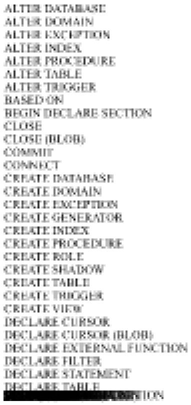
Table of Contents

Statement list

This chapter describes the following SQL statements:



ALTER DATABASE
ALTER DOMAIN
ALTER EXCEPTION
ALTER INDEX
ALTER PROCEDURE
ALTER TABLE
ALTER TRIGGER
BASED ON
BEGIN DECLARE SECTION
CLOSE
CLOSE (BLOB)
COMMIT
CONNECT
CREATE DATABASE
CREATE DOMAIN
CREATE EXCEPTION
CREATE GENERATOR
CREATE INDEX
CREATE PROCEDURE
CREATE ROLE
CREATE SHADOW
CREATE TABLE
CREATE TRIGGER
CREATE VIEW
DECLARE CURSOR
DECLARE CURSOR (BLOB)
DECLARE EXTERNAL FUNCTION
DECLARE FILTER
DECLARE STATEMENT
DECLARE TABLE
[redacted]ION

ALTER DATABASE

Adds secondary files to the current database. Available in SQL, DSQL, and isql.

Syntax

```
ALTER {DATABASE | SCHEMA}
   ADD <add_clause>;

<add_clause> = FILE 'filespec' [<fileinfo>] [<add_clause>]

<fileinfo> = LENGTH [=] int [PAGE[S]]
   | STARTING [AT [PAGE]] int [<fileinfo>]
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
SCHEMA	Alternative keyword for DATABASE
ADD FILE '*filespec*'	Adds one or more secondary files to receive database pages after the primary file is filled; for a remote database, associate secondary files with the same node
LENGTH [=] *int* [PAGE[S]]	Specifies the range of pages for a secondary file by providing the number of pages in each file
STARTING [AT [PAGE]] *int*	Specifies a range of pages for a secondary file by providing the starting page number

Description ALTER DATABASE adds secondary files to an existing database. Secondary files permit databases to spread across storage devices, but they must remain on the same node as the primary database file. A database can be altered by its creator, the SYSDBA user, and any users with operating system root privileges.

ALTER DATABASE requires exclusive access to the database.

Note InterBase dynamically expands the last file in a database as needed until it reaches the 4GB limit. You should be aware that specifying a LENGTH for such files has no effect.

You cannot use ALTER DATABASE to split an existing database file. For example, if your existing database is 80,000 pages long and you add a secondary file STARTING AT 50000, InterBase starts the new database file at page 80,001.

Tip To split an existing database file into smaller files, back it up and restore it. When you restore a database, you are free to specify secondary file sizes at will, without reference to the number and size of the original files.

Example The following isql statement adds two secondary files to an existing database. The command creates a secondary database file called employee2.gdb that is 10,000 pages long and another called employee3.gdb. interBase starts using employee2.gdb only when the primary file reaches 10,000 pages.

```
ALTER DATABASE
   ADD FILE 'employee2.gdb'
   STARTING AT PAGE 10001 LENGTH 10000
   ADD FILE 'employee3.gdb';
```

See Also CREATE DATABASE, DROP DATABASE

See the *Data Definition Guide* for more information about multi-file databases and the *Operations Guide* for more information about exclusive database access.

ALTER DOMAIN

Changes a domain definition. Available in SQL, DSQL, and isql.

