Server is started.)
initial_user	Returns the user identifier in effect at the start of the session.
language	Returns the language used in the current session to display messages and prompts.
on_error_state	Returns the current setting for transaction error handling: "rollback transaction" or "rollback statement". To set transaction error handling, use the set session with on_error statement.
_pfault_cnt	Returns the number of page faults for server.
query_language	Returns "sql" or "quel".
security_priv	Returns "Y" if the effective user has the security privilege, or "N" if the effective user does not have the security privilege.
server_class	Returns the class of DBMS server, for example "ingres".
session_id	Returns the internal session identifier in hexadecimal.
session_user	Returns the session's current effective user ID.
system_user	Returns the system user ID.
terminal	Returns the terminal address.
transaction_state	Returns 1 if presently in a transaction, 0 if not.
update_rowcnt	Returns "qualified" if inquire_ingres(rowcount) returns the number of rows that qualified for change by the last query, or "changed" if inquire_ingres(rowcount) returns the number of rows that were actually changed by the last query. For details, see Update_rowcount Option in the "QUEL and EQUEL Statements chapter.
update_syscat	Returns "Y" if the effective user is allowed to update system catalogs, or "N" if the effective user is not allowed to update system catalogs.
username	Returns the session's current effective user ID.
_version	Returns the DBMS version number.

The following additional *request_names* are part of the Knowledge Management Extension:

Request Name	Description
group	Returns the session's group identifier or blanks if no group identifier is in effect.
role	Returns the session's role identifier or blanks if no role identifier is in effect.
query_io_limit	Returns the session's value for query_io_limit or -1 if no limit is defined for the session.
query_row_limit	Returns the session's value for query_row_limit or -1 if no limit is defined for the session.
create_table	Returns "Y" if the session has create_table privileges in the database or "N" if the session does not.
create_procedure	Returns "Y" if the session has create_procedure privileges in the database or "N" if the session does not.
db_admin	Returns "Y" if the session has the db_admin privilege.
lockmode	Returns "Y" if the session can issue the set lockmode statement or "N" if the session cannot.
maxio	Returns the value specified in the last set maxio statement. If no previous set maxio statement was issued or if set nomaxio was specified last, this returns the same value as the request name query_io_limit.
maxquery	Same as maxio.
maxrow	Returns the value specified in the last set maxrow statement. If no previous set maxrow statement was issued or if set nomaxrow was specified last, this returns the same value as the request name query_row_limit.
security_audit_log	Returns the name of the current security auditing log file. For details about security auditing, see the *SQL Reference Guide.*

Runtime Error Processing

By default, all EQUEL and DBMS server errors are returned to the EQUEL program, and messages are displayed on the standard output device. Using the iiseterr feature, you can define your own error-handling routine, which can display or suppress error messages. The iiseterr() function is not supported in all host languages. For more information, see the *Embedded QUEL Companion Guide*.

The program error handler must be declared in your program as an integer function, and declared as a parameter to the EQUEL routine iiseterr().

Avoid issuing any EQUEL statements within a user-written error handler, except for informative messages such as message, prompt, sleep and clear screen, and termination statements such as exit. If an error occurs in the error handler, there is the risk of infinite looping.

Retrieve Statement

In EQUEL, the retrieve statement returns data to a set of host language variables. In EQUEL programs, the retrieve statement is normally followed immediately by a block of program code enclosed by the delimiters "##{" and "##}". At runtime, the program retrieves a row into host variables and executes this block of code once for each row of data retrieved. If no rows are retrieved, the code block is not executed. The retrieve loop normally terminates after all rows have been processed.

You can terminate the loop before all rows are retrieved, using the endretrieve or endloop statements. You must not use a host language goto statement to exit the loop; if you do, the next database access statement causes an error.

Retrieve loops must not include other statements that access the database. When the retrieve loop terminates, control passes to the statement following the retrieve loop.

The following example illustrates the use of retrieve loops. This example retrieves a collection of rows, containing an employee's name, salary, and manager's name. For each row, the program statements in the retrieve loop compute and print the ratio of the employee's salary to the manager's.

The program processes at most 10 rows, and executes an endretrieve statement when the loop counter exceeds 10.

```
begin program
## ename            character_string(21)
## mname            character_string(21)
## salary, msalary  float
## eno, n           integer
## ingres personnel
   n = 0
## range of e is employee
## range of m is employee
## retrieve (ename = e.empname, salary = e.#salary,
##   mname = m.empname, msalary = m.#salary)
## where e.manager = m.empnum
## {
## n = n + 1
## if n > 10
##       endretrieve
   else
      print ename, salary, mname, msalary
   end if
## }
## exit
end program
```

The value from the salary column is automatically converted from money, as it is represented in the database, to floating point, as it is stored in the program variable.

The retrieve statement can be formulated as a *repeat query*, thus reducing the overhead required to run the same query repeatedly within an application. For more information, see Repeat Queries.

Using the Retrieve Statement Without a Loop

You can code a retrieve statement without an accompanying loop; in this case, one row, at most, is retrieved. This is appropriate, for instance, when your query seeks an exact match for a unique key. However, if more than one row qualifies according to the where clause, only one of the matching rows is returned.

Using the Sort Clause

The sort clause is used to sort result rows based on the contents of one or more columns. The names of result columns in the EQUEL retrieve statement are also names of program variables (the variables that receive the data from the retrieve). When coding the sort clause, you must typically dereference the sort column names.

For example:

```
begin program
## ename  character_string(26)
## eno    integer
## salary float
## ingres "personnel"
## range of e is emp
## retrieve (eno = e.empnum, ename = e.empname,
##  salary = e.#salary)
## sort by #eno
## {
     print eno, ename, salary
## }
## exit
end program
```

In this example, the sort column in the sort by clause must be dereferenced to sort on the "eno" column. If the column were not dereferenced, EQUEL assumes that the variable "eno" contained the name of the sort by column.

In the following example, the application prompts the user for the desired sort column; the user-specified sort key is read into the "sort_key" variable, which is used in the sort by clause. In this example, the variable must not be dereferenced: it is a variable and not a column name.

```
begin program
## ename    character_string(26)
## sort_key character_string(24)
## eno      integer
## salary   float
## ingres "personnel"
   print "Select sort column to use for employee list;"
   print "choices are eno, ename, or salary: "
   read sort_key from terminal
## range of e is emp
## retrieve (eno = e.empnum, ename = e.empname,
##   salary = e.#salary)
## sort by sort_key
## {
   print eno, ename, salary
## }
## exit
end program
```

Other Data Manipulation Statements

Unlike retrieve, other EQUEL database access statements do not have an inherent loop structure. The following example shows the use of the EQUEL append, replace, and delete statements.

```
begin program
## ename   character_string(21)
## salary  float
## eno     integer
   ename = "smith"
   salary = 15000
## ingres "personnel"
## range of e is employee
## append to employee (empname = ename,
###salary = salary)
   salary = 17500
## replace e (#salary = salary)
## where e.empname = ename
## delete e where e.#salary = salary
## exit
end program
```

As with the retrieve statement, the non-cursor versions of the delete, append, and replace statements can be formulated as repeated queries.

Repeat Queries

To reduce processing overhead for frequently executed queries, EQUEL allows you to specify retrieve, replace, append, or delete statements as "repeat queries." The first time a repeat query is executed, the DBMS Server retains the query execution plan (QEP). For subsequent executions of the repeat query, the retained QEP is used. For non-repeated queries, the DBMS Server must recreate the QEP every time the query executes. The first execution of a repeat query is slightly slower than an ordinary non-repeat query, because of the effort required to store the query plan. On subsequent executions, the query runs significantly faster than a non-repeat query.

The DBMS Server stores one QEP for each repeat query. To minimize the number of QEPs that must be managed, you must place code containing repeat queries in separate modules. When running applications containing repeat queries, each user has its own set of QEPs.

Variables containing values that can change from one pass to the next must be flagged by the "@" character. Any variable not marked as a parameter variable has its value fixed in the execution plan at the time the query is first executed. Typically, parameter variables occur in the where clause of queries, and the target list of append and replace statements. Result variables in the target list of a retrieve statement must not be flagged.

Flagged variables can substitute only for constants in the query. They must not contain qualifications (an entire "where clause") or the names of tables, range variables, or columns. The maximum number of flagged variables in one query is 127.

The following program illustrates the use of repeat queries:

```
begin program
## ename    character_string(26)
## salary   float
## eno      integer
   quit    character_string(10)
   responsecharacter_string(10)
   count   integer
## ingres personnel
## range of e is emp
   loop while quit = "no"
   print "enter an employee number: "
   read eno from terminal
   print "retrieving data ..."
   count = 0
## /* in the following query, eno is flagged */
##   repeat retrieve (ename = e.empname,
##   salary = e.#salary)
##   where e.empnum = @eno
##   {
     count = count + 1
     print ename, salary
##   endretrieve
##   }
if count 0 then
  print "delete that record? [yes or no]: "
  read response from terminal
  if response = "yes" then
##  repeat delete e where e.empnum = @eno
  end if
  else if count = 0 then
   print "no rows matched that employee number"
   print "adding employee number to table"
## repeat append to employee (empnum = @eno)
    end if
print "inquire about another employee? [yes or no]: "
read quit from terminal
  end loop
## exit
end program
```

Chapter 6: QUEL and EQUEL Statements

This chapter presents QUEL statements individually, describing each statement's purpose, syntax, and use.

QUEL Release

This chapter describes the release of QUEL indicated by the following values in the iidbcapabilities catalog:

CAP_CAPABILITY	CAP_VALUE
INGRES/QUEL_LEVEL	0850 (00605 for Ingres 2.0)

For more information about standard catalogs, see the *Database Administrator Guide.*

Statement Context

At the beginning of each statement description, you see the following table:

QUEL	EQUEL	KME
*	*	*

The columns in this table have the following meanings:

- An asterisk under QUEL indicates you can use the statement in an interactive session.
- An asterisk under EQUEL indicates that you can use the statement in embedded programs.
- An asterisk under KME indicates that the statement is part of the Knowledge Management Extension or has features that are part of the Knowledge Management Extension.

Ingres Forms Statements

This chapter does not describe Ingres Forms statements. For information about these statements, see the *Forms-based Application Development Tools User Guide.*

Abort

QUEL	EQUEL	KME
*	*	

Undoes some or all of the effects of a multi-query transaction (MQT).

Syntax

[##] **abort** [**to** *savepoint_name*]

Description

The abort statement reverses some or all of the updates performed by a multi-query transaction. If you do not specify a savepoint, abort undoes all the updates that were performed by the transaction, closes any open cursors, and terminates the transaction.

If you specify a savepoint, abort undoes all the updates that were performed between the savepoint *savepoint_name* and the abort statement. Open cursors are closed, but the entire transaction is not terminated (as shown in the second example, below).

For more information, see Savepoint.

Embedded Usage

You can specify savepoints using host string variables or integer literals.

Examples

The following examples provide details.

Example 1

The following examples show the use of abort to undo all the updates performed by the transaction.

```
## begin transaction
## append to emp(empname="jones,bill",
##   sal=100000, bdate=1814)
## append to emp(empname="jones,bill", sal=100000,
##   bdate=1714)
## abort  /* undoes both appends; table is unchanged */
```

Example 2

The following example shows the use of savepoints to undo the updates performed between savepoints "setone" and "settwo."

```
## begin transaction
## append to emp(empname="jones,bill", sal=10000,
##   bdate=1945)
## savepoint 1
## append to emp(empname="smith,stan", sal=50000,
##   bdate=1911)
## savepoint settwo
## abort to 1
## /*undoes 2nd append, deactivates savepoint settwo */
## append to emp(empname="smith,stan", sal=50000,
##   bdate=1948)
## abort to 1
## end transaction
/* only the first append is committed */
```

Append

QUEL	EQUEL	KME
*	*	

Adds a row to a database table.

Syntax

[##] [**repeat**] **append** [**to**] *tablename* (*columnname* = *expression*
{,*columnname* = *expression*}) [**where** *qual*]

Description

The append statement adds a row to the specified table. The columns of the row contain the values assigned in the *columnname* = *expression* clauses.

To reduce processing overhead for frequently repeated appends, specify the repeat option. Repeat directs the DBMS Server to save an execution plan after the append is first executed. In repeat append statements, you must flag variables if their values change (or can possibly change) each time the append is executed. If the variable appears on the right side of an equal sign (=), it must be preceded by an "at" sign (@). The @ flag tells the EQUEL preprocessor that the value of the variable must be checked each time the query is executed.

Embedded Usage

You can specify *tablename*, *columnname*, expressions in the target list or in the where clause, or the entire where clause, using host string variables.

Considerations

- Some host languages support the param version of append. See the *Embedded QUEL Companion Guide* for more information.
- The append statement fires any rules defined on the specified table that is fired by an equivalent SQL insert statement. Rules are part of the Knowledge Management Extension. For more information, see the *SQL Reference Guide*.

Examples

The following examples provide details.

Example 1

This example illustrates the use of the append statement to add a row to the "employee" table, based on values in variables "namevar" and "numvar".

```
## append to employee
##   (empname = namevar, sal = sal * 1.1, eno = numvar)
```

Example 2

This example illustrates the use of the append statement to add interviewees that tested above the minimum grade value to the "employee" table.

```
## range of i is interviewee
## append to employee (empname = i.name)
##      where i.evaluate >= minimum grade
```

Example 3

This example appends data from an array of 100 names into the "employee" table. Because the statement is issued many times, the repeat keyword is specified. This example assumes that "names" has been declared and initialized as an array of 100 character strings, and "i" has been declared as an integer.

```
i = 1
loop until i > 100   ##    repeat append to employee (empname = @names(i))
i = i + 1
end loop
```

Example 4

This example shows the use of a null indicator to assign null to the "age" column if the employee's age is not known.

```
loop while more rows in data set
    read name, salary, dept, age from data set
    if eage = 0 then
      nullind = -1
    else
      nullind = 0
##  append to employee
##    (empname = name, #salary = salary, edept = dept,
  ##  eage = age:nullind)
end loop
```

Begin Transaction

QUEL	EQUEL	KME
*	*	

Declares the beginning of a multi-query transaction (MQT).

Syntax

[##] **begin transaction**

Description

The begin transaction statement marks the beginning of a multi-query transaction (MQT). A begin transaction statement cannot be issued if any cursors are open. For information about transaction processing and cursors, see the "Embedded QUEL" chapter.

Example

This example shows a simple transaction that adds two rows to the table "emp."

```
## begin transaction
## append to emp(empname="jones,bill", sal=10000,
##   bdate=1914)
## append to emp(empname="smith,stan", sal=20000,
##   bdate=1948)
## end transaction  /* commits both appends to table */
```

Call

QUEL	EQUEL	KME
	*	

Calls an Ingres tool (such as RBF or Report-Writer) or the operating system.

Syntax

To call an Ingres tool:

call *subsystem* (**database** = *dbname* {, *parameter* = *value*})

To call the operating system:

call system (**command =** *command_string*)

Description

The call statement enables you to call an Ingres tool from within an embedded QUEL program. When calling an Ingres tool:

- *subsystem* must be the name of an Ingres tool.
- *dbname* must be the name of the current database. You cannot invoke the Ingres tool on a different database.

- *parameter* must be the name of a parameter accepted by the Ingres tool being called.
- *value* must be the value to be assigned to the parameter. If a particular parameter has no value, a null string (empty quotes) must be used.

Note: When your application calls an Ingres tool, the state of open transactions, open cursors, and repeat queries is not preserved. Each call to an Ingres tool must be considered as a separate DBMS server session.

When the user exits from the Ingres tool, control passes to the statement following the call. When used to call the operating system, the specified *command_string* is executed as if the user typed it at the operating system command line.

If *command_string* is a null, empty, or blank string, the statement transfers control to the operating system. The user can execute any operating system commands. Logging out returns the user to the application.

For more information about calling Ingres tools, see the "Calling Ingres Tools from Embedded QUEL" appendix.

Embedded Usage

Command_string, subsystem, *dbname*, and *parameter* must be specified using a (quoted or unquoted) string literal or host string variable. *Value* must be a quoted string or a string variable.

Examples

The following examples provide details.

Example 1

The following example runs a default report on the "employee" table in column mode.

```
## call report (database="personnel",
##    name="employee", mode="column")
```

Example 2

The following example runs QBF in the append mode with the QBFName "expenses," suppressing verbose messages.

```
## call qbf (database="personnel",
##  qbfname="expenses", flags="-mappend -s")
```

Close Cursor

QUEL	EQUEL	KME
	*	

Closes an open cursor.

Syntax

close cursor *cursor_name*

Description

The close cursor statement closes the specified cursor. Once closed, the cursor cannot be used for further processing unless reopened. An abort or end transaction statement implicitly closes all open cursors. *Cursor_name* must defined (using declare cursor) before it can be opened and closed.

Embedded Usage

You can specify *cursor_name* using a string constant or a host language variable.

Example

The following is an example of cursor processing.

```
begin program
## ename character_string ename(26)
## eno   integer
## ingres "personnel"
## range of e is employee
## declare cursor c1 for retrieve (e.empname, e.empnum)
##    where e.empnum 1000
## open cursor c1
loop until no more rows
##  retrieve cursor c1 (ename, eno)
    print ename, eno
 end loop
## close cursor c1
## exit
end program
```

Copy

QUEL	EQUEL	KME
*	*	

Copies data from a table into a file or from a file into a table.

Syntax

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

The *with-clause* consists of a comma-separated list of one or more of the following items:

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

The following options are valid only for bulk copy operations. For details about these settings, see Modify. The value specified for any of these options becomes the new setting for the table, and override any settings you have made previously (either using the modify statement or during a previous copy operation).

allocation = *n*
extend = *n*
fillfactor = *n* (isam, hash, and btree only)
minpages = *n* (hash only)
maxpages = *n* (hash only)
leaffill = *n* (btree only)
nonleaffill = *n* (btree only)

Description

The copy statement enables you to copy the contents of a table to a data file (copy into) or copy the contents of a file to a table (copy from). The following table briefly describes the valid statement parameters. Details about the parameters are provided in the following sections. For more information and procedures for using the copy statement, see the *Database Administrator Guide.*

Copy Statement Parameters

Parameter	Description
tablename	Specifies an existing table from which data is read or to which data is written.
columnname	Specifies the column from which data is read or to which data is written.
format	Specifies the format in which a value is stored in the file.
filename	Specifies the file from which data is read or to which data is written
filetype	Specifies the file type: text, binary, or variable. (Optional)

Binary Copying

To copy all rows of a table to a file using the order and format of the columns in the table, omit the column list from the copy statement. This operation is referred to as a *binary* copy. For example, to copy the entire "employee" table into the file "emp_name", issue the following statement:

```
copy table employee () into 'emp_name';
```

You must include the parentheses in the statement, even though no columns are listed. The resulting file contains data stored in proprietary binary formats. To load data from a file that was created by a binary copy (copy into) you must use a binary copy (copy from).

VMS

Bulk copy always creates a binary file.

Bulk Copying

To improve performance when loading data from a file into a table, you can use a *bulk copy*. The requirements for performing a bulk copy are:

- The table is not journaled
- The table has no secondary indexes

- The table is empty and occupies fewer than 18 pages if the table is other than heap
- The table is not partitioned

If the DBMS Server determines that all these requirements are met, the data is loading using bulk copy. If the requirements are not met, data is loaded using a less rapid technique. For detailed information about bulk copying, see the *Database Administrator Guide.*

Data File Format and Table Format

Table columns need not be the same data type or length as their corresponding entries in the data file. For example, numeric data from a table can be stored in char(0) or varchar(0) fields in a data file. The copy statement converts data types as necessary. When converting data types (except character to character), copy checks for overflow. When converting from character to character, copy pads character strings with blanks or nulls, or truncates strings from the right, as necessary.