```
ALTER DOMAIN name {
   SET DEFAULT {literal | NULL | USER}
   | DROP DEFAULT
   | ADD [CONSTRAINT] CHECK (<dom_search_condition>)
   | DROP CONSTRAINT
   | new_col_name
   | TYPE datatype};

<dom_search_condition> = {
   VALUE <operator> <val>
   | VALUE [NOT] BETWEEN <val> AND <val>
   | VALUE [NOT] LIKE <val> [ESCAPE <val>]
   | VALUE [NOT] IN (<val> [, <val> …])
   | VALUE IS [NOT] NULL
   | VALUE [NOT] CONTAINING <val>
   | VALUE [NOT] STARTING [WITH] <val>
   | (<dom_search_condition>)
   | NOT <dom_search_condition>
   | <dom_search_condition> OR <dom_search_condition>
   | <dom_search_condition> AND <dom_search_condition>
   }

<operator> = {= | < | > | <= | >= | !< | !> | <> | !=}
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
name	Name of an existing domain
SET DEFAULT	Specifies a default column value that is entered when no other entry is made. Values: ▪ *literal*—Inserts a specified string, numeric value, or date value ▪ NULL—Enters a NULL value ▪ USER—Enters the user name of the current user; column must be of compatible text type to use the default ▪ Defaults set at column level override defaults set at the domain level
DROP DEFAULT	Drops an existing default
ADD [CONSTRAINT] CHECK *dom_search_condition*	Adds a CHECK constraint to the domain definition; a domain definition can include only one CHECK constraint
DROP CONSTRAINT	Drops CHECK constraint from the domain definition

new_col_name	Changes the domain name
TYPE data_type	Changes the domain datatype

Description ALTER DOMAIN changes any aspect of an existing domain except its NOT NULL setting. Changes made to a domain definition affect all column definitions based on the domain that have not been overridden at the table level.

Note To change a datatype or NOT NULL setting of a domain, drop the domain and recreate it with the desired combination of features.

A domain can be altered by its creator, the SYSDBA user, and any users with operating system root privileges.

Example The following isql statements create a domain that must have a value > 1,000, then alter it by setting a default of 9,999:

```
CREATE DOMAIN CUSTNO
   AS INTEGER
   CHECK (VALUE > 1000);
ALTER DOMAIN CUSTNO SET DEFAULT 9999;
```

See Also CREATE DOMAIN, CREATE TABLE, DROP DOMAIN

For a complete discussion of creating domains, and using them to create column definitions, see the *Data Definition Guide*.

ALTER EXCEPTION

Changes the message associated with an existing exception. Available in DSQL and isql.

Syntax

```
ALTER EXCEPTION name 'message'
```

Argument	Description
name	Name of an existing exception message
'message'	Quoted string containing ASCII values

Description ALTER EXCEPTION changes the text of an exception error message.

An exception can be altered by its creator, the SYSDBA user, and any users with operating system root privileges.

Example This isql statement alters the message of an exception:

```
ALTER EXCEPTION CUSTOMER_CHECK 'Hold shipment for customer
   remittance.';
```

See Also ALTER PROCEDURE, ALTER TRIGGER, CREATE EXCEPTION, CREATE PROCEDURE, CREATE TRIGGER, DROP EXCEPTION

For more information on creating, raising, and handling exceptions, see the *Data Definition Guide*.

ALTER INDEX

Activates or deactivates an index. Available in SQL, DSQL, and isql.

Syntax

```
ALTER INDEX name {ACTIVE | INACTIVE};
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name of an existing index
ACTIVE	Changes an INACTIVE index to an ACTIVE one
INACTIVE	Changes an ACTIVE index to an INACTIVE one

Description ALTER INDEX makes an inactive index available for use, or disables the use of an active index. Deactivating and reactivating an index is useful when changes in the distribution of indexed data cause the index to become unbalanced.

Before inserting or updating a large number of rows, deactivate a table's indexes to avoid altering the index incrementally. When finished, reactivate the index. Reactivating a deactivated index rebuilds and rebalances an index.

If an index is in use, ALTER INDEX does not take effect until the index is no longer in use.

ALTER INDEX fails and returns an error if the index is defined for a UNIQUE, PRIMARY KEY, or FOREIGN KEY constraint. To alter such an index, use DROP INDEX to delete the index, then recreate it with CREATE INDEX.

An index can be altered by its creator, the SYSDBA user, and any users with operating system root privileges.

Note To add or drop index columns or keys, use DROP INDEX to delete the index, then recreate it with CREATE INDEX.

Example The following isql statements deactivate and reactivate an index to rebuild it:

```
ALTER INDEX BUDGETX INACTIVE;
ALTER INDEX BUDGETX ACTIVE;
```

See Also ALTER TABLE, CREATE INDEX, DROP INDEX, SET STATISTICS

ALTER PROCEDURE

Changes the definition of an existing stored procedure. Available in DSQL and isql.

Syntax

```
ALTER PROCEDURE name
   [(param <datatype> [, param <datatype> …])]
   [RETURNS (param <datatype> [, param <datatype> …])]
   AS <procedure_body> [terminator]
```

Argument	**Description**
name	Name of an existing procedure
param datatype	Input parameters used by the procedure; legal datatypes are listed under CREATE PROCEDURE
RETURNS *param datatype*	Output parameters used by the procedure; legal datatypes are listed under CREATE PROCEDURE
procedure_body	The procedure body includes: ▪ Local variable declarations ▪ A block of statements in procedure and trigger language See CREATE PROCEDURE for a complete description
terminator	Terminator defined by the isql SET TERM command to signify the end of the procedure body; required by **isql**

Description ALTER PROCEDURE changes an existing stored procedure without affecting its dependencies. It can modify a procedure's input parameters, output parameters, and body.

The complete procedure header and body must be included in the ALTER PROCEDURE statement. The syntax is exactly the same as CREATE PROCEDURE, except CREATE is replaced by ALTER.

Important Be careful about changing the type, number, and order of input and output parameters to a procedure, since existing application code may assume the procedure has its original format.

A procedure can be altered by its creator, the SYSDBA user, and any users with operating system root privileges.

Procedures in use are not altered until they are no longer in use.

ALTER PROCEDURE changes take effect when they are committed. Changes are then reflected in all applications that use the procedure without recompiling or relinking.

Example The following isql statements alter the GET_EMP_PROJ procedure, changing the return parameter to have a datatype of VARCHAR(20):

```
SET TERM !! ;
```

```
ALTER PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID VARCHAR(20)) AS
   BEGIN
      FOR SELECT PROJ_ID
      FROM EMPLOYEE_PROJECT
      WHERE EMP_NO = :emp_no
      INTO :proj_id
      DO
         SUSPEND;
   END !!
SET TERM ; !!
```

See Also CREATE PROCEDURE, DROP PROCEDURE, EXECUTE PROCEDURE

For more information on creating and using procedures, see the *Data Definition Guide*.

ALTER TABLE

Changes a table by adding, dropping, or modifying columns or integrity constraints. Available in SQL, DSQL, and isql.

Syntax

```
ALTER TABLE table <operation> [, <operation> …];

<operation> = {ADD <col_def>
    | ADD <tconstraint>
    | ALTER [COLUMN] column_name <alt_col_clause>
    | DROP col
    | DROP CONSTRAINT constraint}

<alt_col_clause> = {TO new_col_name
    | TYPE new_col_datatype
    | POSITION new_col_position}

<col_def> = col {<datatype> | COMPUTED [BY] (<expr>) | domain}
    [DEFAULT {literal | NULL | USER}]
    [NOT NULL]
    [<col_constraint>]
    [COLLATE collation]

<datatype> =
    {SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION}[<array_dim>]
    | (DATE | TIME | TIMESTAMP}[<array_dim>]
    | {DECIMAL | NUMERIC} [(precision [, scale])] [<array_dim>]
    | {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR} [(int)]
        [<array_dim>] [CHARACTER SET charname]
    | {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
        [VARYING] [(int)] [<array_dim>]
    | BLOB [SUB_TYPE {int | subtype_name}] [SEGMENT SIZE int]
        [CHARACTER SET charname]
    | BLOB [(seglen [, subtype])]<array_dim> = [[x:]y [, [x:]y …]]

<expr> = A valid SQL expression that results in a single value.