When copying from a table to a file, you must specify the column names in the order you want the values to be written to the file. The order of the columns in the data file can be different from the order of columns in the table. When copying from a file to a table, you must sequence the table columns according to the order of fields in the data file.

Column Formats

The following sections describe how you specify the data file format for table columns. The format specifies how each is stored and delimited in the data file.

Storage Format

This section describes how you specify the format of fields in the data file. Be aware of the following points when specifying storage formats for copy into:

- Data from numeric columns, when written to text fields in the data file, is right-justified and filled with blanks on the left.
- If you are copying data from a floating-point table column to a text field in a data file, the data is formatted according to the options specified by the -i and -f flags. (For information about these flags, see the quel command description in the *Command Reference Guide.*)

- To avoid rounding of large floating point values, use the quel command -f flag to specify a floating point format that correctly accommodates the largest value you want to copy. (For information about this flag, see the quel command description in the *Command Reference Guide.*)

The following Copy Data File Storage Formats table explains the data file formats for the various QUEL data types. Delimiters are described in the section following the table:

Format	How Stored (Copy Into)	How Read (Copy From)
integer1, smallint, integer	Stored as integers of 1-, 2- or 4-byte length, respectively.	Read as integers of 1-, 2- or 4-byte length, respectively.
float4, float	Stored as floating point numbers (single or double precision, respectively).	Read as floating point numbers (single or double precision, respectively).
char(1),...,char(*n*)	Stored as fixed-length strings; *n* represents the lesser of the maximum configured row size and 32,000.	Read as fixed-length string
char(0)	Stored as fixed-length strings (padded with blanks if necessary). For character data, the length of the string written to the file is the same as the column length.	Read as variable-length character string terminated by the first comma, tab, or newline encountered.
char(0)*delim*	Stored padded to the column's declared width. The one-character delimiter is inserted immediately after the value. Because this format uses spaces to pad data, a space (sp) is not a valid delimiter for this format.	Read as variable-length character string terminated by the specified character.
varchar(1),...,varchar(*n*)	Stored as fixed-length strings preceded by a 5-character, right-justified length specifier. If necessary, the value is padded with null characters to the specified length; *n* represents the lesser of the maximum configured row size and 32,000.	Read as fixed-length string, preceded by a 5-character, right-justified length specifier.
varchar(0)	Stored as a variable-length string preceded by a 5-character, right-justified length specifier.	Read as variable-length string, preceded by a 5-character, right-justified length specifier.
d0	(Not applicable)	Dummy field: read as a variable-length character string terminated by the first comma, tab, or newline encountered. The data in the field is skipped.

Format	How Stored (Copy Into)	How Read (Copy From)
d*n*	Dummy column: instead of placing a value in the file, copy inserts the name of the column *n* times. For example, if you specify x=d1, the column name "x" is inserted once, if you specify x=d2, copy inserts the column name "x" twice, and so on. You can specify a delimiter as a column name, for example, nl=d1.	Dummy field: read as a variable-length character string of the specified length. The data in the field is skipped.
d0*delim*	Indicates a delimited dummy column. Instead of placing a value in the file, copy inserts the specified *delim.* (Unlike the dn format, does not insert the column name.)	Dummy field: read as a variable-length character string delimited by the specified character. The data in the field is skipped.
date	Stored in date format.	Read as a date field.
money	Stored in money format.	Read as a money field.

Note: The dummy format (d*n*) behaves differently for copy from and copy into. When you copy a table into a file, *n* specifies the number of times the column name is repeated. When you copy from a file to a table, *n* specifies the number of bytes to skip.

For user-defined data types (UDTs), use char or varchar. For details about UDTs, see the *Object Management Extension User Guide.*

Delimiters

Delimiters are those characters in the data file that separate fields and mark the end of records. Valid delimiters are listed in the following Data File Delimiters table:

Delimiter	Description
nl	newline character
tab	tab character
sp	space
nul or null	null character
comma	comma
colon	colon
dash	dash

Delimiter	Description
lparen	left parenthesis
rparen	right parenthesis
x	any non-numeric character

When you specify a single character as the delimiter, you must enclose that character in quotes. If the data type specification is d0, the quotes must enclose the entire format. For example, d0% specifies a dummy column delimited by a percent sign (%).

If the data type specification is char(0) or varchar(0), only the delimiter character must be quoted. For example, char(0)"%" specifies a char field delimited by a percent sign.

Do not use the space delimiter (sp) with char(0) fields: the char(0) format uses spaces as padding for character and numeric columns.

When copying from a table into a file, you can insert delimiters independently of columns. For example, to insert a newline character at the end of a line, specify nl=d1 at the end of the column list. This directs the DBMS Server to add one (d1) newline (nl) character. (Do not confuse lowercase "l" with the number "1".)

With Null Clause

When copying data from a table to a file, the with null clause directs copy to put the specified value in the file in place of null fields. You must specify the with null clause for any column that is nullable; if you omit the with null clause, the DBMS Server returns an error when it encounters null data, and aborts the copy statement.

When copying data from a file to a table, the with null clause specifies the value in the file to be interpreted as a null. When copy encounters this value in the file, it writes a null to the corresponding table column.

To prevent conflicts between valid data and null entries, choose a value that does not occur as part of the data in your table. The value you choose to represent nulls must be compatible with the format of the field in the file. Character formats require quoted values and numeric formats require unquoted numeric values.

For example, the following example is incorrect, because the value specified for nulls (numeric zero) conflicts with the character data type of the field:

Wrong:

```
c0comma with null(0)
```

The following example, however, is correct:

Right:

```
c0comma with null("0")
```

The null value is character data, specified in quotes, and does not conflict with the data type of the field. Do not use the keyword null, quoted or unquoted, for a numeric format.

When copying from a table to a file, be sure that the field format you specify is at least as large as the value you specify for the with null clause. If the column format is too small, the DBMS Server truncates the null value written to the data file to fit the specified format. For example, in the following statement the string "NULL" is truncated to "N" because the format is incorrectly specified as one character:

Wrong:

```
copy table t1 (col1 = varchar(1) with null ("NULL")) into "t1.dat"
```

The correct version specifies a 4-character format for the column.

Right:

```
copy table t1 (col1 = varchar(4) with null ("NULL")) into "t1.dat"
```

If you specify with null but omit *value*, copy appends a trailing byte indicating whether the field is null. For null fields, copy inserts an undefined data value in place of the null and sets the trailing byte to indicate a null field. You must specify *value* for nullable char(0) and varchar(0) columns.

Filename Specification

Filename must be enclosed in single quotation marks; the file specification can include a directory/path name. For copy into, if the file does not exist, copy creates the file.

UNIX

For copy into, if the file already exists, copy overwrites it.

VMS

For copy into, if the file already exists, copy creates another version of the file.

VMS File Types

You can specify file type using the optional *type* parameter. *Type* must be one of the values listed in the following VMS File Types table:

Type	Record Format	Record Attributes
text	Variable length	Records delimited by carriage return
binary	Fixed length	None
variable	Variable length	None

If you omit *type*, **copy** determines the file type as follows:

- If all fields in the file are character types (char, varchar), and all records end in <newline>, copy creates a text file.
- If the file contains variable length records, its file type is variable. Variable length records occur if one or more fields are stored as varchar(0).
- If none of the preceding conditions apply, copy creates a binary file.

If you specify *type*, the contents of the file must be in accordance with these rules. If it is not, copy creates the data file according to the preceding rules.

With Clause Options

The following sections describe the valid with clause options.

On_error Option

To direct copy to continue after encountering conversion errors, specify the on_error option. To direct copy to continue until a specified number of conversion errors have occurred, specify the error_count option (instead of on_error). By default, copy terminates when an error occurs while converting a table row into file format.

When on_error is set to continue, copy displays a warning whenever a conversion error occurs, skips the row that caused the error, and continues processing the remaining rows. At the end of the processing, copy displays a message that indicates how many warnings were issued and how many rows were successfully copied.

Setting on_error to continue does not affect how copy responds to errors other than conversion errors. Any other error, such as an error writing the file, terminates the copy operation.

Error_count Option

To specify how many errors can occur before processing terminates, use the error_count option. The default error_count is 1. If on_error is set to continue, setting error_count has no effect.

Log Option

To store any rows that copy cannot process to a file, specify the with log option. With log can only be used if you specify on_error continue. When you specify with log, copy places any rows that it cannot process into the specified log file. The rows in the log file are in the same format as the rows in the database.

Logging works as follows:

Windows **UNIX**

Copy opens the log file prior to the start of data transfer. If it cannot open the log file, copy halts. If an error occurs writing to the log file, copy issues a warning, but continues. If the specified log file already exists, it is overwritten with the new values (or truncated if the copy operation encounters no bad rows). 

VMS

Copy attempts to open the log file prior to the start of data transfer. If it cannot open the log file, copy halts. If an error occurs writing to the log file, copy issues a warning, but continues. If the log file already exists, copy creates a new version. 

If you are copying from a data file that contains duplicate rows (or rows that duplicate rows that are already in the table) to a table that has a unique key, copy displays a warning message and does not add the duplicate rows. Note that, if you specified the with log option, copy does not write the duplicate rows to the log file.

With Rollback Option

To direct the DBMS Server to back out all rows appended by the copy if the copy is terminated due to an error, specify with rollback=enabled. To retain the appended rows, specify with rollback=disabled. The default is with rollback=enabled. When copying to a file, the with rollback clause has no effect.

The rollback=disabled option does not mean that a transaction cannot be rolled back. Database server errors that indicate data corruption still causes rollback, and rows are committed until the transaction is complete.

With Row_estimate Option

To specify the estimated number of rows to be copies from a file to a table, use the with row_estimate option. The DBMS Server uses the specified value to allocate memory for sorting rows before inserting them into the table. An accurate estimate can enhance the performance of the copy operation.

The estimated number of rows must be no less than 0 and no greater than 2,147,483,647. If you omit this parameter, the default value is 0, in which case the DBMS Server makes its own estimates for disk and memory requirements.

Permissions

To copy from a table into a file or from a file to a table, one of the following must apply:

- You own the table, or
- The table has select (for copy into) or insert (for copy from) permission granted to public, or
- The current session is running with a user, role, or group identifier that has been granted select (copy into) or insert (copy from) privilege on the table.

Locking

When you copy from a table into a file, the DBMS Server takes a shared lock on the table. When you copy into a table, the DBMS Server takes an exclusive lock on the table.

Restrictions and Considerations

- You cannot use copy to add data to a view, index, or system catalog.
- When copying data into a table, copy ignores any integrity constraints (defined using the define integrity statement) defined against the table.
- When copying data into a table, copy ignores ANSI/ISO Entry SQL-92 check and referential constraints, but does not ignore unique (and primary key) constraints. For details about ANSI/ISO table constraints, see the *SQL Reference Guide.*
- The copy statement does not fire any rules defined against the table.

Examples

The following examples illustrate various features of the copy statement.

Data File Format

Example 1

In the following example, the contents of the file "emp.txt" are copied into the "employee" table. A dummy column is used to omit the "city" column. The format of the" employee" table is as follows:

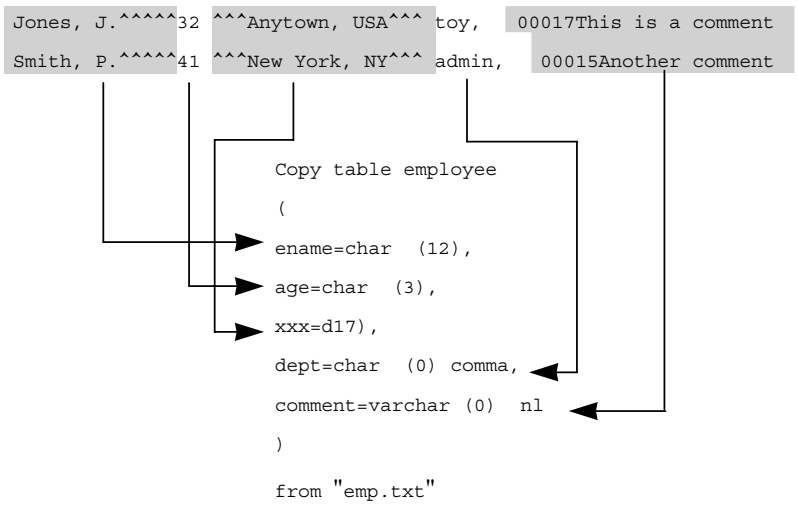
```
ename     char(15)
age       integer4
dept      char(10)
comment   varchar(20)
```

The "emp.txt" file contains the following data:

```
Jones,J.   32   Anytown,USA   toy,00017A comment
Smith,P. 41   New York,NY   admin,00015Another comment
```

The following diagram illustrates the copy statement that copies the file "emp.txt" into the employee table and maps the fields in the file to the portions of the statement that specify how the field is to be copied. Note the following points:

- A dummy column is used to skip the city and state field in the data file, because there is no matching column in the employee table.
- The "department" field is delimited by a comma.
- The "comment" field is a variable-length varchar field, preceded by a five-character length specifier.



Example 2:

Load the "employee" table from a data file. The data file contains binary data (as opposed to character data that can be changed using a text editor):

```
copy table employee (eno=integer2, ename=char(10),
     age=integer2, job=integer2, sal=float4,
     dept=integer2, xxx=d1)
     from "myfile.in"
```

Example 3:

Copy data from the "employee" table into a file. The example copies employee names, employee numbers, and salaries into a file, inserting commas and newline characters so that the file can be printed or edited. All items are stored as character data. The "sal" column is converted from its table format (money) to ASCII characters in the data file.

```
copy table employee (ename=char(0)tab,
     eno=char(0)tab, sal= char(0)nl)
     into "myfile.out"
```

```
Joe Smith       ,   101,    $25000.00
Shirley Scott   ,   102,    $30000.00
```

Example 4:

Bulk copy the "employee" table into a file. Resulting data file contains binary data:

```
copy table employee () into "ourfile.dat"
```

Example 5:

Bulk copy from the file created in the preceding example:

```
copy table other_employee_table () from "ourfile.dat"
```

Example 6:

Copy the "acct_recv" table into a file. The following statement skips the address column, uses the percent sign (%) as a field delimiter, uses "xx" to indicate null "debit" and "credit" fields, and inserts a newline at the end of each record:

```
copy table acct_recv
  (acct_name=char(0)"%",
   address="d0%",
   credit=char(0)"%" with null("xx"),
   debit=char(0)"%" with null("xx"),
   acct_mngr=char(15),
   nl=d1)
   into "qtr_result";
```

Example 7:

Copy a table called "gifts" to a file for archiving. This table contains a record of all non-monetary gifts received by a charity foundation. The columns in the table contain the name of the item, when it was received, and who sent it. Because givers are often anonymous, the column representing the sender is nullable.

```
copy table gifts
    (item_name=char(0)tab,
    date_recd=char(0)tab,
    sender=char(20)nl with null("anonymous"))
    into "giftdata";
```

Create

QUEL	EQUEL	KME
*	*	

Creates a new database table.

Syntax

[##] **create** [*locationname:*]*tablename*
(*columnname=format* [*null_clause*]
{**,** *columnname=format*} [*null_clause*]**)**
[**with** [**location =** *locationname*]
[**no**]**journaling**
[**no**]**duplicates**]]

Description

The create statement creates an empty table, owned by the user issuing the statement. The table is created as a heap. To change to a different storage structure, use the modify statement. The following table describes the create statement parameters:

Parameter	Description
tablename	Specifies the name of the table. Table names must not begin with "ii".
columnname	Specifies the name of each column in the new table. The column name must be a valid object name.
format	Specifies the data type, length, and null characteristics of each column. *Format* has the syntax *datatype* [not null [with | not default] | with null]

The optional with clause consists of a comma-separated list of any of the following parameters:

- location = locationname
- [no]journaling
- [no]duplicates

The optional *null_clause* enables you to set the location, journaling, and duplicate row characteristics of the table. The following table summarizes the possible null and default settings of the *with_clause* and the resulting column attributes (the default is not null with default):

Null/Default Specification	Nulls Allowed?	Defaults Allowed?
None	No	Yes
with null	Yes	Yes (null is default)
not null	No	Yes
not null with default	No	Yes
not null not default (mandatory column)	No	No

A table can have a maximum of 1024 columns. A row can have a maximum of 2008 bytes. A text or varchar column uses two bytes in addition to the specified length (to store the string length). Nullable columns (columns defined with null) use an additional byte for a null flag. In tables having a compressed format (chash, cbtree, cheap, or cisam), c columns require 1 byte in addition to the declared length, and char columns require 2 additional bytes. These space requirements must be considered as part of the maximum allowable 2008 bytes per row.

In the optional with clause*, locationname* refers to the areas where the new table is created. The locations must be defined on the system, and the database must have been extended to the corresponding areas. If you do not specify a location, the default area for the database is assumed. If you specify multiple locations, the table is physically partitioned across the areas. For more information, see the *Database Administrator Guide*.

If you specify with journaling, all append, replace and delete statements that update this table are logged in the journal for this database, if journaling for the database is enabled. (To enable database journaling, use the ckpdb command.) Journaling allows the recovery system to reconstruct the table after a disk crash. You need not enable journaling to recover from operating system or server failures because this is handled by normal query processing. Journaling also allows an audit trail to be built for the table. You can use this audit trail to monitor updates to a table or maintain change histories.

If you specify with duplicates, the table accepts duplicate rows even if the table is subsequently modified to a storage structure which does not ordinarily permit duplicate rows. The default is with noduplicates. The duplicates|noduplicates parameter is irrelevant when the table is a heap. For more details, see Modify.

Embedded Usage

You can use unquoted strings or host string variables to specify *locationname*, *tablename*, *columnname*, *format*, and the complete target list (the list of column names and format descriptions.)

The EQUEL preprocessor does not validate the syntax of the *with_clause*.

Considerations

- If you are creating a table in a distributed database, the syntax of create is different. For a full description of creating tables in a distributed database, see the *Database Administrator Guide*.
- Tables are created with no expiration date. To set an expiration date for a table, use the save statement.

Examples

The following examples provide details.

Example 1

The following example shows the use of the create statement to create the "employee" table:

```
## create employee
## (eno = i2,
## ename = c20,
## age = i1,
## job = i2,
## sal = money with null,
## dept = i2)
## with journaling
```

Example 2

The following example illustrates the use of a host variable with the create statement to create a table whose definition is decided at runtime:

```
tablevar = "mytable"
descvar = "name = c20, phone = c11"
## create tablevar (descvar)
```

Declare Cursor

QUEL	EQUEL	KME
	*	

Declares a cursor.

Syntax

```
## declare cursor cursor_name
##         for retrieve [unique] (target_list)
##         [where qual]
##         [sort by column[:sortorder] {, column[:sortorder]}]
##         [for [deferred | direct] update [of (column {, column})]]
```

Description

The declare cursor statement associates a cursor name with a set of retrieval criteria. The declare cursor statement must occur before any other references to the cursor–declare cursor statements cannot be embedded in a host language variable declaration section.

If the cursor is used to update or delete rows, you must specify the for update clause and include all columns that are updated.

If the sort by clause is specified, the cursor retrieves rows sorted by result column, as specified. For each column on which you are sorting, you can specify the *sortorder* parameter as asc for ascending, or desc for descending.

The declare cursor statement does not retrieve any data (despite the presence of the retrieve clause). Data is retrieved into host variables when you open the cursor and issue a retrieve cursor statement.

Embedded Usage

Cursor_name can be either a constant or a host language variable. The maximum length of a cursor name is 32 characters.