<col_constraint> = [CONSTRAINT constraint]
    { UNIQUE
    | PRIMARY KEY
    | REFERENCES other_table [(other_col [, other_col …])]
        [ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
        [ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
    | CHECK (<search_condition>)}

<tconstraint> = [CONSTRAINT constraint]
    {{PRIMARY KEY | UNIQUE} (col [, col …])
    | FOREIGN KEY (col [, col …]) REFERENCES other_table
        [ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
        [ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
    | CHECK (<search_condition>)}

<search_condition> = <val> <operator> {<val> | (<select_one>)}
    | <val> [NOT] BETWEEN <val> AND <val>
    | <val> [NOT] LIKE <val> [ESCAPE <val>]
    | <val> [NOT] IN (<val> [, <val> …] | <select_list>)
    | <val> IS [NOT] NULL
    | <val> {>= | <=}
    | <val> [NOT] {= | < | >}
```

```
   | {ALL | SOME | ANY} (<select_list>)
   | EXISTS (<select_expr>)
   | SINGULAR (<select_expr>)
   | <val> [NOT] CONTAINING <val>
   | <val> [NOT] STARTING [WITH] <val>
   | (<search_condition>)
   | NOT <search_condition>
   | <search_condition> OR <search_condition>
   | <search_condition> AND <search_condition>

<val> = { col [<array_dim>] | :variable
   | <constant> | <expr> | <function>
   | udf ([<val> [, <val> …]])
   | NULL | USER | RDB$DB_KEY | ? }
   [COLLATE collation]

<constant> = num | 'string' | charsetname 'string'

<function> = COUNT (* | [ALL] <val> | DISTINCT <val>)
   | SUM ([ALL] <val> | DISTINCT <val>)
   | AVG ([ALL] <val> | DISTINCT <val>)
   | MAX ([ALL] <val> | DISTINCT <val>)
   | MIN ([ALL] <val> | DISTINCT <val>)
   | CAST (<val> AS <datatype>)
   | UPPER (<val>)
   | GEN_ID (generator, <val>)

<operator> = {= | < | > | <= | >= | !< | !> | <> | !=}

<select_one> = SELECT on a single column; returns exactly one value.

<select_list> = SELECT on a single column; returns zero or more values.

<select_expr> = SELECT on a list of values; returns zero or more values.
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Notes on ALTER TABLE syntax

- The column constraints for referential integrity were new in InterBase 5. See *constraint_def* in Table 1 and the Description for ALTER TABLE on page 17.
- You cannot specify a COLLATE clause for Blob columns.
- When declaring arrays, you must include the outermost brackets, shown below in bold. For example, the following statement creates a 5 by 5 two-dimensional array of strings, each of which is 6 characters long:

  ```
  my_array = varchar(6)[5,5]
  ```

  Use the colon (:) to specify an array with a starting point other than 1. The following example creates an array of integers that begins at 20 and ends at 30:

  ```
  my_array = integer[20:30]
  ```

- For the full syntax of *search_condition*, see CREATE TABLE.

TABLE 1 The ALTER TABLE statement

Argument	**Description**
table	Name of an existing table to modify
operation	Action to perform on the table. Valid options are: ▪ ADD a new column or table constraint to a table ▪ DROP an existing column or constraint from a table
col_def	Description of a new column to add ▪ Must include a column name and *datatype* ▪ Can also include default values, column constraints, and a specific collation order
col	Name of the column to add or drop; column name must be unique within the table
datatype	Datatype of the column
ALTER [COLUMN]	Modifies column names, datatypes, and positions.
COMPUTED [BY] *expr*	Specifies that the value of the column's data is calculated from *expr* at runtime and is therefore not allocated storage space in the database ▪ *expr* can be any arithmetic expression valid for the datatypes in the expression ▪ Any columns referenced in *expr* must exist before they can be used in *expr* ▪ *expr* cannot reference Blob columns ▪ *expr* must return a single value, and cannot return an array
domain	Name of an existing domain
DEFAULT	Specifies a default value for column data; this value is entered when no other entry is made; possible values are: ▪ *literal*: Inserts a specified string, numeric value, or date value ▪ NULL: Enters a NULL value ▪ USER: Enters the user name of the current user; column must be of compatible text type to use the default Defaults set at column level override defaults set at the domain level
CONSTRAINT *constraint*	Name of a column or table constraint; the constraint name must be unique within the table
constraint_def	Specifies the kind of column constraint; valid options are UNIQUE, PRIMARY KEY, CHECK, and