Considerations

- The same cursor name cannot be declared twice in a single program.
- The scope of a cursor is the source file. A cursor name declared in one source file cannot be referred to in another file.
- The EQUEL preprocessor does not generate any code for the declare cursor statement. If your host language does not allow empty control blocks, (for example, empty if blocks), the declare cursor statement must not be the only statement in the block.
- Result columns which have the same name as a host variable must be dereferenced with the number (#) sign.
- The retrieve clause for the declare cursor statement must obey the rules for the retrieve statement, with an additional restriction. A cursor retrieve clause does not allow you to store the results of the query in host variables. The target list assignments, when used, are result column names, not receiving host variables.
- A cursor cannot be declared for update if its retrieve statement refers to more than one table or to a view, or includes a unique or a sort by clause.
- You can use a host variable to specify the where clause and succeeding clauses (such as sort by or update). You can use host variables to specify table or column names.

Examples

The following examples provide details.

Example 1

The following example declares a cursor for retrieval of employees from the shoe department, ordered by name (ascending) and salary (descending):

```
## declare cursor cursor1 for
## retrieve (employee.empname, employee.salary)
## where employee.dept = "shoes"
## sort by #ename:asc, sal:desc
```

Example 2

The following example declares a cursor for updating the salaries and departments of shoe department employees:

```
## declare cursor cursor2 for
## retrieve (employee.empname, employee.salary)
## where employee.dept = "shoes"
## for update of (salary, dept)
```

Example 3

The following example declares a cursor for retrieval and update of employee information:

```
begin program
## eno      integer
## age      integer
## thatsall  integer
## ename     character_string(26)
## salary      float
## newsalary  float
## ingres "personnel"

## declare cursor cursor4 for
##  retrieve (employee.empname, employee.#age,
##  employee.empnum, employee.#salary)
## for direct update of (#salary)
## open cursor cursor4
loop while no errors and endquery not reached
##  retrieve cursor cursor4 (ename, age, eno, salary)
##  inquire_ingres (thatsall = endquery)
    if thatsall = 0 then
      print ename, age, eno, salary
      print "enter new salary: "
      read newsalary from terminal
      if newsalary  > 0 and newsalary != salary then
  ##      replace cursor cursor4 (#salary = newsalary)
        end if
      end if
end loop
```

```
##close cursor cursor4
##exit
end program
```

Example 4

In the following example, the "for update" clause refers to the column named "salary" and not result column "res". The variables "eno_low" and "eno_high" must have previously been declared:

```
## declare cursor cursor5 for
##  retrieve (employee.empname, res = employee.salary)
##    where employee.empnum >= eno_low and
##     employee.empnum <= eno_high
##    for update of (#salary)
loop while more input
    read eno_low, eno_high
##  open cursor cursor5
      print and process rows
##  close cursor cursor5
end loop
```

Example 5

The following example declares two cursors for the "department" and "employee" tables and opens them in master-detail fashion:

```
## declare cursor master_cursor for
##  retrieve (dept.all)
##  sort by dno

## declare cursor detail_cursor for
##  where employee.edept = dept.dno
##  sort by empname

## begin transaction

## open cursor master_cursor
loop while more departments
## retrieve cursor master_cursor
## inquire_equel (thatsall = endquery)

if thatsall = 0 then
## /* for each department retrieve all the employees
## and display the department and employee data.*/
##  open cursor detail_cursor
loop while more employees
##  retrieve cursor detail_cursor
##  (name, age, idno, salary,   edept)
## /* for each department retrieve all the employees
##  and display the department and employee data.*/

##   inquire_equel (thatsall = endquery)
     if thatsall = 0 then
       process and display data
     end if
      end loop
## close cursor detail_cursor
   end loop
## end transaction
```

Example 6

The following example declares a cursor that is a union of three tables with identically typed columns (the columns have different names). The expression "one + 1" in the target list of the declare cursor must be assigned to a result column. This is not the case with the other target list items, which are verbatim table columns. The name "two" must be dereferenced (with a # sign), because there is a host variable of the same name. The host variable is used later, in the retrieve cursor statement, to receive the value of the result column. The sort key names mentioned in the sort by clause are result column names, and must be dereferenced if they have the same names as any host variables.

```
begin program
## age      integer
## thatsall integer
## one      integer
## two      integer
## minage    integer
## ename     character_string(26)
## salary    float

## ingres "personnel"

## declare cursor cursor7 for
##  retrieve (#two = one + 1, employee.empname,
##   employee.#age)
##  where employee.age     minage
##  sort by empname, #age

one = 1
minage = 21

## open cursor cursor7
loop while no errors and endquery not reached
##  retrieve cursor cursor7 (two, ename, age)
##  inquire_ingres (thatsall = endquery)
    if thatsall = 0 then
      print two, ename, age
    end if
 end loop

##  close cursor cursor7
##  exit
end program
```

Define Integrity

QUEL	EQUEL	KME
*	*	

Defines integrity constraints.

Syntax

[##] **define integrity** [**on**] *range_variable* [**is**] *qual*

Description

The define integrity statement creates an integrity constraint for the specified base table. Only the owner of a table is allowed to define integrities on the table. The integrity constraint you specify in the *qual* parameter must be true for the table at the time the define integrity statement is issued. If not, the DBMS Server displays an error message and does not create the integrity. If the constraint includes one or more columns that contain nulls, you must specify or is null in the *qual* parameter.

While executing, the define integrity statement takes out an exclusive lock on the table.

Embedded Usage

You can use host string variables to specify *range_variable*, *qual*, or the table names, column names and constant expressions that constitute *qual*.

Examples

The following examples provide details.

Example 1

The following example makes sure that all employees salaries are greater than or equal to $6000:

```
## range of e is employee
## define integrity on e is e.salary >= 6000
```

Example 2

The following example defines an integrity using a variable:

```
## define integrity on e is e.salary >= salvar
```

Define Permit

QUEL	EQUEL	KME
*	*	

Adds permissions to a table.

Syntax

[##] **define permit** *oplist* **on** | **of** | **to** *range_var*
[**(***columnname* {**,** *columnname*}**)**] **to** *user_name* | **all** [**at** *term*]
[**from** *time* **to** *time*] [**on** *day1* **to** *day2*] [**where** *qual*]

Description

The define permit statement adds permissions to the table specified by *range_var*. The following table lists the define permit statement parameters:

Parameter	Description
oplist	A comma-separated list of any of the following operations: retrieve, replace, delete, append, or all.
user_name	The login name of a user or the word all (meaning all users).
term	Must be one of the following: a two-character generic device name, such as *tt, rt, tx or op,* a three-character device name, such as *tta* or *ttb*, or a four-character terminal identifier, such as *tta1* or *ttb4*. All terminal names that match the specified term names are given the permissions. Omitting this phrase is equivalent to specifying all.
time	Must be specified in *hh:mm* format, using the twenty-four hour clock. Time specifies the times of the day during which this permission applies. At other times, the permission is not granted.

Parameter	Description
days	Must be three-character abbreviations for days of the week (mon, tue, wed, thu, fri, sat, sun).

The DBMS Server appends the where clause to the specified type of query (append, retrieve, replace, or delete) when the query is executed by the specified user. To append, replace, and delete columns using a where clause, a user must have retrieve permission for the columns. Do not specify column names in a define permit for the delete statement (because you delete rows, not columns).

When you define permissions, the DBMS Server "ands" the separate parts of a single define permit statement and "ors" separate define permit statements. For example, if you issue the following define permit statement:

```
define permit replace on e to eric at tta4 [...]
```

the permit applies only to "eric" when logged in on "tta4", but if you issue two define permit statements:

```
define permit replace of e to eric at tta4 [...]
define permit retrieve of e to all at all [...]
```

When "eric" logs in at "tta4", his login is affected by the union of the permissions specified by the two statements. That is, "eric" can both retrieve and update data from the "employee" table. If "eric" logs in at "ttb2", he is granted only the permissions specified in the second define permit statement: he can only retrieve rows from the employee table. If another user logs in on "tta4" or any other terminal, he or she is granted only the permissions specified in the second define permit statement.

You must be the DBA to issue the define permit statement. The database administrator (DBA) is typically responsible for maintaining database security using permissions. Permissions cannot be granted to users on a table that is not owned by the DBA.

Permissions cannot be defined on views, although the DBMS Server honors permissions defined on the base tables on which the view is based.

Example

In the following example, define permit is used to enable "ariane" to retrieve names, ages, and salaries of employees whom she manages from the "employee" table at terminal "tta2" between 8:00 am and 5:00 pm, Monday through Friday:

```
range of d is dept
range of e is employee
define permit retrieve of e (ename, age, salary)
    to ariane at "tta2" from 8:00 to 17:00 on mon to fri
    where e.dept=d.dno and d.mgr="ariane*"
```

Define View

QUEL	EQUEL	KME
*	*	

Defines a virtual table.

Syntax

[##] **define view** *view_name* **(***target_list***)** [**where** *qual*]

Description

The define view statement defines a view. A view is a virtual table: the view definition is stored, but define view does not create any new base tables. The syntax of the define view statement is similar to the retrieve statement. When the view is used to form queries, the DBMS Server interprets the query to retrieve data from the base tables that define the view.

Considerations

- In general, no updates are supported on views derived from more than one base table.
- You cannot update columns that are in the where clause of the view definition.
- You can only update simple columns from the *target_list* of a view definition; you cannot update columns that are not simple columns (such as aggregates, derived columns, or constants).
- You cannot update a row with a value that causes the row to be dropped from the view.
- You can only define views based on tables for which you have retrieve permission.
- When you destroy a table that is referenced by a view, the DBMS Server automatically destroys the view.

Example

The following example defines a view of employee data that includes name, salary and manager's name:

```
## range of e is employee
## range of d is dept
## define view empdpt
##  (ename = e.name, e.sal, dname = d.name)
##  where e.mgr = d.mgr
```

Delete

QUEL	EQUEL	KME
*	*	

Deletes rows from a database table.

Syntax

[##] [**repeat**] **delete** *range_variable* | *tablename* [**where** *qual*]

Description

Delete removes rows that satisfy the where clause from the table to which *range_variable* refers. If no where clause is specified, all rows are deleted.

To reduce the overhead required to execute a frequently repeated delete, specify repeat delete. Repeat directs the DBMS Server to encode the delete and save its execution plan. Program variables that change each time the query is executed and that appear on the right-hand side of an equal sign (=) must be preceded by the @ sign.

The delete statement fires any rules that is fired by an equivalent SQL delete statement. Rules are part of the Knowledge Management Extension. For more information, see the *SQL Reference Guide*.

Embedded Usage

You can use host variables to specify *range_variables*, table names, column names and expressions. Host variables that correspond to expressions (in the where clause) can include null indicator variables. You can use a host string variable to specify the where clause.

Considerations

- To delete rows, you must own the table or have delete permission.
- Do not mix range variables with table names in a delete statement: the resulting disjoint query gives unexpected results. The following example illustrates a disjoint query; the table name following the keyword delete is not the same as the range variable specified in the where clause. All rows are deleted as a result of this disjoint query.

  Wrong:

  ```
  range of e is employee
  delete employee where e.salary  >  35000
  ```

- After deleting a large number of rows from a table, you can use the modify statement to recover empty space. To delete all rows in a table, you can use the modify *tablename* to truncated. For more information, see Modify.

Examples

The following examples provide details.

Example 1

The following example deletes the row in the "employee" table corresponding to the employee number specified by host variable *numvar*:

```
delete employee where employee.empnum = numvar
```

Example 2

The following example deletes the row in the "employee" table whose name corresponds to the specified by host variable *namevar*. Notice the use of repeat, and the use of @ to flag a program variable that changes with each execution of the delete statement:

```
range of e is employee
repeat delete e
    where e.empname = @namevar
```

Example 3

The following embedded example shows the use of delete in a loop; the loop reads entries from an array of employee IDs and deletes the corresponding row from the database:

```
i = 1
 loop until (numbers(i)=end of list)
##  repeat delete employee
##   where employee.empnum = @numbers(i)
##  i = i + 1
 end loop
```

Example 4

The following embedded example shows the use of delete in conjunction with a host string variable containing search criteria:

```
construct search_condition
## delete employee
##  where search_condition
```

Example 5

In the following embedded example, employees whose salary is null are on a leave of absence. If they were hired after a certain date (supplied by the program), they are deleted.

```
## delete employee
##  where employee.salary is null
##  and employee.hire_date > base_date
```

Delete Cursor

QUEL	EQUEL	KME
	*	

Deletes a row from a database table.

Syntax

delete cursor *cursor_name*

Description

The delete cursor statement deletes the row to which the cursor currently points. If the cursor is not pointing to a row (for example, the cursor has reached the end of the set or is positioned after a row due to a previous delete), the DBMS Server issues a runtime error.

After the row is deleted, the cursor points to a position after the deleted row, but before the next row (if any). To advance the cursor, the application program must issue a retrieve cursor statement.

If the cursor is opened for direct update, the deletion occurs immediately. If the cursor is opened for deferred update (the default), the deletion occurs when the cursor is closed.

Embedded Usage

You can specify *cursor_name* using a string constant or a host language variable.

Considerations

- The end transaction and abort statements close all open cursors. You cannot, for example, delete a row, commit the delete (by issuing an end transaction statement), and retrieve the next row, because the end transaction statement closed the cursor.
- *Cursor_name* must be an open cursor.

Example

The following example deletes the row in the "employee" table to which the cursor is pointing:

```
## declare cursor cursor1 for
## retrieve (employee.empname, employee.empnum)

## open cursor1
 loop until no more rows

## retrieve cursor cursor1 (name, idno)

if idno < 1000 then
    print "deleting " name
## delete cursor cursor1
 end if

end loop
```

Destroy

QUEL	EQUEL	KME
*	*	

Destroys existing tables, views, permissions, or integrities.

Syntax

[##] **destroy** *tablename* {**,** *tablename*}

[##] **destroy permit | integrity** *tablename integer* {, *integer*} | **all**

Description

Destroy removes tables from the database, or integrity constraints and permissions from a table or view. Only the owner is allowed to destroy a table or its permissions and integrity constraints. Destroying a table destroys all views built on that table.

If the table being destroyed has secondary indexes, the secondary indexes are also destroyed. You can destroy a secondary index separately, without affecting the base table.

To destroy individual permissions or constraints for a table, you must use the *integer* argument. Use the help permit statement (for permissions) or a help integrity statement (for constraints) to display the argument values for the various individual permissions and constraints. To destroy all constraints or permissions, specify all.

Destroy accepts a maximum of 30 arguments. To destroy more than 30 objects, you must use multiple destroy statements.

Embedded Usage

You can use host string variables to specify *tablename*, *viewname*, and the integer arguments.

Examples

The following examples provide details.

Example 1

The following example destroys the "employee" and "dept" tables:

```
destroy employee, dept
```

Example 2

The following example destroys specific permissions on the "job" table, and all integrity constraints on the "employee" table:

```
destroy permit job 2, 4, 5
destroy integrity employee all
```

Endretrieve

QUEL	EQUEL	KME
	*	

Terminates a **retrieve** loop.

Syntax

endretrieve

Description

The endretrieve statement terminates EQUEL retrieve loops. A retrieve loop is a retrieve statement followed by a block of code delimited by curly braces ({ }). The endretrieve statement is used within the code block to terminate the retrieve loop and transfer control to the first statement following the loop.

If the endretrieve statement is issued inside a forms display loop that is nested within a retrieve loop, the endretrieve statement terminates the display loop as well as the retrieve loop.

Example

The following example illustrates the use of the endretrieve statement to break out of a retrieve loop in the event of an error:

```
## retrieve (ename = employee.empname,
    eno = employee.empnum)
## {
 load ename, eno into data set
 if error then print "error while loading!"
##  endretrieve
 end if
## }
## /* endretrieve transfers control to here */
```

End Transaction

QUEL	EQUEL	KME
*	*	

Terminates a multi-query transaction (MQT) and commits updates to the database.

Syntax

[##] **end transaction**

Description

The end transaction statement terminates a successful multi-query transaction (MQT) and commits its updates to the database. When the updates are committed, the effects (on the database) become visible to other users. For more information, see Begin Transaction in this chapter and Transactions in the "Embedded QUEL" chapter.

Considerations

The end transaction statement closes all open cursors.

Example

The following example shows a simple MQT: two append statements framed by begin transaction and end transaction statements:

```
##  begin transaction
##  append to employee(empname="jones,bill",
##   sal=10000, bdate=1914)
##  append to employee(empname="smith,stan",
##   sal=20000, bdate=1948)
##  end transaction
## /* commits new rows to table */
```

Exit

QUEL	EQUEL	KME
	*	

Terminates access to the database.

Syntax

exit

Description

The exit statement terminates access to a database. The application program must have previously connected to the database using the ingres statement. The exit statement is equivalent to the Terminal Monitor q command. After access is terminated with the exit statement, your application can issue another ingres statement to connect to the same or a different database. An EQUEL program can access only one database at a time.

Considerations

- If the exit statement is issued during a multi-query transaction, all updates performed because the previous begin transaction is aborted.
- The exit statement closes all open cursors.

Help

QUEL	EQUEL	KME
*		

Gets information about a variety of database objects.

Syntax

help [*objectname* {, *objectname*}]
help comment column *table columnname* {, *columnname*}
help comment table *table* {, *table* }
help constraint *constraintname*
{, *constraintname*}
help default *tablename*
help help
help index *indexname* {, *indexname*}
help integrity *tablename* {, *tablename*}
help permit on procedure | **table** | **view**
objectname {**,** *objectname*}
help procedure *procedure_name*
{, *procedure_name*}
help register *objectname*
help synonym *synonym* {, *synonym*}
help table *tablename* {, *tablename*}
help view *viewname* {, *viewname*}

The following help statements are part of the Knowledge Management Extension:

help rule *rulename,* {*rulename*}
help permit on dbevent
objectname {**,** *objectname*}
help security_alarm *tablename*

Description

The help statement displays information about database objects. In general, to display high-level information, specify help *objectname* (for example, help mytable); to display detailed information, specify help *objecttype objectname* (for example, help table mytable).

You can use the asterisk wildcard character (*) in object name specifications to display a selected set of objects. For details, see Wildcards and Help.

The following table lists help options:

Statement	Description
help	Displays object name, owner, and type for all tables, views, and indexes to which the user has access, and all synonyms owned by the user. System tables and temporary tables are not listed. Information is displayed in a one-line-per-object format.
help *objectname* {, *objectname*} (where *objectname* is the name of a table, view, or index)	Displays detailed information for specified objects; display format is block-style.
help comment column *tablename columnname* {, *columnname*}	Displays any comments defined for the specified columns.
help comment table *tablename* {, *tablename*}	Displays any comments defined for the specified tables.
help constraint *tablename*	Displays any constraints defined on columns of the specified table or on the entire table. For details about table constraints, see the create table statement description in the *SQL Reference Guide.* These constraints are not the same as the integrities displayed by the help integrities statement.
help default *tablename*	Displays any user-defined defaults defined on columns of the specified table
help help	Displays valid help statements.
help index *indexname* {, *indexname*}	Displays detailed information about the specified indexes.
help integrity *objectname* {, *objectname*}(where *objectname* is the name of a table or index)	Displays any integrity constraints defined on the specified tables or indexes. Integrity constraints are defined using the create integrity statement, described in this chapter.
help permit on procedure | table | dbevent | view *objectname* {, *objectname*} (where *objectname* is the name of a database procedure, table, event, or view)	Displays the permit numbers and text. The permit numbers are required for the corresponding drop permit statement.
help procedure *procedure_name* {, *procedure_name*}	Displays detailed information about the specified procedure.
help register *objectname*	Displays information about registered objects. For details about registering objects, see the *Database Administrator Guide.*

Statement	Description
help rule *rulename, {rulename}*	Displays the text of the create rule statement that defined the rule.
help security_alarm *tablename{, tablename}*	Displays all security alarms defined for the specified table. The information includes an index number you can use to delete the security alarm (using the drop security_alarm statement).
help synonym *synonym{synonym}*	Displays information about the specified synonyms. To display all the synonyms you own, specify help synonym *.
help table *tablename {, tablename}*	Displays detailed information about the specified tables.
help view *viewname {, viewname}*	Displays detailed information about the specified views.