	REFERENCES
CHECK *search_condition*	An attempt to enter a new value in the column fails if the value does not meet the *search_condition*
REFERENCES	Specifies that the column values are derived from column values in another table; if you do not specify column names, InterBase looks for a column with the same name as the referencing column in the referenced table
ON DELETE | ON UPDATE	Used with REFERENCES: Changes a foreign key whenever the referenced primary key changes; valid options are: ▪ [Default] NO ACTION: Does not change the foreign key; may cause the primary key update to fail due to referential integrity checks ▪ CASCADE: For ON DELETE, deletes the corresponding foreign key; for ON UPDATE, updates the corresponding foreign key to the new value of the primary key ▪ SET NULL: Sets all the columns of the corresponding foreign key to NULL ▪ SET DEFAULT: Sets every column of the corresponding foreign key is set to its default value in effect when the referential integrity constraint is defined; when the default for a foreign column changes after the referential integrity constraint is defined, the change does not have an effect on the default value used in the referential integrity constraint
NOT NULL	Specifies that a column cannot contain a NULL value ▪ If a table already has rows, a new column cannot be NOT NULL ▪ NOT NULL is a column attribute only
DROP CONSTRAINT	Drops the specified table constraint
table_constraint	Description of the new table constraint; constraints can be PRIMARY KEY, UNIQUE, FOREIGN KEY, or CHECK
COLLATE *collation*	Establishes a default sorting behavior for the column

Description ALTER TABLE modifies the structure of an existing table. A single ALTER TABLE statement can perform multiple adds and drops.

- A table can be altered by its creator, the SYSDBA user, and any users with operating system superuser privileges.
- ALTER TABLE fails if the new data in a table violates a PRIMARY KEY or UNIQUE

constraint definition added to the table. Dropping a column fails if any of the following are true:

· The column is part of a UNIQUE, PRIMARY, or FOREIGN KEY constraint

· The column is used in a CHECK constraint

· The column is used in the *value* expression of a computed column

· The column is referenced by another database object such as a view

Important When a column is dropped, all data stored in it is lost.

Referential integrity constraints

- To ensure that the referential integrity of foreign keys is preserved, use the ON UPDATE and ON DELETE options for all REFERENCES statements. The values for these cascading referential integrity options are given in Table 1, "The ALTER TABLE statement," on page 15.
- If you do not use the ON UPDATE and ON DELETE options, you must drop the constraint or computed column before dropping the table column. To drop a PRIMARY KEY or UNIQUE constraints that is referenced by FOREIGN KEY constraints, drop the FOREIGN KEY constraint before dropping the PRIMARY KEY or UNIQUE key it references.
- You can create a FOREIGN KEY reference to a table that is owned by someone else only if that owner has explicitly granted you the REFERENCES privilege on that table using GRANT. Any user who updates your foreign key table must have REFERENCES or SELECT privileges on the referenced primary key table.
- You can add a check constraint to a column that is based on a domain, but be aware that changes to tables that contain CHECK constraints with subqueries may cause constraint violations.
- Naming column constraints is optional. If you do not specify a name, InterBase assigns a system-generated name. Assigning a descriptive name can make a constraint easier to find for changing or dropping, and easier to find when its name appears in a constraint violation error message.