Wildcards and Help

You can use the asterisk (*) wildcard to specify all or part of the owner or object name parameters in a help statement. The help statement displays only objects to which the user has access, which are:

- Objects owned by the user
- Objects owned by other users who have granted permissions to the user
- Objects owned by the DBA to which you have access

If you specify wildcards for both the owner and object name (*.*), help displays all objects to which you have access. To display help information about objects you do not own, you must specify the owner name, using the *schema.objectname* syntax. For details about schemas, see the *SQL Reference Guide*.

If you omit the owner name wildcard (that is, specify * instead of *.*), help displays the objects you can access without the owner prefix. The following examples illustrate the effects of the wildcard character:

`help *`	Display objects owned by the session's effective user.
`help davey.*`	Display all objects owned by "davey".
`help *.mytable`	Display all objects named "mytable" regardless of owner.
`help d*.*`	Display all objects owned by users beginning with "d".
`help *.d*`	Display all objects beginning with "d" regardless of owner.
`help *.*`	Display all objects regardless of owner.

Permissions

This statement is available to any user.

Examples

The following examples provide details.

Example 1

The following example displays a list of all tables in the database:

```
help
```

Example 2

The following example displays information about the "employee" table:

```
help employee
```

Example 3

The following example displays information about the "employee" and "dept" tables:

```
help employee, dept
```

Example 4

The following example displays information about the definition of the "highpay" view:

```
help view highpay
```

Example 5

The following example displays all permits issued on the "job" and "employee" tables:

```
help permit job, employee
```

Example 6

The following example lists all integrity constraints issued on the "dept" and "employee" tables:

```
help integrity dept, employee
```

Include

QUEL	EQUEL	KME
	*	

Includes an external file in source code.

Syntax

include [**inline**] *filename*

Description

The include statement provides a means to include external files in your program's source code (for example, variable declarations).

Filename must be a string constant that specifies the file to be included. If the file is a simple name (with an extension) it can be specified without quotes; however, if *filename* includes non-alphanumeric characters, the string constant must be quoted. *Filename* can be a logical (VMS) or system variable (UNIX) that specifies the location and name of the file to be included.

The file specified in an include statement can contain variable declarations and host code. Include files can contain include statements.

When the preprocessor encounters an include statement, it processes the include file and creates work files. Default filename extensions, both for the included file and work files, are host-language dependent. For more information, see the *Embedded QUEL Companion Guide*. The default extensions can be overridden with the -n and -o flags of the preprocessor command.

In addition to translating the include file, the preprocessor translates the ## include statement to the equivalent host language statement.

Include and include inline are processed differently. When the preprocessor encounters an include statement, it preprocesses the specified file separately, before including it. When the preprocessor encounters an include inline statement, it preprocesses the include file as if it were part of the original file. As a result, you can use include inline to complete partial statements.

Examples

The following examples provide details.

Example 1

The following example includes the contents of the file named "global.var" into an EQUEL program:

```
## include "global.var"
## /*
## ** the equel program can reference the data items
## ** declared in "global.var"
## */
```

Example 2

The following example incorporates the contents of the file named "messages.src" into a message statement. In this example, "messages.src" contains the text "Retrieved employee ". At runtime, the program retrieves an employee and displays "Retrieved employee *employee name*".

```
## retrieve (msgvar = employee.empname)
## {
##   message include inline "messages.src" msgvar
## }
```

Index

QUEL	EQUEL	KME
*	*	

Creates an index on an existing table.

Syntax

[##] **index** [**unique**] **on** *tablename* **is** *indexname*
(*columnname* {**,** *columnname*}**)**
[*with_clause*]

The optional *with* clause must consist of a comma-separated list of one of more of the following:

structure = btree | **cbtree** | **isam** | **cisam** | **hash** | **chash**
key = (*columnname* {**,** *columnname*})
fillfactor = *n*
minpages = *n*
maxpages = *n*
leaffill = *n*
nonleaffill = *n*
maxindexfill = *n*
location = (*locationname* {**,** *locationname*}**)**
allocation = *n*
extend = *n*

Description

The index statement creates an index on an existing table. Indexes can make retrieval and updating more efficient. A key is constructed from base table columns in the order you specify. A maximum of 32 columns can be specified per index. You can create any number of indexes for a base table.

Locationname specifies the location of the index you are creating; the location must exist, and the database must have been extended to the location. If no *locationname* is specified, the index is created in the default database location. If multiple *locationnames* are specified, the index is physically partitioned across the areas.

Structure is specified using the with structure option. The default index structure is isam. To specify the default index structure, use the -n flag when you invoke QUEL. To modify the storage structure of indexes, use the modify statement.

If key=(*column list*) is specified, the columns in *column list* must be an ordered subset of the leading columns specified in the index definition. For example, an index defined on columns "a," "b," "c" and "d" can be keyed on "a", or "ab," or "abc" or "abcd." (The default is "abcd" if the key clause is omitted.)

Indexes cannot be directly updated. When a primary table is changed, its indexes are automatically updated by the system. To minimize the time it takes to update a table, limit the number of indexes.

If you modify or destroy a primary table, indexes on the primary table are destroyed. You can modify and destroy an index (an index is also a table).

For details about the with clause options, see Modify.

Embedded Usage

Host string variables can be used to specify *tablename, indexname, columnname*, and the *with* clause.

The EQUEL preprocessor does not validate the syntax of the *with_clause*.

Considerations

- Only the owner of a table is allowed to create indexes on that table.
- You cannot create indexes on other indexes or on system tables.

Examples

The following examples provide details.

Example 1

The following example creates an index called "x" for the columns ename and age on table employee:

```
index on employee is x(ename, age)
```

Example 2

The following example creates an index called "ename", located on the area referred to by the locationname "remote":

```
index on employee is remote:ename(ename, age)
```

Ingres

QUEL	EQUEL	KME
	*	

Connects to a database.

Syntax

ingres [*flag* {,*flag*}] *dbname*

Description

The ingres statement connects an application program to a database. The ingres statement must precede any statements that operate on the database. A program can access only one database at a time. However you can connect, one at a time, to any number of databases. For information about flags, see the quel command description in the *Command Reference Guide.*

Embedded Usage

You can specify each *flag* using a quoted character string or a string variable. You can specify *dbname* using a host character string with or without quotes, or a string variable.

Example

The following example connects to the "personnel" database as user "neil", locking the database for exclusive use by specifying the -l option:

```
## userid     character_string(10)
   userid = "-uneil"
## ingres userid "-l" "personnel"
```

Inquire_ingres

QUEL	EQUEL	KME
	*	

Returns diagnostic information about the program's interaction with the database.

Syntax

inquire_ingres (*variable* **=** *object* {, *variable* = *object*})

Description

The inquire_ingres statement returns diagnostic information about the last database statement that was executed. Inquire_ingres and inquire_equel are synonymous.

Inquire_ingres must be issued immediately following the database statement in question, because the next EQUEL statement resets this diagnostic information.

Valid values for *object* are listed in the following Inquire_ingres Statement Parameters table:

Object	Data Type	Description
connection_target	character	Returns the node and database to which the current session is connected; for example, "bignode::mydatabase".
dbmserror	integer	Returns the number of the error caused by the last query.
endquery	integer	Returns 1 if the previous fetch statement was issued after the last row of the cursor, 0 if the last fetch statement returned a valid row. If endquery returns 1, the variables assigned values from the fetch are left unchanged.
errorno	integer	Returns the error number of the last query as a positive integer. The error number is cleared before each embedded QUEL statement; errorno is meaningful only immediately after the statement in question.
errortext	character	Returns the error text of the last query. The error text is only valid immediately after the database statement in question. The error text that is returned is the complete error message of the last error. A character string result variable of size 256 is sufficient to retrieve all error messages. If the result variable is shorter than the error message, the message is truncated. If there is no error message, a blank message is returned.
errortype	character	Returns "genericerror" if the DBMS Server returns generic error numbers to errorno, or "dbmserror" if it returns local DBMS error numbers to errorno. For information about generic and local errors, see the *SQL Reference Guide.*
prefetchrows	integer	Returns the number of rows the DBMS Server buffers when fetching data using readonly cursors. This value is reset every time a readonly cursor is opened; if your application is using this feature, be sure to set the value after opening a readonly cursor.

Object	Data Type	Description
programquit	integer	Returns 1 if the programquit option is enabled. If programquit is enabled, the following errors cause EQUEL applications to abort: ▪ Issuing a query when not connected to a database ▪ Failure of the DBMS Server ▪ Failure of communications services Returns 0 if applications continue after encountering such errors.
querytext	character	Returns the text of the last query issued; valid only if this feature is enabled. To enable or disable the saving of query text, use the set_ingres(savequery) statement. A maximum of 1024 characters is returned. If the query is longer, it is truncated to 1024 characters. If the receiving variable is smaller than the query text being returned, the text is truncated to fit. If you specify a null indicator variable in conjunction with the receiving host variable, the indicator variable is set to -1 if query text cannot be returned, 0 if query text is returned successfully. Query text cannot be returned if (1) savequery is disabled, (2) no query has been issued in the current session, or (3) the inquire_ingres statement is issued outside of a connected session.
rowcount	integer	Returns the number of rows affected by the last query. The following statements affect rows: append, delete, replace, retrieve, fetch, modify, index, and copy. If these statements generate errors, or if statements other than these are run, the value of rowcount is negative. Exception: for modify to truncated, inquire_ingres(rowcount) always returns 0. The value returned for rowcount is determined by the set update_rowcount option. For details, see Update_rowcount Option.
savequery	integer	Returns 1 if query text saving is enabled, 0 if disabled.
transaction	integer	Returns a value of 1 if there is a transaction open.

Example

This example shows the use of inquire_ingres to retrieve error message text:

```
## range of e is employee
loop until (i  > 10)

##  repeat replace e (sal = e.sal*1.1)
##   where e.empname = goodemps(i))

##  inquire_ingres (errno = errorno)

if errno  > 0
##    inquire_ingres (errmsg = errortext)
      print "ingres error: ", errno
      print errmsg
end if

  i = i + 1
end loop
```

Modify

QUEL	EQUEL	KME
*	*	

Changes properties of a table or index.

Syntax

[**##**] **modify** *tablename*|*indexname*
to *storage_structure* [**unique**]
[**on** *columnname* [**asc**|**desc**]{**,** *columnname* [**asc**|**desc**]}]
[*with_clause*]

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

allocation = *n*
extend = *n*
fillfactor=*n* (**isam**, **hash**, and **btree** only)
minpages=*n* (**hash** only)
maxpages=*n* (**hash** only)
leaffill=*n* (**btree** only)
nonleaffill=*n* (**btree** only)
newlocation=(*location_name* {**,** *location_name*})
oldlocation=(*location_name* {**,** *location_name*})
location=(*location_name* {**,** *location_name*})
compression [= ([[**no**]**key**] [**,**[**no**]**data**])] | **nocompression**
[no]persistence
unique_scope = row | statement

To move a table:

[**##**] **modify** *tablename*|*indexname* **to relocate**
with oldlocation = (*locationname* {**,** *locationname*}**),**
newlocation = (*locationname* {**,** *locationname*}**),**

To change locations for a table:

[**##**] **modify** *tablename*|*indexname* **to reorganize**
with location = (*locationname* {**,** *locationname*}**)**

To delete all data in a table:

[**##**] **modify** *tablename*|*indexname* **to truncated**

To reorganize a **btree** table's index:

[**##**] **modify** *tablename*|*indexname* **to merge**

To add pages to a table:

[**##**] **modify** *tablename*|*indexname* **to add_extend**
[**with extend =** *number_of_pages*]

where *number_of_pages* is 1 to 8,388,607.

Description

The modify statement enables you to perform the following operations:

- Change the storage structure of the specified table or index.
- Specify the number of pages allocated for a table or index, and the number of pages by which it grows when it requires more space.
- Add pages to a table.
- Reorganize a btree index.
- Move a table or index, or portion thereof, from one location to another.
- Spread a table over many locations or consolidate a table onto fewer locations.
- Delete all rows from a table and release its file space back to the operating system.
- Specify whether an index is recreated when its base table is modified.
- Specify how unique columns are checked during updates: after each row is inserted or after the update statement is completed.

You can change a table's location and storage structure in the same modify statement.

The modify statement operates on existing tables and indexes. When you modify a table, the DBMS Server destroys any indexes that exist for the specified table (unless the index was created with persistence, or the table is a btree and you are modifying the table to reorganize its index).

(The modify statement does not fire rules defined for the specified tables. For details about rules, see the *SQL Reference Guide.*)

Storage Structure Specification

Changing the storage structure of a table or index is often done to improve performance of access to the table. For example, change the structure of a table to heap before performing a bulk copy into the table to improve the performance of copy.

The *storage_structure* parameter must be one of the following table storage structures:

Structure	Description
isam	Indexed sequential access method structure, duplicate rows allowed unless the with noduplicates clause is specified when the table is created.
hash	Random hash storage structure, duplicate rows allowed unless the with noduplicates clause is specified when the table is created
heap	Unkeyed and unstructured, duplicated rows allowed, even if the with noduplicates clause is specified when the table is created.
heapsort	Heap with rows sorted and duplicate rows allowed unless the with noduplicates clause is specified when the table is created (sort order not retained if rows are added or replaced).
btree	Dynamic tree-structured organization with duplicate rows allowed unless the with noduplicates clause is specified when the table is created.

You cannot modify an index to heap or heapsort.

The DBMS Server uses existing data to build the index (for isam tables), calculate hash values (for hash tables) or for sorting (heapsort tables).

To optimize the storage structure of heavily-used tables (tables containing data that is frequently added to, changed, or deleted), modify those tables periodically.

The optional keyword unique requires each key value in the restructured table to be unique. (The key value is the concatenation of all key columns in a row.) If you specify unique for a table that contains non-unique keys, the DBMS Server returns an error and does not change the table's storage structure. For the purposes of determining uniqueness, a null is considered to be equal to another null.

You cannot specify unique for heap or heapsort tables.

The optional on clause determines the table's primary keys. You can only specify this clause when modifying to one of the following storage structures: isam, hash, heapsort, or btree. When the table is sorted after modification, the first column specified in this clause is the most significant key, and each successive specified column is the next most significant key.

If you omit the on clause when modifying to isam, hash, or btree, the table is keyed, by default, on the first column. When you modify a table to heap, you must omit the on clause.

When you modify a table to heapsort, you can specify the sort order as asc (ascending) or desc (descending). The default is ascending. When sorting, the DBMS Server considers nulls greater than any non-null value.

Modify...to Merge Option

When data is added to a btree table, the index automatically expands. However, a btree index does not shrink when rows are deleted from the btree table. To shrink a btree index, use the modify... to merge option. Modify...to merge affects only the index, and therefore usually runs a good deal faster than the other modify variants. Modify...to merge does not require any temporary disk space.

Modify...to Relocate Option

To move the data without changing the number of locations or storage structure, specify relocate. For example, to relocate the employee table to three different areas:

```
modify employee to relocate
with  oldlocation = (area1, area2, area3),
      newlocation = (area4, area5, area6);
```

The data on "area1" is moved to "area4", the data on "area2" is moved to "area5", and the data on "area3" is moved to area6. The number of areas listed in the oldlocation and newlocation options must be equal. The data in each area listed in the oldlocation list is moved without change to the corresponding area in the newlocation list. You can only use the oldlocation and newlocation options in the with clause when you specify relocate.

To change some but not all locations, specify only the locations to be changed. For example, move only the data on "area1" of the employee table:

```
modify employee to relocate
with  oldlocation = (area1),
      newlocation = (area4);
```

The DBMS Server is very efficient at spreading a table or index across multiple locations. For example, if a table is to be spread over three locations:

```
create table large (wide varchar(2000),
with location = (area1, area2, area3);
```

Rows are added to each location in turn, in 16-page (approximately 32 kilobyte) chunks. If it is not possible to allocate 16 full pages on an area when it is that area's turn to be filled, the table is out of space, even if there is plenty of room in the table's other areas.

Modify...to Reorganize Option

To move the data and change the number of locations without changing storage structure, specify reorganize. For example, to spread an employee table over three locations:

```
modify employee to reorganize
with location = (area1, area2, area3);
```

When you specify reorganize, the only valid with clause option is location.

Modify...to Truncated Option

To delete all the rows in the table and release the file space back to the operating system, specify modify...to truncated. For example, the following statement deletes all rows in the "acct_payable" table and releases the space:

```
modify acct_payable to truncated;
```

Using truncated converts the storage structure of the table to heap. You cannot specify any of the *with_clause* options when you modify to truncated.

Modify...to Add_extend Option

To add pages to a table, specify modify...to add_extend. To specify the number of pages to be added, use the extend=*number_of_pages* option. If you omit the with extend=*number_of_pages* option, the DBMS Server adds the default number of pages specified for extending the table. To specify the default, use the modify...with extend statement. If no default has been specified for the table, 16 pages are added.

With Clause Options

The following sections describe the remaining with clause options for the modify statement.

Fillfactor, Minpages, and Maxpages Options

Fillfactor specifies the percentage (from 1 to 100) of each primary data page that must be filled with rows, under ideal conditions. For example, if you specify a fillfactor of 40, the DBMS Server fills 40% of each of the primary data pages in the restructured table with rows. You can specify this option with the isam, hash, or btree structures. Take care when specifying large fillfactors because a nonuniform distribution of key values can later result in overflow pages and thus degrade access performance for the table.

Minpages specifies the minimum number of primary pages a hash table must have. Maxpages specifies the maximum number of primary pages a hash table can have. Minpages and maxpages must be at least 1. If both minpages and maxpages are specified in a modify statement, minpages must not exceed maxpages.

For best performance, the values that you choose for minpages and maxpages must be a power of 2. If you choose a number other than a power of 2, the DBMS Server can change the number to the nearest power of 2 when the modify executes. If you want to ensure that the number you specify is not changed, set both minpages and maxpages to that number.

Default values for fillfactor, minpages and maxpages are listed in the following table:

Storage Structure	Fillfactor	Minpages	Maxpages
hash	50	16	no limit
compressed hash	75	1	no limit
isam	80	n/a	n/a
compressed isam	100	n/a	n/a
btree	80	n/a	n/a
compressed btree	100	n/a	n/a

Leaffill and Nonleaffill Options

For btree tables, the leaffill parameter specifies to the DBMS Server how full to fill the leaf index pages. Leaf index pages are the index pages that are directly above the data pages. Nonleaffill specifies how full to fill the non-leaf index pages. Non-leaf index pages are the pages above the leaf pages. Specify leaffill and nonleaffill as percentages. For example, if you modify a table to btree, specifying nonleaffill=75, each non-leaf index page is 75% full when the modification is complete.

The leaffill and nonleaffill parameters can assist you in controlling locking contention in btree index pages. If you retain some open space on these pages, concurrent users can access the btree with less likelihood of contention while their queries descend the index tree. You must strike a balance between preserving space in index pages and creating a greater number of index pages; more levels of index pages require more I/O to locate a data row.

Default values for leaffill and nonleaffill are 70% and 80%, respectively.

Allocation Option

To specify the number of pages initially allocated to the table or index, use the with allocation option. By allocating disk space to a table, you can avoid runtime errors that result from running out of disk space.

The number of pages specified must be between 4 and 8,388,607 (the maximum number of pages in a table). If the specified number of pages cannot be allocated, the modify statement is aborted.

You can modify a table to a smaller size. If the table requires more pages that you specify, the table is extended and no data is lost. You can modify a table to a larger size, to reserve disk space for the table.

If the table is spread across multiple locations, space is allocated across all locations.

Extend Option

To specify the number of pages by which a table or index grows when it requires more space, use the with extend clause. The number of pages specified must be between 1 and 8,388,607 (the maximum number of pages in a table). By default, tables and indexes are extended by groups of 16 pages. If the specified number of pages cannot be allocated when the table must be extended (for example, during an insert operation), the DBMS Server aborts the statement and issues an error.

Compression Option

To specify data and key compression, use the with compression clause. Not all storage structures can be compressed, as shown in the following table:

Storage Structure	Base Table or Secondary Index	Can Compress Data?	Can Compress Key?
Hash	Base Table	Yes	No
	Secondary Index	Yes	No
Heap	Base Table	Yes	No
	Secondary Index	No	No
Btree	Base Table	Yes	Yes
	Secondary Index	No	Yes
ISAM	Base Table	Yes	No
	Secondary Index	Yes	No

To specify an uncompressed storage structure, specify with nocompression.

To compress both key and data for tables where this is valid (primarily btree), specify with compression, omitting the key/data clause. To compress data or keys independently of one another, specify with compression = (key|data). To explicitly suppress compression of data or keys, specify with compression = (nokey | nodata).

Location Option

To change the location of a table when modifying its storage structure, specify the location option. This option allows you to specify one or more new locations for the table. The locations specified must exist when the statement executes and the database must have been extended to those locations. For information about areas and extending databases, see the *Database Administrator Guide.*

Unique_scope Option

The unique_scope option specifies, for tables or indexes with unique storage structures, how uniqueness is checked during an update option. There are two options:

- unique_scope = row

  Uniqueness is checked as each row is inserted (for multi-row updates).

- unique_scope = statement

  Uniqueness is checked after the update statement is completed.

The default unique_scope is row.

[No]persistence Option

The [no]persistence option specifies whether an index is recreated when its related base table is modified. The [no]persistence option is valid only for indexes. There are two options:

- with persistence

  The index is recreated when its base table is modified.

- with nopersistence

  The index is dropped when its base table is modified.

  By default, indexes are created with nopersistence.

Embedded Usage

The preprocessor does not verify the syntax of the *with_clause*. You can specify the values in the *with_clause* options using host variables.

Permissions

You must be the owner of the specified table or a user with the security privilege.

Locking

The modify statement requires an exclusive table lock. Other sessions, even those using readlock=nolock, cannot access the table until the transaction containing the modify statement is committed.

Examples

The following examples provide details.

Example 1

Modify the "employee" table to an indexed sequential storage structure with eno as the keyed column:

```
modify employee to isam on eno
```

If "eno" is the first column of the "employee" table, the same result can be achieved by

```
modify employee to isam
```

Example 2

Perform the same modify, but request a 60% occupancy on all primary pages:

```
modify employee to isam on eno
  with fillfactor = 60
```

Example 3

Modify the "job" table to compressed hash storage structure with "jid" and "salary" as keyed columns:

```
modify job to hash on jid, salary
  with compression
```

Example 4

Perform the same modify, but also request 75% occupancy on all primary pages, a minimum of 7 primary pages and a maximum of 43 primary pages:

```
modify job to hash on jid, salary
  with compression, fillfactor = 75,
  minpages = 7, maxpages = 43
```

Example 5

Perform the same modify again but only request a minimum of 16 primary pages:

```
modify job to hash on jid, salary
  with compression, minpages = 16
```

Example 6

Modify the "dept" table to a heap storage structure and move it to a new location:

```
modify dept to heap with location=(area4)
```

Example 7

Modify the "dept" table to a heap again, but have rows sorted on the "dno" column and have any duplicate rows removed:

```
modify dept to heapsort on dno
```

Example 8

Modify the "employee" table in ascending order by "ename," descending order by "age," and have any duplicate rows removed:

```
modify employee to heapsort on ename asc,
  age desc
```

Example 9

Modify the "employee" table to btree on "ename," so that data pages are 50% full and index pages are initially 40% full:

```
modify employee to btree on ename
  with fillfactor = 50, leaffill = 40
```

Example 10

Modify a table to btree with data compression, no key compression. This is the format used by the (obsolete) cbtree storage structure:

```
modify table1 to btree
  with compression=(nokey, data)
```

Example 11

Modify an index to btree using key compression:

```
modify index1 to btree with compression=(key)
```

Example 12

Modify an index so it is retained when its base table is modified:

```
modify empidx to btree with persistence
```

Example 13

Modify a table, specifying the number of pages to be initially allocated to it and the number of pages by which it is extended when it requires more space:

```
modify inventory to btree
  with allocation = 10000, extend = 1000
```

Example 14

Modify an index to have uniqueness checked after the completion of an update statement:

```
modify empidx to btree unique on empid
  with unique_scope = statement
```

Open Cursor

QUEL	EQUEL	KME
	*	

Opens a cursor for processing.

Syntax

open cursor *cursor_name* [**for readonly**]

Description

The open cursor statement opens a cursor for processing. A cursor must be opened before it can be used to retrieve, append, or delete rows.

The retrieve clause of a declare cursor statement is evaluated when the cursor is opened. After you open a cursor, it is positioned immediately prior to the first row of the result table. To advance the cursor and retrieve the first row, your application program must issue a retrieve cursor statement.

The for readonly clause indicates that, even though the cursor has been declared for update, it is being opened for reading only. Opening a cursor for readonly improves database access time. You cannot use a readonly cursor to append, update, or delete rows.

Different cursors can be open at the same time only within a multi-query transaction. The same cursor can be opened and closed any number of times in a single program. A cursor must be closed, however, before it can be reopened.

The *cursor_name* parameter must be a declared cursor.

Embedded Usage

You can specify *cursor_name* using a string constant or a host language variable.

Print

QUEL	EQUEL	KME
*		

Prints tables.

Syntax

print *tablename* {**,** *tablename*}

Description

The print statement displays the contents of the specified tables on your terminal (or standard output). The format of the display is determined by flags that are specified when QUEL is invoked. For information about these flags, see the quel command description in the *Command Reference Guide.*

Print truncates and pads as necessary. Non-printing and control characters are displayed in a manner similar to the way they are specified in string constants. For example, carriage return is printed as "\r" and the "bell" character (octal value 7) is printed as "\007".

The print statement leaves enough space in each text column to accommodate the declared column width. If there are control characters in a text string, the number of characters printed can be greater than the width of the column. In this case, the printed columns do not align.

To print a table, you must own the table or have retrieve permission.

Examples

The following examples provide details.

Example 1

The following example prints the "employee" table:

```
print employee
```

Example 2

The following example prints both the "employee" and "dept" tables:

```
print employee, dept
```

Range

QUEL	EQUEL	KME
*	*	

Associates a range variable with a table.

Syntax

[##] **range of** *range_var* **is** *tablename* {, *range_var* **is** *tablename*}

Description

A range statement associates a range variable with the table or view specified by *tablename*. A range declaration remains in effect until:

- The session ends, or
- The variable is redeclared by another range statement, or
- Its table or view definition is destroyed

Not all range variables are declared using the range statement: the table name you specify in a retrieve statement is considered to be an implicit or default range variable.

Range variables enable you to treat the same table as though it were two separate tables. In the following example, range variables "e" and "m" allow you to extract employees and their managers from the same table.

```
range of d is dept
range of e is employee
range of m is employee
retrieve (e.ename, mgr=m.ename)
where e.dept=d.dno and d.mgr=m.eno
```

Considerations

- Default and explicitly declared range variables cannot be used as though they were identical, because they refer to different copies of the same table. Using a default and a declared range variable in the same query results in a *disjoint query*, whose results are seldom what you want and often disastrous.
- If you are using EQUEL, *range_var* and *tablename* can be specified using host variables.
- A maximum of 126 range variables can be in effect at any time. After the 126th range statement, the least recently used range variable is replaced by the next range statement. This limit includes both default and explicitly declared range variables.

Examples

The following examples provide details.

The first three examples illustrate a common error: *disjoint queries*, which are queries that incorrectly mix declared and implicit range variables.

Example 1

This example inadvertently deletes all rows in the "dept" table, not just the rows where the "dno" value is 1, because "d" and "dept" refer to different copies of the same table:

Wrong:

```
range of d is dept
delete d where dept.dno=1
```

Example 2

The following example inadvertently returns the Cartesian product of the two tables, not just the "ename" and "dept" values from each row:

Wrong:

```
range of e is emp
retrieve (e.ename, emp.dept)
```

Example 3

The following example inadvertently replaces the age value for all rows, not only for the "jones" rows:

Wrong:

```
range of e is emp
replace emp (e.age=e.age+1) where e.ename = "jones*"
```

Example 4

This example correctly declares range variable "e" to range over the "employee" table and "d" over the "dept" table:

Right:

```
range of e is employee, d is dept
```

Relocate

QUEL	EQUEL	KME
*	*	

Relocates tables.

Syntax

[##] **relocate** *tablename* **to** *locationname*

Description

The relocate statement moves a table from its current location to the location specified by *locationname*. You can't relocate a table that exists on multiple locations–you must use the modify statement to move it. Relocate takes an exclusive table lock. Other sessions cannot access the table until the relocation is complete.

Embedded Usage

You can specify *tablename* and *locationname* using host string variables.

Considerations

- You must own the specified table.
- *Locationname* must exist, and the database must have been extended to that location.

Example

The following example relocates the table "employee" to the area defined as the "remote_loc" location.

```
## relocate employee to remote_loc
```

Replace

QUEL	EQUEL	KME
*	*	

Replaces values of columns in a table.

Syntax

[##] [**repeat**] **replace** *range_variable* (*target_list*) [**where** *qual*]

Description

The replace statement updates the values of the columns specified in the *target_list* for all rows in the table that satisfy the where clause. Only columns to be modified need appear in the *target_list*. To set a nullable column to null, specify the keyword null.

The replace statement also has a param version; the param function replaces the list of column names and expressions in the target list. Param statements are not supported in all host languages. For more information, see the *Embedded QUEL Companion Guide*.

You can reduce the overhead of frequently executed replace statements by specifying repeat replace. The repeat keyword directs the DBMS Server to encode and save its execution plan when the replace is first executed. This encoding can account for significant performance improvements on subsequent executions of the same replace. (If the repeat option is specified, program variables which appear on the right hand side of an equals sign (=) must be preceded by the @ sign.)

Embedded Usage

You can use host variables to specify range variables, column names, and expressions (including expressions that use null indicator variables). You can use a host string variable to specify the where clause (useful in conjunction with the forms system query mode when your application must construct queries from user-specified parameters).

Considerations

- Only the owner of a table or a user with replace permission on the table can replace values.
- The replace statement fires any rules defined on the specified table that is fired by an equivalent SQL update statement. Rules are part of the Knowledge Management Extension. For more information, see the *SQL Reference Guide*.
- If the table was created with no duplicate rows allowed, you cannot execute a replace that creates a duplicate row.
- If the row update violates an integrity constraint, the update is not performed.
- Do not mix explicitly declared range variables with default range variables. Mixing range variables results in a disjoint query, which can seriously corrupt your data. For details, see Range.

Examples

The following examples provide details.

Example 1

The following example replaces the name and salary of the employee whose ID number is specified by the variable "numvar":

```
## range of e is employee
## replace e (empname = namevar,
    salary = salvar:indvar)
## where e.empnum = numvar
```

Example 2

A param version of the above. This version uses a dynamically created where clause, specified in a host string variable:

```
addresses(1) = address_of(namevar)
addresses(2) = address_of(salvar)
addresses(3) = address_of(indvar)
target_list = "empname = %c, salary = %f4:%i2"
## replace e (param (target_list, addresses))
##  where wherevar
```

Example 3

The following example gives all employees who work for Smith a 10% raise:

```
range of e is employee, m is employee, d is dept
replace e(salary=1.1*e.salary) where e.dept=d.dno and
    d.mgr=m.eno  and m.ename="*smith*"
```

Example 4

The following example replaces Jones' salary with null:

```
range of e is employee
replace e (salary=null) where e.ename="jones"
```

Example 5

Do not do this! This disjoint query changes all rows (because "e" and "employee" are separate range variables):

Wrong:

```
range of e is employee
replace e (salary=3500) where employee.ename="jones"
```

Replace Cursor

QUEL	EQUEL	KME
	*	

Updates values of columns in a single row in a table.

Syntax

replace cursor *cursor_name* (*target_list*)

Description

The replace cursor statement updates the values in the row currently pointed to by *cursor_name*. If the cursor is not pointing to a row, (for example, after an open cursor or a delete cursor statement), the DBMS Server displays an error message. If the row the cursor is pointing to has been deleted from the underlying database table (as the result of a non-cursor delete, for example), no row is updated. Replace cursor does not advance the cursor.

There are two update modes:

- Deferred mode

  Updates take effect when the cursor is closed. There can be only one cursor open in deferred mode at a time.

- Direct mode

  Updates are performed immediately. If you are using direct mode, avoid performing updates or deletes that change the order of rows because the sequence in which the cursor returns rows is affected.

If your application issues two cursor replace statements without advancing the cursor before the second (using retrieve cursor):

- A direct update mode cursor updates the same row twice
- A deferred update mode cursor does not perform the second update; an error message is issued

The cursor specified in the replace cursor must be open. The columns in the target list must have been declared for update and must be updatable. For example, you cannot update derived columns. See Declaring a Cursor.

If your host language supports the param version of the target list, see the *Embedded QUEL Companion Guide* for details.

Embedded Usage

You can use host language variables to specify *cursor_name*, column names, and constant expressions in the target list or the entire target list.

Examples

This example gives all employees except employee number 150 a 30% raise:

```
## range of e is employee
## declare cursor cursor1 for
##  retrieve (e.empname, e.empnum, sal = e.salary)
##  where e.empnum <> 150
##  for update of (salary)

## loop until no more rows
##  retrieve cursor cursor1 (namevar, numvar, salvar)

/* last row? */
##   inquire_ingres (thatsall = endquery)
if thatsall = 0 then
    if salvar < 30,000 then
##   replace cursor cursor1 (salary = salvar * 1.3)
    end if
  end if

  end loop
```

Retrieve

QUEL	EQUEL	KME
*	*	

Retrieve rows from a table.

Syntax

Interactive QUEL syntax:

retrieve [[**into**] *tablename*] [**unique**]
(**target_list**) [**where** *qual*]
[**sort** [**by**] *columnname* [**:***sortorder*] {**,** *columnname* [**:***sortorder*]}]
[**order** [**by**] *columnname* [:*sortorder*] {,*columnname*[**:***sortorder*]}]
[**with** *with_clause*]

The with clause is valid only when retrieving into a table. The with clause consists of the keyword with followed by a comma-separated list of one or more of the following options:

structure=*storage structure name*
key=(*column list*)
[**no**]**journaling**
[**no**]**duplicates**
location=*locationame*
fillfactor=1...100%
minpages=(>0)
maxpages=(>0)
nonleaffill=1...100%
leaffill/indexfill=1...100%
maxindexfill=ignored
allocation=(>0) (only for **retrieve into**)
extend=(>0) (only for **retrieve into**)

Embedded QUEL syntax, to retrieve into host variables:

[**repeat**] **retrieve** [**unique**] **(***variable=result_expression*
{*,variable = result_expression*})
[**where** *qual*] [**sort** [**by**] *result_column* {*, result_column*}]
[{
*program code*
}]

Description

The retrieve statement fetches all rows that satisfy the criteria specified in the where clause, and optionally stores the rows in a new table. To retrieve all columns from a table, specify *tablename.*all.

If you are using interactive QUEL, you can display the results; if you are using embedded QUEL (EQUEL), you can store the resulting rows in host language variables, enabling your application program to process the rows.

To store the results of the retrieve in a new table, specify into *tablename*. (In interactive QUEL, if you do not specify into *tablename*, the result is displayed.) A table with this name (owned by the user) must not already exist. The current user is the owner of the new table.

The new table's column names are specified in the *target_list* result column names. If the source column has a simple default defined, the result column inherits the default. For details about column defaults, see the create table statement description in the *SQL Reference Guide.*

The default structure for *tablename* is cheap (compressed heap); if sort by is specified, the default structure is cheapsort. You can override the default structure using the set ret_into statement, described in this chapter, or the with structure clause. You can specify the characteristics of the new table using the optional with clause. For details about these parameters, see Modify.

Locationname specifies the location where the table is to be created. The location must exist, and the database must have been extended to the corresponding area. If no location is specified, the default location for the database is assumed.

To remove duplicate rows from the result, specify the keyword unique. If you specify unique, rows are sorted on all the columns in the *target_list* (beginning with the first column) and duplicate rows are removed from the result.

To sort a table without removing duplicate rows, specify order by. To sort and remove duplicate rows, specify sort by. (Order by and sort by are mutually exclusive options.

Retrieve unique with an order by clause is functionally equivalent to retrieve with a sort by clause.) By default, rows are sorted in ascending order. You can override this default by specifying a *sortorder* of descending (or d) in the sort by or order by clause.

When you use the sort by or order by clause, you must specify the column name that appears in the result table. For example, the following two retrieve statements produce the same results: the first one sorts by base table column "ename", and the second assigns "ename" to the result column "person" and sorts by "person".

```
retrieve (e.ename, e.dept) sort by ename
retrieve (person=e.ename, e.dept) sort by person
```

However,

```
retrieve (person=e.ename, e.dept) sort by ename
```

is incorrect: the result column is "person", but the sort clause incorrectly specifies the base table column "ename".

Retrievals in Embedded QUEL

To retrieve a single row, omit the code block (enclosed in curly braces) that follows the retrieve statement. If more than one row fulfills the where clause of the query, the DBMS Server returns a single row, though not necessarily the first qualifying row.

To retrieve a set of rows, you must create a retrieve loop. To create a retrieve loop, follow the retrieve statement with a block of code enclosed in curly braces. The code block can contain a mixture of host language and EQUEL statements.

Within the retrieve loop code block, your application must not issue any other statements that access the database–this causes a runtime error. To see how rows and tables can be manipulated and updated while data is being retrieved, see Data Manipulation with Cursors in the "Embedded QUEL" chapter.

To abort a retrieve loop, use the endretrieve statement. The endretrieve statement must be within the block delimited by curly braces. Do not use a host language goto statement or return statement to exit the loop: exiting a loop using a goto or return causes the next EQUEL statement that accesses the database to fail, and the DBMS Server displays a message indicating that database statements cannot be nested within a retrieve loop.

To find out how many rows have been retrieved, use the inquire_ingres statement with the rowcount parameter. Used within the retrieve loop, rowcount indicates the number of rows retrieved so far. Placed immediately following the loop, it indicates the total number of rows retrieved. After a non-looped retrieve, inquire_ingres(rowcount) returns the number of rows that met the where clause.

The results of the retrieval are assigned to the specified host variables. If no rows are retrieved, the contents of the host variables remains unchanged. You must use numeric variables to receive numeric results and string variables to receive string results. Each result variable can be associated with an indicator variable to detect null data. For more information, see Indicator Variables in the "Embedded QUEL" chapter.

You can reduce the overhead required by frequently executed retrieve statements by specifying repeat retrieve. The repeat keyword directs the DBMS Server to encode the retrieve and save its execution plan when it is first executed. This encoding can improve performance on subsequent executions of the same retrieve.

Embedded Usage

You can use host variables to specify *target_list* expressions, including range variables, table names, column names, and numeric or character expressions. Host variables can be used to specify expressions in the where clause, or the complete where clause.

If a result column name is the same as the name of a host variable, you must dereference the column name by preceding it with a pound sign (#).

The EQUEL preprocessor does not validate the syntax of the with clause.

If repeat is specified, program variables which change each time the query is executed and which appear on the right hand side of an equals sign (=) must be preceded by the @ sign. Result variables must not be marked in this way. Variables (including @ variables) cannot be changed within a retrieve loop.