Example The following isql statement adds a column to a table and drops a column:

```
ALTER TABLE COUNTRY
   ADD CAPITAL VARCHAR(25),
   DROP CURRENCY;
```

This statement results in the loss of all data in the dropped CURRENCY column.

The next isql statement adds two columns to a table and defines a UNIQUE constraint on one of them:

```
ALTER TABLE COUNTRY
   ADD CAPITAL VARCHAR(25) NOT NULL UNIQUE,
   ADD LARGEST_CITY VARCHAR(25) NOT NULL;
```

The next isql statement changes the name of the LARGEST_CITY column to BIGGEST_CITY:

```
ALTER TABLE COUNTRY ALTER LARGEST_CITY TO BIGGEST_CITY;
```

See Also ALTER DOMAIN, CREATE DOMAIN, CREATE TABLE

For more information about altering tables, see the *Embedded SQL Guide*.

ALTER TRIGGER

Changes an existing trigger. Available in DSQL and isql.

Syntax

```
ALTER TRIGGER name
    [ACTIVE | INACTIVE]
    [{BEFORE | AFTER} {DELETE | INSERT | UPDATE}]
    [POSITION number]
    [AS <trigger_body>] [terminator]
```

Argument	**Description**
name	Name of an existing trigger
ACTIVE	[Default] Specifies that a trigger action takes effect when fired
INACTIVE	Specifies that a trigger action does *not* take effect
BEFORE	Specifies the trigger fires before the associated operation takes place
AFTER	Specifies the trigger fires after the associated operation takes place
DELETE| INSERT |UPDATE	Specifies the table operation that causes the trigger to fire
POSITION *number*	Specifies order of firing for triggers before the same action or after the same action ▪ *number* must be an integer between 0 and 32,767, inclusive ▪ Lower-number triggers fire first ▪ Triggers for a table need not be consecutive; triggers on the same action with the same position number fire in random order
trigger_body	Body of the trigger: a block of statements in procedure and trigger language ▪ See CREATE TRIGGER for a complete description
terminator	Terminator defined by the **isql** SET TERM command to signify the end of the trigger body; not needed when altering only the trigger header

Description ALTER TRIGGER changes the definition of an existing trigger. If any of the arguments to ALTER TRIGGER are omitted, then they default to their current values, that is the value specified by CREATE TRIGGER, or the last ALTER TRIGGER.

ALTER TRIGGER can change:

- Header information only, including the trigger activation status, when it performs its actions, the event that fires the trigger, and the order in which the trigger fires compared to other triggers.

- Body information only, the trigger statements that follow the AS clause.
- Header and trigger body information. In this case, the new trigger definition replaces the old trigger definition.

A trigger can be altered by its creator, the SYSDBA user, and any users with operating system root privileges.

Note To alter a trigger defined automatically by a CHECK constraint on a table, use ALTER TABLE to change the constraint definition.

Examples The following isql statement modifies the trigger, SET_CUST_NO, to be inactive:

```
ALTER TRIGGER SET_CUST_NO INACTIVE;
```

The next isql statement modifies the trigger, SET_CUST_NO, to insert a row into the table, NEW_CUSTOMERS, for each new customer.

```
SET TERM !! ;
ALTER TRIGGER SET_CUST_NO FOR CUSTOMER
BEFORE INSERT AS
   BEGIN
      NEW.CUST_NO = GEN_ID(CUST_NO_GEN, 1);
      INSERT INTO NEW_CUSTOMERS(NEW.CUST_NO, TODAY)
   END !!
SET TERM ; !!
```

See Also CREATE TRIGGER, DROP TRIGGER

For more information about triggers, see the *Data Definition Guide*.

AVG( )

Calculates the average of numeric values in a specified column or expression. Available in SQL, DSQL, and isql.

Syntax

```
AVG ([ALL] value | DISTINCT value)
```

Argument	Description
ALL	Returns the average of all values
DISTINCT	Eliminates duplicate values before calculating the average
value	A column or expression that evaluates to a numeric datatype

Description AVG() is an aggregate function that returns the average of the values in a specified column or expression. Only numeric datatypes are allowed as input to AVG().