Considerations

- Only the table's owner and users with retrieve permission can retrieve from a table.
- (Interactive) The format in which columns are displayed can be defined when the Terminal Monitor connects to the database. For details, see the quel command description in the *Command Reference Guide.*
- (Interactive) Retrieve displays non-printing and control characters in a manner similar to the way they are specified in string constants. For example, carriage return is printed as "\r" and the "bell" character (octal value 7) is printed as "\007". The retrieve statement leaves enough space in each text column to accommodate the declared length of the column. If there are control characters in a text string, it is possible that the number of characters printed are greater than the width of the column, and the printed columns do not line up.
- The retrieve statement also has a param version for greater flexibility at runtime. Param statements are not supported in all host languages. For more information, see the *Embedded QUEL Companion Guide*.

Examples

The following examples provide details.

Example 1

The following example illustrates the use of retrieve to look up the name and salary of the employee number specified in host variable "numvar". Note the use of the null indicator variable with column "salary":

```
## range of e is employee
## retrieve (namevar = e.empname,
## salvar:indvar = e.salary)
##   where e.empnum = numvar
```

Example 2

In the following example, the constant "Name:" is assigned to result column "title", and the content of the "empname" column is assigned to host variable "namevar":

```
## range of e is employee
## retrieve (title = "name: ", namevar = e.empname)
##  where e.empnum >= 148 and e.age = agevar
```

Example 3

The following example illustrates the use of retrieve to print information from the "employee" table for the employee whose number and name correspond to "numvar" and "namevar". Because this statement is issued many times (in a subprogram, perhaps), it is formulated as a repeat query. The "@" sign is required on only those variables substituting for constant values:

```
## repeat retrieve (empgrade = e.egrade,
##  empsal = e.salary)
##  where e.eno = @numvar and e.ename = @namevar
```

Example 4

The following example illustrates the use of retrieve to scan an entire table and generate a report. If an error occurs, the retrieve loop is aborted. No database statements are allowed inside the retrieve loop (within the curly braces).

Note the sort clause and the use of dereferenced variable names as result column names in the sort by clause:

```
error = 0
## range of e is employee
## range of d is department

## retrieve (empid = e.empnum, empname = e.#empname,
##   empage = e.eage, empsal = e.salary,
##   empdept = d.dname)
##  where e.edept = d.deptno
##  sort by #empdept, #empname
## {
 generate report of information
 if error condition then
   error = 1
## endretrieve
   end if
## }
## /* transferred here by completing the retrieval
## or because the endretrieve statement was issued
## */
  if error = 1
##  inquire_ingres (countvar = rowcount)
    print "error encountered after row", countvar
  else
    print "successful addition and reporting"
  end if
```

Example 5

The following example illustrates the use of a string variable to specify the where clause:

```
run forms in query mode
construct where_cond from user input on form
## range of employee is e
## retrieve ( empname = e.#empname, empid = e.empnum,
##    empmgr = e.mgr ) where where_cond
## {
 load a table field with empname, empid, empmgr
## }
display table field for browsing
```

Retrieve Cursor

QUEL	EQUEL	KME
	*	

Retrieves data into host variables using a cursor.

Syntax

retrieve cursor *cursor_name* (*variable*
{**,** *variable*})

Description

The retrieve cursor statement advances the cursor one row and loads the values specified in the retrieve clause of the declare cursor statement into the specified host variables. The retrieve cursor statement is the only way to advance a cursor.

Embedded Usage

You can specify *cursor_name* using a string constant or a host language variable.

Considerations

- A retrieve cursor can only be issued if *cursor_name* has been declared and is currently open.

- There must be a one-to-one correspondence between variables specified in the retrieve cursor target list and columns in the retrieve clause of the declare cursor statement. The preprocessor generates a warning if there is a mismatch between the cursor declaration and its use in a retrieve cursor statement.
- The retrieve cursor statement also has a param version in some host languages. For more information on param statements, see the *Embedded QUEL Companion Guide*.

Examples

The following examples provide details.

Example 1

The following example illustrates typical use of retrieve cursor, with associated cursor statements:

```
## ename    character_string(26)
## age      integer
## declare cursor cursor1 for
##  retrieve (employee.empname, employee.#age)
##  sort by empname
## open cursor cursor1
loop until no more rows
##  retrieve cursor cursor1 (ename, age)
    print ename, age
end loop
## close cursor cursor1
```

Example 2

A param version of the above:

```
addresses(1) = address_of(name)
addresses(2) = address_of(age)
target_list = "%c, %i4"
## retrieve cursor cursor1
##  (param (target_list, addresses))
```

Example 3

The following example illustrates the use of an indicator variable in a retrieve cursor statement:

```
## retrieve cursor2 (name, salary:indicator_var)
```

Save

QUEL	EQUEL	KME
*	*	

Saves a base table until a specified date.

Syntax

[##] **save** *tablename* until *month day year*

Description

The save statement enables you to specify a table's expiration date. The verifydb command destroys expired tables. (Tables are not destroyed automatically upon expiration.) Only the owner of a table can save that table. By default, tables have no expiration date when created. To clear an expiration date, omit the until clause. For example, save mytable clears any expiration date from "mytable".

Month must be an integer from 1 through 12 or the name of the month, abbreviated or spelled out. *Day* must be the day of the month (1-31), and *year* must be fully specified (for example, 1999 or 2003).

Embedded Usage

You can specify *tablename* using a host variable. You can specify the complete date using a host variable, or specify the month as a (quoted or unquoted) string and the day and year using integer literals or variables.

Considerations

System tables have no expiration date.

Examples

The following examples provide details.

Example 1

The following example saves the "employee" table until December 31, 2003:

```
## save employee until december 31 2003
```

Example 2

The following example uses a variable to specify the expiration date:

```
save_date = "december 31 2003"
## save employee until save_date
```

Savepoint

QUEL	EQUEL	KME
*	*	

Declares a savepoint marker within a multi-query transaction.

Syntax

[##] **savepoint** *savepoint_name*

Description

The savepoint statement declares a savepoint within a multi-query transaction (MQT). Savepoints are used in conjunction with the abort statement. Abort allows you to specify a savepoint. An abort to that savepoint undoes all updates performed between the savepoint and the abort statement. *Savepoint_name* must be a valid, unquoted object name. Declaring a savepoint closes any open cursors.

There is no limit to the number of savepoints that you can declare within an MQT. You can use the same *savepoint_name* more than once. However, only the most recently declared use of a *savepoint_name* is active within the MQT. In other words, if you abort to a savepoint whose name is used more than once, the transaction is backed out to the most recent use of the *savepoint_name*.

All savepoints within a transaction are rendered inactive when the transaction is terminated (either committed with end transaction, or aborted with abort or by the system as the result of deadlock or timeout).

For more information on transactions, see Transactions in the "Embedded QUEL" chapter and Aborting Transactions in this chapter.

Example

This example shows a typical use of the savepoint statement. During each loop of the program, a savepoint is declared, enabling the program to back out updates in the event of an error:

```
## begin transaction
  saveindex = 0
  loop until finished processing
 saveindex = saveindex + 1
## savepoint saveindex
  process data
  if error condition then
##  abort to saveindex
    saveindex = saveindex - 1
  end if
 end loop
## end transaction
```

Set

QUEL	EQUEL	KME
	*	

Sets a session option.

Syntax

[##] **set aggregate** [**no**]**project**

[##] **set joinop** [**no**]**timeout**

[##] **set journaling**|**nojournaling** [**on** *tablename*]

[##] **set lockmode session**|**on** *tablename* **where**
[**level = page**|**table**|**session**|**system**]
[, **readlock = nolock**|**shared**|**exclusive**|**session**|**system**]
[, **maxlocks** = *n*|**session**|**system**]
[**, timeout =** *n*|**session**|**system**|**nowait**]

[##] **set** [**no**]**printqry**

[##] **set** [**no**]**qep**

[##] **set ret_into heap**|**cheap**|**heapsort**|**cheapsort**|**hash**|
chash|**isam**|**cisam**|**btree**|**cbtree**

[##] **set [no]logging**

[##] **set [no]optimizeonly**

[##] **set session with on_error = rollback**
statement | **transaction**

[##] **set update_rowcount changed** | **qualified**

The following **set** options are part of the Knowledge Management Extension.

[##] **set nomaxio** | **maxio** *value*

[##] **set nomaxrow** | **maxrow** *value*

[##] **set nomaxquery** | **maxquery** *value*

Description

The set statement specifies a runtime option for a single session (interactive or embedded). The option remains in effect until the end of the session, or until changed by another set statement.

Set Aggregate [No]project Option

Specifies whether the DBMS Server returns zero values for aggregate functions that use the by clause. In the following example, if you specify the project option, the DBMS Server returns a value for each department, zero for those departments that have no employees earning over $10,000. If you specify noproject, departments that have no employees earning over $10,000 are omitted:

```
count(emp.salary by emp.dept where emp.salary > 10000)
```

Set Joinop [No]timeout Option

This statement turns the optimizer's timeout feature on and off. If timeout is on (the default), the query optimizer stops checking query execution plans when it believes that the best plan that it has found takes less time to execute than the amount of time already spent searching for a plan. If you issue a set joinop notimeout statement, the optimizer does not time out when checking query execution plans.

This option is often used with the set qep option to ensure that the optimizer is picking the best possible query plan.

The default is set joinop timeout.

Set [No]journaling Option

The set journaling|nojournaling statement sets the session default for the create statement. If you specify set journaling, tables are created with journaling unless you specify no journaling in the create statement. If you specify set nojournaling, tables are created with journaling turned off, unless you specify journaling in the create statement.

If you set journaling on an individual table, journaling for the specified table begins at the next checkpoint. For more information about checkpoints, see the *Database Administrator Guide*.

Set Lockmode Option

The set lockmode option allows you to specify a number of different types and levels of locks. Valid values for set lockmode parameters are listed in the following table:

Lockmode	Description	
level	Specifies locking level as follows:	
	page	Specifies locking at the level of the data page (subject to escalation criteria as described below).
	table	Specifies table-level locking.
	session	Specifies the current default for your session.
	system	Specifies that the DBMS Server starts with page level locking, unless it estimates that more than *maxlocks* pages are locked, in which case, table level locking is used.
readlock	Specifies lock mode for tables being read but not updated. You can specify any of the following readlock modes:	
	nolock	Specifies no locking when reading data.
	shared	Specifies the default mode of locking when reading data.
	exclusive	Specifies exclusive locking when reading data (useful in "retrieve-for-update" processing within a multi-query transaction).
	session	Specifies the session default readlock.

Lockmode	Description	
	system	Specifies the system readlock default.
maxlocks	Specifies the number of locks at which locking escalates from page level to table level. The number of locks available to you is dependent upon your system configuration. The following are valid values for maxlocks:	
	n	Specifies the number of page locks to allow before escalating to table level locking. The default is 10; n must be greater than 0.
	session	Specifies the current maxlocks default for your session.
	system	Specifies the system default for maxlocks. If you specify page level locking and the number of locks granted during a query exceeds the system-wide lock limit or the operating system's locking resources are depleted, locking escalates to table level. This escalation occurs automatically and is independent of the user.
timeout	Specifies the amount of time to wait for a lock. If the DBMS Server cannot grant the lock request within the specified time, the query that requested the lock aborts. Valid values for timeout are:	
	n	Specifies the number of seconds to wait for a lock; to specify an indefinite wait, set timeout to 0.
	session	Specifies the session default.
	system	Specifies the system timeout default.
	nowait	Specifies that when a lock request is made that cannot be granted without incurring a wait, control is immediately returned to the application that issued the request.

The system defaults for each of the parameters are listed in the following table:

Parameter	Default
level	Dynamically determined by the DBMS Server
readlock	Shared
maxlocks	50
timeout	0 (no timeout)

At the beginning of a session, the system defaults are in effect. If you override them with other values using the set lockmode statement, you can revert back to the system defaults by specifying system, or the session defaults by specifying session.

The set lockmode statement cannot be issued within a transaction, except for the following statement:

set lockmode … with timeout=<n|session|system|nowait>

Set [No]printqry Option

The set printqry statement displays each query and its parameters as it is passed to the DBMS Server for processing. Set noprintqry disables this feature.

Set [No]qep Option

The set qep statement displays a summary of the query execution plan chosen for each query by the optimizer. To disable this option, use set noqep. For a discussion of query execution plans, see the *Database Administrator Guide*.

Set Ret_into Option

The set ret_into statement sets the storage structure for tables created by retrieve into statements that do not specify the with structure clause.

For example, this first sequence of statements:

```
set ret_into hash
retrieve into temp (id = ...)
```

does the same as this second sequence of statements:

```
retrieve into temp (id = ...)
modify temp to hash
```

Both examples create table "temp" as hash. For all table types except heap and cheap, the table is automatically indexed on the first column. (The default storage structure for a table created by the retrieve into statement is cheap.)

Set [No]logging Option

To suspend transaction logging for the current session, issue the set nologging statement; to resume logging, issue the set logging statement. To issue this statement, you must be the DBA of the database to which your session is connected.

Suspending transaction logging can improve the performance of large batch updates. However, use set nologging with extreme caution, and consider checkpointing the database before suspending logging. Be advised that, when transaction logging is suspended:

- The abort statement has no effect.
- Any error (including errors resulting from a database statement, forced abort, deadlock, or timeout) leaves the database in an unknown state.

Set [No]optimizeonly Option

This statement specifies whether query execution halts after the optimization phase. To halt execution after the query has been optimized, specify set optimizeonly; to continue query execution after the query is optimized, specify set nooptimizeonly. To view query execution plans (QEP's) without executing a query, use set optimizeonly in conjunction with set qep.

Set [No]maxio Option

This statement is part of the Knowledge Management Extension.

The set maxio statement restricts the estimated number of I/O operations that can be used by one query. *Value* must be less than or equal to query_io_limit. If you issue a set nomaxio statement, the maximum number of I/O operations is set to query_io_limit.

Set [No]maxrow Option

This statement is part of the Knowledge Management Extension.

The set maxrow statement restricts the estimated number of rows that can be returned by one query. *Value* must not exceed query_row_limit. If you issue a set nomaxrow statement, the allowed number of rows is set to query_row_limit.

Set [No]maxquery Option

The set maxquery statement is an alias for the set maxio statement, discussed above.

Session With On_error Option

The set session with on_error statement enables you to specify how transaction errors are handled in the current session. To direct the DBMS Server to roll back the effects of the entire current transaction if an error occurs, specify rollback transaction. To rollback only the current statement (the default), specify rollback statement. To determine the current status of transaction error handling, issue the retrieve (x=dbmsinfo("on_error_state")) statement.

Specifying rollback transaction reduces logging overhead, and can help performance; the performance gain is offset by the fact that, if an error occurs, the entire transaction is rolled back, not the single statement that caused the error.

The following errors always roll back the current transaction, regardless of the current transaction error-handling setting:

- Deadlock
- Forced abort
- Lock quota exceeded

To determine if a transaction was aborted as the result of a database statement error, issue the retrieve (x=dbmsinfo("transaction_state")) statement. If the error aborted the transaction, this statement returns 0, indicating that the application is currently not in a transaction.

You cannot issue the set session with on_error statement from within a database procedure or multi-query transaction.

Update_rowcount Option

The set update_rowcount statement specifies the nature of the value returned by the inquire_ingres(rowcount) statement. Valid options are:

- Changed

  Inquire_ingres(rowcount) returns the number of rows changed by the last query.

- Qualified

  Inquire_ingres(rowcount) returns the number of rows that qualified for change by the last query.

Qualified is the default setting, for example, for the following table:

column1	column2	column3
Jones	000	green
Smith	000	green
Smith	000	green

and for the following query:

```
replace test_table (column1 = "Jones")
  where column2 = 000 and column3 = "green";
```

The DBMS Server, for reasons of efficiency, does not actually update the first row, because its "column1" already contains "Jones." However, the row does qualify for updating by the query. For the preceding query, if the update_rowcount option is set to changed, inquire_ingres(rowcount) returns 2 (the number of rows actually changed), but if the update_rowcount option is set to qualified, inquire_ingres(rowcount) returns 3 (the number of rows that qualified to be changed).

To determine the setting for the update_rowcount option, issue the retrieve (dbmsinfo(x='update_rowcnt')) statement.

Examples

The following examples provide details.

Example 1

The following example illustrates the use of the set statement to create three tables with journaling enabled and one without:

```
set journaling
create withlog1 ( ... )
retrieve into withlog2 ( ... )
set nojournaling
create withlog3 ( ... ) with journaling
create nolog1 ( ... )
```

Example 2

The following example creates a few tables with different structures:

```
retrieve into a ( ... )    /* heap */
set ret_into hash
retrieve into b (id = ... )   /* hash on key id */
retrieve into c (id = ... ) sort by id
/* heap, sorted on id */
set ret_into heap
retrieve into d (id = ... )   /* heap again */
```

Example 3

The following example illustrates the setting of lockmode parameters for the session and for a specific table:

```
set lockmode session where level = page,
readlock = nolock, maxlocks = 50, timeout = 10
set lockmode on employee where level = table,
readlock = exclusive, maxlocks = session, timeout = 0
```

Example 4

The following example resets your session default locking characteristics to the system defaults:

```
set lockmode session where level = system,
readlock = system, maxlocks = system, timeout = system
```

Set_ingres

QUEL	EQUEL	KME
	*	

Enables or disables various runtime attributes.

Syntax

set_ingres (*object* = *value* {**,** *object* = *value*})

Description

The set_ingres statement allows your application program to:

- Enable or disable debugging features
- Specify the number of rows the DBMS Server prefetches when retrieving rows using cursors
- Specify whether the DBMS Server aborts or continues a session when certain errors occur

The set_ingres statement overrides any settings of II_EMBED_SET. For more information about II_EMBED_SET, see the *System Administrator Guide.* You must terminate the set_ingres statement according to the rules of your host language.

The following are valid parameters for the set_ingres statement (SQL-specific parameters are omitted):

Parameter	Data Type	Description
prefetchrows	integer	Specifies the number of rows the DBMS Server prefetches when retrieving data using cursors. Valid arguments are: 0: (default) the DBMS Server determines the number of rows to prefetch. 1: Disables prefetching; each row is fetched individually. *n*: (positive integer) Specifies the number of rows the DBMS Server prefetches.
printqry	integer	Turns the printqry debugging feature on or off. As the application executes, printqry prints query text and timing information to the file "iiprtqry.log" in the current directory. Specify 1 to turn printqry on, 0 to turn printqry off.
qryfile	string	Specifies an alternate text file to which the DBMS Server writes query information. The default filename is "iiprtqry.log". To enable this feature, use the set_ingres printqry option. If you omit a directory or path specification, the file is created in the current default directory.

Parameter	Data Type	Description
printgca	integer	Turns the printgca debugging feature on or off. As the application executes, printgca prints communications messages to the file "iiprintgca.log" in the current directory. Specify 1 to turn printgca on, 0 to turn printgca off.
gcafile	string	Specifies an alternate text file to which the DBMS Server writes GCA information. The default filename is "iiprtgca.log". To enable this feature, use the set_ingres printgca option. If you omit a directory or path specification, the file is created in the current default directory.
printtrace	integer	Enables or disables trapping of DBMS Server trace messages to a text file (iiprttrc.log, in the current directory). Specify 1 to enable trapping of trace output, 0 to disable trapping.
trcfile	string	Specifies an alternate text file to which the DBMS Server writes tracepoint information. The default filename is "iiprttrc.log". To enable this feature, use the set_ingres printtrace option. If you omit a directory or path specification, the file is created in the current default directory.
programquit	integer	Specifies whether the session aborts on the following errors: ▪ An application issues a query but is not connected to a database, or ▪ The DBMS Server fails, or ▪ Communications services fail. Specify 1 to abort on these conditions, 0 to continue.
savequery	integer	Enables or disables saving of the text of the last query issued. Specify 1 to enable, 0 to disable. To obtain the text of the last query, you must issue the inquire_ingres(*query*=querytext) statement. To determine whether saving is enabled, use the inquire_ingres(*status*=savequery) statement.
errorno	integer	Sets the value of the error return variable errorno.