If a field value involved in a calculation is NULL or unknown, it is automatically excluded from the calculation. Automatic exclusion prevents averages from being skewed by meaningless data.

AVG() computes its value over a range of selected rows. If the number of rows returned by a SELECT is zero, AVG() returns a NULL value.

Examples The following embedded SQL statement returns the average of all rows in a table:

```
EXEC SQL
   SELECT AVG (BUDGET) FROM DEPARTMENT INTO :avg_budget;
```

The next embedded SQL statement demonstrates the use of SUM(), AVG(), MIN(), and MAX() over a subset of rows in a table:

```
EXEC SQL
   SELECT SUM (BUDGET), AVG (BUDGET), MIN (BUDGET), MAX (BUDGET)
   FROM DEPARTMENT
   WHERE HEAD_DEPT = :head_dept
   INTO :tot_budget, :avg_budget, :min_budget, :max_budget;
```

See Also COUNT( ), MAX( ), MIN( ), SUM( )

BASED ON

Declares a host-language variable based on a column. Available in SQL.

Syntax

```
BASED [ON] [dbhandle.]table.col[.SEGMENT] variable;
```

Argument	**Description**
dbhandle	Handle for the database in which a table resides in a multi-database program; *dbhandle* must be previously declared in a SET DATABASE statement
table.col	Name of table and name of column on which the variable is based
.SEGMENT	Bases the local variable size on the segment length of the Blob column during BLOB FETCH operations; use only when *table.col* refers to a column of BLOB datatype
variable	Name of the host-language variable that inherits the characteristics of a database column

Description BASED ON is a preprocessor directive that creates a host-language variable based on a column definition. The host variable inherits the attributes described for the column and any characteristics that make the variable type consistent with the programming language in use. For example, in C, BASED ON adds one byte to CHAR and VARCHAR variables to accommodate the NULL character terminator.

Use BASED ON in a program's variable declaration section.

Note BASED ON does not require the EXEC SQL keywords.

To declare a host-language variable large enough to hold a Blob segment during FETCH operations, use the SEGMENT option of the BASED ON clause. The variable's size is derived from the segment length of a Blob column. If the segment length for the Blob column is changed in the database, recompile the program to adjust the size of host variables created with BASED ON.

Examples The following embedded statements declare a host variable based on a column:

```
EXEC SQL
   BEGIN DECLARE SECTION
      BASED_ON EMPLOYEE.SALARY salary;
EXEC SQL
   END DECLARE SECTION;
```

See Also BEGIN DECLARE SECTION, CREATE TABLE, END DECLARE SECTION

BEGIN DECLARE SECTION

Identifies the start of a host-language variable declaration section. Available in SQL.

Syntax

```
BEGIN DECLARE SECTION;
```

Description BEGIN DECLARE SECTION is used in embedded SQL applications to identify the start of host-language variable declarations for variables that will be used in subsequent SQL statements. BEGIN DECLARE SECTION is also a preprocessor directive that instructs gpre to declare SQLCODE automatically for the applications programmer.

Important BEGIN DECLARE SECTION must always appear within a module's global variable declaration section.

Example The following embedded SQL statements declare a section and a host-language variable:

```
EXEC SQL
   BEGIN DECLARE SECTION;
      BASED ON EMPLOYEE.SALARY salary;
EXEC SQL
   END DECLARE SECTION;
```

See Also BASED ON, END DECLARE SECTION

CAST( )

Converts a column from one datatype to another. Available in SQL, DSQL, and isql.

Syntax

```
CAST (value AS datatype)
```

Argument	**Description**
val	A column, constant, or expression; in SQL, *val* can also be a host-language variable, function, or UDF
datatype	Datatype to which to convert

Description CAST() allows mixing of numerics and characters in a single expression by converting *val* to a specified datatype.