Appendix A: Keywords

The following table lists Ingres keywords and indicates the contexts in which they are reserved. This list enables you to avoid assigning object names that conflict with reserved words.

Note: The keywords in this list do not necessarily correspond to supported features. Some words are reserved for future or internal use, and some words are reserved to provide backward compatibility with older features.

In the following table, the column headings have the following meanings:

ISQL	Interactive SQL. These keywords are reserved by the DBMS.
ESQL	Embedded SQL. These keywords are reserved by the SQL preprocessors.
IQUEL	Interactive QUEL. These keywords are reserved by the DBMS.
EQUEL	Embedded QUEL. These keywords are reserved by the QUEL preprocessors.
4GL	Ingres 4GL. These keywords are reserved in the context of SQL or QUEL in 4GL routines.

Note: The ESQL and EQUEL preprocessors also reserve forms statements. Forms statements are described in the *Forms-based Application Development Tools User Guide*.

Single Keywords

Keywords are listed in the following table:

	SQL			QUEL		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
abort	*	*	*	*	*	*
activate		*			*	
add	*	*	*			
addform		*			*	
after			*			*

	SQL			QUEL		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
all	*	*		*	*	
alter	*		*			
and	*	*		*	*	
any	*	*	*	*	*	
append				*	*	*
array			*			
as	*	*		*	*	*
asc	*		*			
at	*	*	*	*	*	*
authorization	*	*				
avg	*	*	*	*	*	
avgu		*		*	*	
before			*			*
begin		*	*	*		*
between	*	*	*			
breakdisplay		*			*	
by	*	*		*	*	*
byref	*		*			*
call		*	*		*	*
callframe			*			*
callproc	*		*			*
cascade	*	*				
check	*	*	*			
clear		*	*		*	*
clearrow		*	*		*	*
close	*	*		*		
column		*			*	
command		*			*	
comment	*		*			

	SQL			QUEL		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
commit	*	*				
connect		*				
constraint	*	*	*			
continue	*	*				
copy	*	*	*	*	*	*
count	*	*	*	*	*	
countu		*		*	*	
create	*	*	*	*	*	*
current	*	*				
current_user	*					
cursor	*	*				
datahandler		*				
declare	*	*	*			*
default	*	*	*			*
define	*			*		*
delete	*	*	*	*	*	*
deleterow		*	*		*	*
desc			*			
describe	*	*				
descriptor		*				
destroy				*	*	*
direct			*			*
disable	*		*			
disconnect		*				
display		*	*		*	*
distinct	*	*	*			
distribute				*		
do	*		*			*
down		*			*	

	SQL			QUEL		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
drop	*	*	*			
else	*		*			*
elseif	*		*			*
enable	*		*			
end	*	*	*	*	*	*
end-exec		*				
enddata		*			*	
enddisplay		*			*	
endfor	*					
endforms		*			*	
endif	*		*			*
endloop	*	*	*		*	*
endrepeat	*					
endretrieve					*	
endselect		*				
endwhile	*		*			*
escape	*	*				
exclude				*		
excluding	*			*		
execute	*	*		*		
exists	*	*	*			
exit			*		*	*
fetch	*	*				
field		*			*	
finalize		*			*	
first	*	*				
for	*	*	*	*	*	
foreign	*	*	*			
formdata		*			*	

	SQL			QUEL		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
forminit		*			*	
forms		*			*	
from	*	*	*	*	*	*
full	*	*	*			
get			*			
getform		*			*	
getoper		*			*	
getrow		*			*	
global	*	*	*			
goto		*				
grant	*	*	*			
granted	*	*	*			
group	*	*	*			
having	*	*	*			
help		*		*	*	
help_forms			*			*
help_frs		*			*	
helpfile		*	*		*	*
identified		*	*			
if	*		*			*
iimessage		*			*	
iiprintf		*			*	
iiprompt		*			*	
iistatement					*	
immediate	*	*	*			*
import	*					
in	*	*	*	*	*	
include		*		*		
index	*	*	*	*	*	*

	SQL			QUEL		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
indicator		*				
ingres					*	
initial_user		*				
initialize		*	*		*	*
inittable		*	*		*	*
inner	*	*	*			
inquire_equel					*	
inquire_forms			*			*
inquire_frs		*			*	
inquire_ingres		*	*		*	*
inquire_sql		*	*			
insert	*	*	*			
insertrow		*	*		*	*
integrity	*	*		*		*
into	*	*	*	*	*	*
is	*	*	*	*	*	*
join	*	*				
key	*	*	*			*
leave	*					
left	*	*	*			
level	*	*		*	*	
like	*	*				
loadtable		*	*		*	*
local	*					
max	*	*	*	*	*	
menuitem		*			*	
message	*	*	*		*	*
min	*	*	*	*	*	
mode			*			*

	SQL			QUEL		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
modify	*	*	*	*	*	*
module	*					
move				*		
natural	*	*				
next		*			*	
noecho			*			*
not	*	*		*	*	
notrim		*			*	
null	*	*	*		*	*
of	*	*	*	*	*	*
on	*	*		*	*	*
only				*		*
open	*	*		*		
option	*					
or	*	*		*	*	
order	*	*	*	*	*	*
out		*				
param					*	
permit	*	*		*		*
prepare	*	*				
preserve	*	*				
primary	*	*	*			
print		*		*	*	
printscreen		*	*		*	*
privileges	*					
procedure	*	*	*			*
prompt		*	*		*	*
public	*	*				
putform		*			*	

	SQL			QUEL		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
putoper		*			*	
putrow		*			*	
qualification			*			*
raise	*		*			
range				*	*	*
redisplay		*	*		*	*
references	*	*	*			
referencing	*		*			
register	*	*	*	*	*	*
relocate	*	*	*	*	*	*
remove	*	*	*		*	*
rename				*		
repeat	*	*	*		*	*
repeated		*	*			
replace				*	*	*
replicate				*		
restrict	*	*				
result	*	*				
resume		*	*		*	*
retrieve				*	*	*
return	*		*			*
revoke	*	*	*			
right	*	*	*			
role	*	*	*			
rollback	*	*	*			
row	*	*				
rows	*	*				
run			*			*
save	*	*	*	*	*	*

	SQL			QUEL		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
savepoint	*	*	*	*	*	*
schema	*	*				
screen		*	*		*	*
scroll		*	*		*	*
scrolldown		*			*	
scrollup		*			*	
section		*				
select	*	*	*			
session	*	*				
session_user		*				
set	*	*	*	*	*	*
set_4gl			*			*
set_equel					*	
set_forms			*			*
set_frs		*			*	
set_ingres		*	*		*	*
set_sql		*	*			
sleep		*	*		*	*
some	*	*	*			
sort				*	*	*
sql	*					
stop		*				
submenu		*			*	
substring	*	*				
sum	*	*	*	*	*	
sumu		*		*	*	
system			*			*
system_maintained	*	*		*	*	

	SQL			QUEL		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
system_user		*				
table	*	*				
tabledata		*			*	
temporary	*	*				
then	*	*	*			*
to	*	*		*	*	*
type			*			
union	*	*	*			
unique	*	*	*	*	*	*
unloadtable		*	*		*	*
until	*	*	*	*	*	*
up		*			*	
update	*	*	*	*		
user	*	*	*			
using	*	*				
validate		*	*		*	*
validrow		*	*		*	*
values	*	*	*			
view	*	*		*		*
when	*	*				
whenever		*				
where	*	*	*	*	*	*
while	*					*
with	*	*	*	*	*	*
work	*		*			

Double Keywords

Double keywords are listed in the following table:

Double Keyword	SQL			QUEL		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
after field			*			*
alter group	*	*	*			
alter location	*	*	*			
alter role	*	*	*			
alter security_audit	*	*	*			
alter table	*	*	*			
alter user	*	*	*			
array of			*			
before field			*			*
begin transaction	*	*	*	*	*	*
by user	*		*			
call on			*			
call procedure			*			
class of			*			
close cursor		*		*	*	
comment on	*	*	*			
connect to			*			
copy table			*			
create dbevent	*	*	*			
create group	*		*			
create integrity	*		*			
create link	*	*				
create location	*	*	*			
create permit	*		*			
create procedure			*			
create role	*	*	*			

Double Keyword	SQL			QUEL		
Reserved in:	ISQL	ESQL	4GL	IQUEL	EQUEL	4GL
create rule	*	*	*			
create security_alarm	*	*	*			
create synonym	*	*	*			
create user	*	*	*			
create view	*		*			
current installation			*			
define cursor				*		
declare cursor					*	
define integrity				*	*	*
define link					*	
define location				*		
define permit				*	*	*
define qry				*		*
define query				*		
define view				*	*	*
delete cursor				*	*	
destroy integrity		*		*	*	*
destroy link		*			*	
destroy permit		*		*	*	*
destroy table		*			*	
destroy view						*
direct connect		*	*		*	*
direct disconnect		*	*		*	*
direct execute		*				*
disable security_audit	*	*	*			
disconnect current			*			
display submenu			*			*
drop dbevent	*	*	*			
drop group	*		*			

Double Keyword	SQL			QUEL		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
drop integrity	*		*			
drop link	*	*	*			
drop location	*	*	*			
drop permit	*		*			
drop procedure			*			
drop role	*	*	*			
drop rule	*	*	*			
drop security_alarm	*	*	*			
drop synonym	*	*	*			
drop user	*	*	*			
drop view	*		*			
enable security_audit	*	*	*			
end transaction	*	*	*	*	*	*
exec sql		*				
execute immediate			*			
execute on			*			
execute procedure			*			
foreign key			*			
for deferred	*			*		
for direct	*			*		
for readonly	*			*		
for retrieve				*		
for update				*		
from group	*		*			
from role	*		*			
from user	*		*			
full join	*		*			
get data		*				
get dbevent		*	*			

Double Keyword	SQL			QUEL		
Reserved in:	ISQL	ESQL	4GL	IQUEL	EQUEL	4GL
global temporary			*			
help comment	*					
help integrity		*			*	
help permit		*			*	
help table	*					
help view		*			*	
identified by			*			
inner join	*		*			
is null				*		
left join	*		*			
modify table			*			
not like	*		*			*
not null				*		
on commit	*	*	*			
on current	*					
on database	*		*			
on dbevent	*		*			
on location	*		*			
on procedure	*					
only where				*		
open cursor		*		*	*	
order by				*		
primary key			*			
procedure returning			*			*
put data		*				
raise dbevent	*	*	*			
raise error	*					
register dbevent	*	*	*			
register table						*

Double Keyword	SQL			QUEL		
Reserved in:	ISQL	ESQL	4GL	IQUEL	EQUEL	4GL
register view			*			*
remove dbevent	*	*	*			
remove table						*
remove view			*			*
replace cursor		*		*	*	*
resume entry			*			*
resume menu			*			*
resume next			*			*
retrieve cursor		*		*	*	
right join	*		*			
run submenu			*			*
session group			*			
session role			*			
session user			*			
set aggregate	*			*		
set autocommit	*			*		
set cpufactor	*			*		
set date_format	*			*		
set ddl_concurrency	*					
set decimal	*			*		
set io_trace	*			*		
set jcpufactor				*		
set joinop	*			*		
set journaling	*			*		
set lock_trace	*			*		
set lockmode	*			*		
set logdbevents	*					
set log_trace	*			*		
set logging	*			*		

Double Keyword	SQL			QUEL		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
set maxcost	*			*		
set maxcpu	*			*		
set maxio	*			*		
set maxpage	*			*		
set maxquery	*			*		
set maxrow	*			*		
set money_format	*			*		
set money_prec	*			*		
set noio_trace	*			*		
set nojoinop	*			*		
set nojournaling	*			*		
set nolock_trace	*			*		
set nologdbevents	*					
set nolog_trace	*			*		
set nologging	*			*		
set nomaxcost	*			*		
set nomaxcpu	*			*		
set nomaxio	*			*		
set nomaxpage	*			*		
set nomaxquery	*			*		
set nomaxrow	*			*		
set nooptimizeonly	*			*		
set noprintdbevents	*					
set noprintqry	*			*		
set noprintrules	*					
set noqep	*			*		
set norules	*					
set nosql				*		
set nostatistics	*			*		

Double Keyword	SQL			QUEL		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
set notrace	*			*		
set optimizeonly	*			*		
set printdbevents	*					
set printqry	*			*		
set qep	*			*		
set result_structure	*			*		
set ret_into	*			*		
set rules	*					
set session	*			*		
set sql				*		
set statistics	*			*		
set trace	*			*		
set work	*					
system user			*			
to group	*		*			
to role	*		*			
to user	*	*				
user authorization			*			
with null				*		
with short_remark	*					

ISO SQL

The following keywords are ISO standard keywords that are not currently reserved in Ingres/SQL or Ingres/Embedded SQL. Use these as reserved words to ensure compatibility with other implementations of SQL.

absolute	except	output
action	exception	overlaps
allocate	exec	pad

alter
are
asc
assertion
bit
bit_length
both
cascaded
case
cast
catalog
char
character
char_length
character_length
coalesce
collate
collation
connection
constraints
convert
corresponding
cross
current_date
current_time
current_timestamp
date
day
deallocate
dec
decimal
deferrable
deferred
external
extract
false
first
float
found
get
go
hour
identity
initially
input
insensitive
int
integer
intersects
interval
isolation
language
last
leading
level
lower
match
minute
module
month
names
national
nchar
no
nullif
numeric
partial
position
precision
prior
privileges
read
real
relative
second
size
smallint
space
sql
sqlcode
sqlerror
substring
then
time
timestamp
timezone_hour
timezone_minute
trailing
transaction
translate
translation
trim
true
unknown
upper
usage
value
varchar
varying

desc	octet_length	work
diagnostics	only	write
domain	option	year
double	nchar	zone
else	outer	

Appendix B: Terminal Monitor

The Terminal Monitor allows you to interactively enter, edit, and execute individual queries or files containing several queries. This appendix describes the commands you use to perform queries from the Terminal Monitor.

You can also perform interactive queries using QBF (Query-By-Forms). For information about QBF, see *Character-based Querying and Reporting Tools User Guide*.

Accessing the Terminal Monitor

To invoke the Terminal Monitor, type quel at the operating system prompt. For information about the quel command line flags, see the quel command description in the *Command Reference Guide.*

Query Buffer

Once you have entered the Terminal Monitor, each query that you type is placed in a query buffer. The queries are executed when you type the execution command (\go). The results of your query are displayed on your terminal. For example,

```
retrieve (employee.name) where employee.city = "San
Francisco"\g
```

In addition to entering queries, you can:

- Edit the queries
- Print the queries
- Write the queries to a file

After a ****go command, the query buffer is cleared if another query is typed in, unless a command that affects the query buffer is typed first. Commands that retain the query buffer contents are:

\append or \a
\edit or \e
\print or \p
\bell
\nobell
\eval or \v
\[no]macro

For example, typing

```
help parts
\go
print parts
```

results in the query buffer containing

```
print parts
```

whereas, typing

```
help parts
\go
\print
print parts
```

results in the query buffer containing

```
help parts
print parts
```

You can override this feature by executing the \append command before you execute the \go command or by specifying the -a flag when you issue the quel command to begin your session.

Terminal Monitor Commands

The Terminal Monitor commands are the commands that you use to manipulate the contents of the query buffer or your environment. Unlike the QUEL statements that you type into the Terminal Monitor, these commands are executed as soon as you press the Return key.

You must precede all of the Terminal Monitor commands with a backslash (\). If you want to enter a backslash literally, you must precede it with another backslash and enclose the pair in quotes. For example, the following statement inserts a backslash into the test table:

```
append to testtable (name="James T. Smith\\n")\g
```

Some Terminal Monitor commands accept a file name as an argument. These commands must appear alone on a single line; the Terminal Monitor interprets all characters appearing on the line after such commands as a file name. Those Terminal Monitor commands that do not accept arguments can be stacked on a single line. For example,

```
\date\go\date
```

returns the date and time before and after execution of the current query buffer.

The following table lists the Terminal Monitor commands:

Command	Description
\r or \reset	Erases the entire query (reset the query buffer). The former contents of the buffer are lost and cannot be retrieved.
\p or \print	Prints the current query. The contents of the buffer are printed on the user's terminal.
\l or \list	Prints the current query as it appears after macro processing. All side effects of macro processing (such as macro definition) occurs. \list clears the query buffer; use \eval to process macros without clearing the query buffer.
\eval or \v	Processes macros in the query buffer and replaces the query buffer with the result. Similar to \list, but the result is placed in the query buffer instead of being displayed on the terminal.
\e or \ed or \edit or \editor [*filename*]	Invokes a text editor (designated by the startup file). Use the appropriate editor command to return to the Terminal Monitor. If no file name is given, the current contents of the query buffer are sent to the editor, and upon return, the query buffer is replaced with the edited query. If a file name is given, the query buffer is written to that file. On exit from the editor, the file contains the edited query, but the query buffer remains unchanged.
\g or \go	Processes the current query. The contents of the buffer are transmitted to the DBMS Server and run.
\a or \append	Appends to the query buffer. Typing \append after completion of a query overrides the auto-clear feature and guarantees that the query buffer is not reset until it is executed again.
\time or \date	Prints out the current time and date.
\s or \sh or \shell	Escape to the operating system.

Command	Description
UNIX	Type Ctrl-D to return to the Terminal Monitor.
VMS	Type logout to return to the Terminal Monitor.
\q or \quit	Exits the Terminal Monitor.
\cd or \chdir *dir_name*	Changes the working directory of the monitor to the specified directory.
\i or \include or \read *filename*	Reads the named file into the query buffer. Backslash characters in the file are processed as they are read.
\w or \write *filename*	Writes the contents of the query buffer to the specified file.
\branch	Transfers control within an \include file. See Branching.
\script [*filename*]	Writes/stops writing the subsequent QUEL statements and their results to the specified file. If no file name is supplied with the \script command, output is logged to a file called "script.ing" in the current directory. The \script command toggles between logging and not logging your session to a file. If you supply a *filename* on the \script command that terminates logging to a file, the *filename* is ignored. You can use this command to save result tables from QUEL statements for output. The \script command in no way impedes the terminal output of your session.
\bell and \nobell	Tells the Terminal Monitor to include (\bell) or not to include (\nobell) a bell (Ctrl-G) with the continue or go prompt. The default is \nobell.
\mark	Sets a label for \branch.
\macro and \nomacro	Enables/disables macro definition. The default is \nomacro.
\continue and \nocontinue	Tells the Terminal Monitor to continue statement processing on error or not to continue (nocontinue). In either case, the error message is displayed. You can abbreviate the command to \co (\continue) or \noco (\nocontinue). The default action is to continue. You can use this command to change that behavior. You can also change the default by setting II_TM_ON_ERROR. For information about II_TM_ON_ERROR, see the *System Administrator Guide*.

Messages and Prompts

The Terminal Monitor has a variety of messages to keep you informed of its status and that of the query buffer.

When you log in, the Terminal Monitor prints a login message that tells the version number and the login time. Following that message, the dayfile appears.

When the Terminal Monitor is ready to accept input and the query buffer is empty, the message go appears. The message continue appears instead, if there is something in the query buffer.

The prompt >editor indicates that you are in the text editor.

Character Input and Output

When you input non-printable ASCII characters through the Terminal Monitor, the Terminal Monitor maps these characters to blanks. Whenever this occurs, the Terminal Monitor displays the following message:

```
Non-printing character nnn converted to blank
```

where *nnn* is replaced with the actual character.

For example, if you enter the following statement:

```
append to test (col1 = "^La")\g
```

the Terminal Monitor converts the ^L to a blank before sending it to the DBMS Server and displays the message described above.

On output, if the data type is char or varchar, any binary data are shown as octal numbers (\000, \035, etc.). Any backslashes in data of the char or varchar type are displayed as double backslashes. For example, if you append the following to the "test" table:

```
append to test (col1 = "\\a")\g
```

when you retrieve that value, you see:

```
\\a
```

but what is actually stored in the table is:

```
\a
```

Help

When you are working in the Terminal Monitor, you can obtain on-line help using the help statement. This statement provides information about a variety of QUEL statements and features. For details, see Help in the "QUEL and EQUEL Statements" chapter.

Branching

The \branch and \mark commands permit arbitrary branching within an \include file. \mark must be followed with a label. Follow \branch with a label to indicate unconditional branch. To indicate conditional branch, follow \branch with an expression preceded by a question mark (?) and followed by a label. The branch is taken if the expression is greater than zero. For example, consider the following Terminal Monitor command:

```
\branch ?{ifsame;@{read Enter data:};a;1;0}=1 valueok
```

This command relies heavily on Terminal Monitor macros. Reading outward from the inside, the {read} macro writes "Enter data:" on the screen and accepts input from the terminal. {read} is preceded with an "@" sign, because it must be pre-scanned in this expression. See Parameter Prescan.

The result of the {read} macro, that is, what is typed at the terminal, becomes the first string in an {ifsame} macro. The {ifsame} macro compares what is entered to "a". If "a" is entered, the value of {ifsame} is 1. If anything other than "a" is entered, the value of {ifsame} is 0. If the result of this nesting of system macros is 1, the Terminal Monitor branches to label "valueok" (that is, if the letter "a" is entered at the terminal).

The expressions usable in \branch statements are somewhat restricted. The following operators are defined in the usual way: +, -, *, /, >, >=, <, <=, != and =. The left unary operator ! can be used to indicate logical negation. There cannot be spaces in the expression because a space terminates the expression.

Restrictions

VMS

Ctrl-Y and Ctrl-C must not be used while you are escaped to an editor or VMS. VMS does not signal these events to the initiating process. The only exception is if the editor catches Ctrl-C for its own use.

Terminal Monitor Macros

The Terminal Monitor macro facility enables you to tailor the QUEL language to your needs. The macro facility allows strings of text to be removed from the query buffer and replaced with other text. Built-in macros allow you to change environment variables. To enable the macro feature you must issue the \macro command within the Terminal Monitor. By default the Terminal Monitor macro facility is disabled.

Basic Concepts

All macros are defined as two parts: the *template* part and the *replacement* part. The template part establishes a symbol that, when encountered in the Terminal Monitor workspace, signals the Terminal Monitor to invoke the symbol's definition. When a macro is encountered, the template part is removed and replaced with the replacement part.

For example, the template ret, when read by the Terminal Monitor, causes the corresponding definition of ret to be invoked. If the replacement part of the ret macro is retrieve, all instances of the word ret in the query text are replaced with the word retrieve. For example: part and the *replacement* part. The template part is replaced at execution time by the replacement part. For example, the following macro definition specifies the macro template ret is to be replaced by the QUEL retrieve statement:

```
{define; ret; retrieve}
```

After you define the ret macro, QUEL replaces the macro ret with retrieve. For example, if you issue the following statement:

```
ret (p.all)
```

the Terminal Monitor expands the ret macro as follows:

```
retrieve (p.all)
```

Macros accept parameters, specified as single letters (or digits) preceded by a dollar sign, such as $2 or $k. For example, the template get $1 enables the get macro to accept a single parameter. If the get macro is defined as:

```
retrieve (p.all) where p.pnum = $1
```

typing **get 35** retrieves all information about part number 35.

Defining Macros

To create your own macros, use the Terminal Monitor {define} macro. The basic form of this command is:

```
{define; $t; $r}
```

where $t and $r are the template and replacement parts of the macro, respectively.

The Terminal Monitor contains a macro processor that substitutes the replacement part of the macro for the template part.

For example, the following macro enables you to shorten range statements:

```
{define; rg $v $r; range of $v is $r}
```

This macro causes the word rg, followed by the next two words, to be removed and replaced by the words range of, followed by the first word that followed rg, followed by the word is, followed by the second word that followed rg:

```
rg p parts
```

is expanded to:

```
range of p is parts
```

Macro Evaluation

When you enter a define statement at your terminal, it is not processed immediately; macro processing occurs when the query buffer is evaluated. The Terminal Monitor commands \go, \list and \eval evaluate the workspace. \go sends the results to the database for execution, \list prints them on your terminal, and \eval puts the result back into the workspace.

The usual process for defining macros requires that you type the following commands:

```
{define . . . }
\eval
\reset
```

The \reset command assures that the workspace is emptied before you enter the next query.

You can use the \eval and \list commands to test a macro invocation before executing it explicitly (with the \g). For example, to test the rg macro above, type:

```
rg e emp
\l
```

The Terminal Monitor types:

```
range of e is emp
```

The range statement is not executed.

Similarly, the \eval (or \v for short) command replaces the macro version with the expanded range statement although the command is not executed. In the case of macro expansion with \eval or \v, to execute the range statement, type:

```
\g
```

Quoting

Sometimes text strings must be passed through the macro processor without being processed. In such cases the grave accent mark and apostrophe (‘ and ’) must surround the literal text. For example, to pass the word ret through without converting it to retrieve type:

```
‘ret’
```

If you want to enter more than one word for substitution into a macro parameter, you must quote the parameter. For example, if you define a macro:

```
{define; r $1 $2; retrieve ($1) where $2}
```

and invoke it with the query:

```
r ‘p.name, weight = p.qoh*p.stk’ ‘p.cnt10’
\l
```

the query is evaluated as:

```
retrieve (p.name, weight = p.qoh*p.stk) where p.cnt10
```

Backslashes

To disallow the special meaning of characters, precede them with the backslash character (\). For example, an accent mark can be included in a quoted parameter by preceding it with a backslash:

```
here is a \’quoted\’ string
```

evaluates to:

```
here is a ’quoted’ string
```

To enter a real backslash, use two backslashes.

To continue a macro definition to another line, terminate the line with a backslash. For example:

```
{define;~get~$n;~retrieve~(e.all)~~\
where~e.name~ = "$n"}
```

You must enter two blanks before the backslash that continues the macro definition to the second line, and you must not enter a blank after that backslash. In other words, to continue the macro definition to the next line, enter four keystrokes: blank, blank, backslash, RETURN.

More on Parameters

Parameters need not be limited to the word that follows. For example, in the template descriptor for {define},

```
{define; $t; $r}
```

the $t parameter ends at the first semicolon, and the $r parameter ends at the first right curly brace. In general, the character that follows the parameter specifier terminates the parameter. If this character is a space, tab, newline or the end of the template, one word or one string appropriately surrounded (‘ and ’) is collected.

There is one important exception to this general rule: because system macros (described below) are always surrounded by curly braces, the macro processor requires them to be properly nested. Thus, in the macro definition:

```
{define; x; {type enter dat-}}
```

the first right curly brace closes the type rather than the define.

System Macros

The macro processor contains several other macros built into it. In the following descriptions, some of the parameter specifiers are marked with two dollar signs rather than one. This feature is discussed in the Parameter Prescan section.

System Macro	Description
{define; *$$t; $$r*}	Defines a macro as discussed above. Special processing, which is discussed in a later section, occurs on the template part.

System Macro	Description
{rawdefine; *$$t; $$r*}	Another form of {define}, where the special processing does not take place. This is rarely used but is seen when listing macros with the \l command, because the DBMS Server converts all {define} statements into their corresponding {rawdefine} form.
{remove; *$$n*}	Removes all macros beginning with name *$n.* For example, typing: {define; get part $n; . . . } {define; get emp $x; . . . } defines two macros that start with "get." Typing: {remove; get} removes both of the get macros. Typing {remove; get part} removes only the first macro.
{type *$$s*}	Types *$s* onto the terminal.
{read *$$s*}	Types *$s* and reads a line from the terminal. The typed line acts as the replacement text for the macro.
{readcount}	Contains the number of characters read in the most recent {read} or {readdefine}. A Ctrl-Z (VMS) or Ctrl-D (UNIX) (end of file) becomes -1, a single newline becomes zero, and so forth, so that the number accurately reflects printing characters.
{readdefine; *$$n; $$s*}	Also types *$s* and reads a line, but it further creates a macro called *$n*, which is set to the line entered at the terminal. This lets you set aside a line for further processing. The replacement text for {readdefine} is the count of the number of characters in the line. {readcount} is also defined with this number.
{ifsame; *$$a; $$b; $t; $f*}	Compares the strings *$a* and *$b.* If they match precisely, the replacement text becomes *$t*; otherwise it becomes *$f*.
{ifeq; *$$a; $$b; $t; $f*}	Similar to {ifsame}, but the comparison is numeric.
{ifgt; *$$a; $$b; $t; $f*}	Like {ifeq}, but the test is for *$a* strictly greater than *$b.*

System Macro	Description
{substr; *$$b; $$e; $$s*}	Returns the part of string *$s* between character positions *$b* and *$e*, numbered from one. If *$b* or *$e* is out of range, it is moved in range as much as possible.
{dump; *$$n*}	Returns the value of the macro (or macros) that match *$n,* using the same algorithm as remove. The {dump} macro produces a listing of macros in {rawdefine} form. Dump without arguments dumps all macros. This macro works in conjunction with the \eval statement.

Special Characters

Certain characters are used internally; normally you do not even see them. But they can appear in the output of a {dump} command and can sometimes be used to create very intricate macros. **\|** matches any number of spaces, tabs or newlines. It even matches zero, but only between words, as can occur with punctuation. For example, **\|** matches the spot between the last character of a word and a comma following it.

Character	Description
|	Matches any number of spaces, tabs or newlines.
\^	Matches exactly one space, tab or newline.
\&	Matches exactly zero spaces, tabs or newlines, but only between words.

Special {define} Processing

When you define a macro using {define}, special processing takes place. In {define}, all sequences of spaces, tabs and newlines in the template, as well as all "non-spaces" between words are turned into a single **\|** character. If the template ends with a parameter, the \& character is added at the end.

If you want to match a real tab or newline, you can use \t or \n, respectively. For example, a macro that reads an entire line and uses it as the name of an employee is defined with:

```
{define; get $n\n; \~~~~ret (e.all) where e.name =
‘‘n’’}
```

This macro can be used by typing:

```
get *Stan*
```

to get all information about everyone with a name that included "Stan." You can nest the ret macro inside the get macro as long as ret is previously defined.

Parameter Prescan

Sometimes it is useful to "macro process" a parameter before using it in the replacement part. This is particularly important when using certain built-in macros.

For prescan to occur, the parameter must be specified in the template with two dollar signs instead of one, and the actual parameter must begin with an "at" sign (@), which is stripped off.

An example of prescan follows:

```
{define; typeit $$s; {type $s}}
{define; line; this is text}
```

For example, the string:

```
typeit line
```

is replaced by:

```
line
```

However, the entry:

```
typeit @line
```

results in:

```
this is text
```

For another example of the use of prescan, see Special Macros.

Special Macros

The following special macros are used by the Terminal Monitor to control the environment and to return results to the user:

Macro	Description
{begintrap}	Executed at the beginning of a query.
{endtrap}	Executed after the body of a query is passed to the DBMS Server.
{continuetrap}	Executed after the query completes. The difference between this and {endtrap} is that {endtrap} occurs after the query is submitted, but before the query executes. {continuetrap}, on the other hand, is executed after the query executes.
{editor}	Defines the on-line editor to use in the \edit command.
Windows	To change the default vi editor to the ed editor, enter: `{define;{editor};/bin/ed}` which invokes the ed editor in response to the \e command.
VMS	To change the default (EDT) editor to the SOS screen editor, enter: `{define;{editor};edit/sos}` which invokes the SOS editor in response to the \e command.
{shell}	Defines the pathname of a shell to use in the \shell command.
{tuplecount}	Is set after every query (but before {continuetrap} is sprung) to be the count of the number of rows that satisfied the qualification of the query in a retrieve, or the number of rows changed in an update. It is not set for some utility functions (such as define view). If multiple queries are run at once, it is set to the number of rows that satisfied the last query run.

For example, to print out the number of rows affected automatically after each query, enter the following commands:

```
{define;~{begintrap};~{remove;~{tuplecount}}}
{define;~{continuetrap};
^{ifsame;@{tuplecount};~{tuplecount};;~~\
^^{type~@{tuplecount}~tuples~touched}}}
```

Appendix C: Calling Ingres Tools from Embedded QUEL

Using the call statement, you can call Ingres tools or execute operating system commands from within an embedded QUEL program. For additional information about the call statement, see Call in the "QUEL and EQUEL Statements" chapter.

To call an Ingres tool, the syntax is:

call *subsystem* **(database =** *dbname* {**,** *parameter* **=** *value*}**);**

To call the operating system, the syntax is:

call system (**command =** *command_string*)

You can specify parameters using (quoted or unquoted) strings or host string variables. If there is no value for a particular parameter, use an empty string ("").

Examples:

```
## call qbf (database = "empdb", table = "employee")
## call rbf (database = "empdb",
## flags = "-s -mblock emptable")
## call report (database = :dbvar, name = :namevar,
## mode = :modevar)
## call system (command = "mail")
```

In the third example, "dbvar", "namevar", and "modevar" are host language string variables.

Ingres Tools and Parameters

This section discusses the specific parameters and flags you can specify when calling an Ingres tool.

When you call an Ingres tool, you can use the flags parameter to specify the values of flags. You must separate the flags using spaces.

If a parameter does not accept an argument, you must specify a dummy argument using empty quotes. For example, the silent parameter of the report command does not accept an argument:

```
## call report (database = "mydb", name = "employee",
## silent = " ")
```

Report

The report command, which invokes the Report-Writer, accepts the following parameters:

Parameter	Description
file	Equivalent to the -f flag. Directs the formatted report to the specified file for output.
silent	Equivalent to the -s flag. Suppresses status messages.
report	Equivalent to the -r flag. Indicates that a report, rather than a table, is being specified. The name of the report is the value for this parameter.
style	Equivalent to the -m flag. Indicates that a table, rather than a report, is being specified. Optional values for this parameter are column, wrap and block. The name of the table is given as the value for the name parameter.
name	Name of a table or view in the database for which a default report is to be formatted.

param	The list of parameters for the report. Each element in the list must be of the form name = value
	
	Name/value combinations must be separated by blanks or tabs. The entire list must be enclosed within quotes. In addition, if name is a character report parameter, value must be enclosed in quotes. (Values of numeric report parameters must not, however, be quoted.) The inner quotes that surround value must be dereferenced according to host language rules so that they can be passed through to the report command. For example, assume that you want to call the Report-Writer from within embedded QUEL with the equivalent of this system-level command:
	
	`report newdb -r myrpt (bin="f01"`
	`wstation="u1"`
	`type=12 sect=11)`

Parameter	Description

	The variables "bin" and "wstation" are character parameters. The variables "type" and "sect" are numeric parameters.
	
	You can specify the parameters using a host string variable. For example:
	
	`## call report`
	`(database = "newdb",`
	`report = "myrpt",`
	`param = :parmvar)`
	
	`where "parmvar" contains`
	
	`bin="f01" wstation="u1"`
	`type=12 sect=11`
	
	Double quotes must surround the constant string values within the variable. If your host language requires the dereferencing of double quotes, be sure to do so, according to the rules of your host language.

forcerep	Equivalent to the -h flag. Report-Writer outputs headers and footers, even if no data is found for the report.
formfeed	Equivalent to the +b flag. Report-Writer forces formfeeds at page breaks, overriding any settings in the report formatting commands.
noformfeed	Equivalent to the -b flag. Report-Writer suppresses formfeeds, overriding any settings in the report formatting commands.
pagelength	Equivalent to the -v flag. Sets the page length, in lines, for the report, overriding any .PL commands in the report.
brkfmt	Equivalent to the +t flag (default). If specified, breaks and calculations for dates and numbers are based on the displayed data, rather than the internal database values.
nobrkfmt	Equivalent to the -t flag. If specified, breaks and calculations for dates and numbers are based on the internal database values, rather than the displayed values.
flags	Can be used for any flags on the command line. Distinct flags must be separated by a blank.

Sreport

The sreport command accepts the following parameters:

Parameter	Description
file	Name of a text file containing report formatting commands for one or more reports.
silent	Equivalent to the -s flag. Suppresses status messages.
flags	Can be used for any flags on the command line. Distinct flags must be separated by a blank.

RBF

The rbf command accepts the following parameters:

Parameter	Description
silent	Equivalent to the -s flag. Suppresses status messages.
report	Equivalent to the -r flag. Indicates that a report, rather than a table, is being specified. The name of the report is the value for this parameter.
style	Equivalent to the -m flag. Indicates that a table, rather than a report, is being specified. Optional values for this parameter are column, wrap and block The name of the table is given as the value for the table parameter.
table	The name of a table or view for which a default report is to be formatted.
emptycat	Equivalent to -e flag. If set, the Catalog form is displayed empty, and the user can enter names directly.
flags	Can be used for any flags on the command line. Distinct flags must be separated by a blank.

QBF

The qbf command accepts the following parameters:

Parameter	Description
qbfname	Equivalent to the -f flag. Invoke QBF using the specified qbfname. If the name is blank, start at Catalogs frame for qbfnames.
joindef	Equivalent to the -j flag. Invoke QBF using the specified JoinDef. If the name is blank, start at Catalogs frame for JoinDefs.
tblfld	Equivalent to the -t flag. Invoke QBF on the specified table, using a table field format to display the data. If the name is blank, start at Catalogs frame for tables.
lookup	Equivalent to the -l flag. Invoke QBF using the specified name. QBF looks up the name in the following order: qbfname, JoinDefname, table name.
silent	Equivalent to the -s flag. Suppresses verbose messages.
mode	Equivalent to the -m flag. Enter QBF directly in the specified mode. Possible values for this parameter are retrieve, append, update or all.
table	Name of the table on which QBF is being invoked. This parameter must be omitted if one of the joindef, qbfname, tblfld or lookup parameters has been used.
emptycat	Equivalent to -e flag. If set, catalogs are displayed empty, and the user can enter names directly.
flags	Can be used for any flags on the command line. Distinct flags must be separated by a blank.

Vifred

The vifred command accepts the following parameters:

Parameter	Description
form	Equivalent to the -f flag. Invoke VIFRED on the specified form.
table	Equivalent to the -t flag. Invoke VIFRED with a default form for the specified table.

Parameter	Description
joindef	Equivalent to the -j flag. Invoke VIFRED with a default form for the specified JoinDef.
emptycat	Equivalent to -e flag. If set, an empty Catalogs form is displayed, and the user can enter names directly.
flags	Can be used for any flags on the command line. Distinct flags must be separated by a blank.

ABF

The abf command accepts the following parameters:

Parameter	Description
application	Name of the application.
flags	Can be used for any flags on the command line. Distinct flags must be separated by a blank.

QUEL

The quel command to call the Terminal Monitor accepts the following parameter:

Parameter	Description
flags	Specifies command line flags. Flags must be separated by a blank.

IQUEL

The iquel command accepts the following parameter:

Parameter	Description
flags	Specifies command line flags. Flags must be separated by a blank.

Ingmenu

The ingmenu command to invoke Ingres Menu accepts the following parameter:

Parameter	Description
flags	Specifies command line flags. Flags must be separated by a blank.

System

The system command accepts the following parameter:

Parameter	Description
command	Executes the operating system level command specified by *command_string*. If *command_string* is null, empty, or blank, and transfers the user to the operating system.

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y