Normally, only similar datatypes can be compared in search conditions. CAST() can be used in search conditions to translate one datatype into another for comparison purposes.

Datatypes can be converted as shown in the following table:

TABLE 2 Compatible datatypes for CAST()

From datatype class	**To datatype class**
Numeric	character, varying character, date, time, timestamp
Character, varying character	numeric, date, time, timestamp
Date	character, varying character, timestamp
Time	character, varying character, timestamp
Timestamp	character, varying character, date, time
Blob, arrays	—

An error results if a given datatype cannot be converted into the datatype specified in CAST().

Example In the following WHERE clause, CAST() is used to translate a CHARACTER datatype, INTERVIEW_DATE, to a DATE datatype to compare against a DATE datatype, HIRE_DATE:

```
. . .
   WHERE HIRE_DATE = CAST (INTERVIEW_DATE AS DATE);
```

See Also UPPER( )

CLOSE

Closes an open cursor. Available in SQL.

Syntax

```
CLOSE cursor;
```

Argument	**Description**
cursor	Name of an open cursor

Description CLOSE terminates the specified cursor, releasing the rows in its active set and any associated system resources. A cursor is a one-way pointer into the ordered set of rows retrieved by the select expression in the DECLARE CURSOR statement. A cursor enables sequential access to retrieved rows in turn and update in place.

There are four related cursor statements:

Stage	**Statement**	**Purpose**
1	DECLARE CURSOR	Declares the cursor; the SELECT statement determines rows retrieved for the cursor
2	OPEN	Retrieves the rows specified for retrieval with DECLARE CURSOR; the resulting rows become the cursor's *active set*
3	FETCH	Retrieves the current row from the active set, starting with the first row; subsequent FETCH statements advance the cursor through the set
4	CLOSE	Closes the cursor and releases system resources

FETCH statements cannot be issued against a closed cursor. Until a cursor is closed and reopened, InterBase does not reevaluate values passed to the search conditions. Another user can commit changes to the database while a cursor is open, making the active set different the next time that cursor is reopened.

Note In addition to CLOSE, COMMIT and ROLLBACK automatically close all cursors in a transaction.

Example The following embedded SQL statement closes a cursor:

```
EXEC SQL
   CLOSE BC;
```

See Also CLOSE (BLOB), COMMIT, DECLARE CURSOR, FETCH, OPEN, ROLLBACK

CLOSE (BLOB)

Terminates a specified Blob cursor and releases associated system resources. Available in SQL.

Syntax

```
CLOSE blob_cursor;
```

Argument	Description
blob_cursor	Name of an open Blob cursor

Description CLOSE closes an opened read or insert Blob cursor. Generally a Blob cursor should only be closed after:

- Fetching all the Blob segments for BLOB READ operations.
- Inserting all the segments for BLOB INSERT operations.

Example The following embedded SQL statement closes a Blob cursor:

```
EXEC SQL
   CLOSE BC;
```

See Also DECLARE CURSOR (BLOB), FETCH (BLOB), INSERT CURSOR (BLOB), OPEN (BLOB)

COMMIT

Makes a transaction's changes to the database permanent, and ends the transaction. Available in SQL, DSQL, and isql.

Syntax

```
COMMIT [WORK] [TRANSACTION name] [RELEASE] [RETAIN [SNAPSHOT]];
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
WORK	An optional word used for compatibility with other relational databases that require it
TRANSACTION *name*	Commits transaction *name* to database. Without this option, COMMIT affects the default transaction
RELEASE	Available for compatibility with earlier versions of InterBase
RETAIN [SNAPSHOT]	Commits changes and retains current transaction context

Description COMMIT is used to end a transaction and:

- Write all updates to the database.
- Make the transaction's changes visible to subsequent SNAPSHOT transactions or READ COMMITTED transactions.
- Close open cursors, unless the RETAIN argument is used.

A transaction ending with COMMIT is considered a successful termination. Always use COMMIT or ROLLBACK to end the default transaction.

Tip After read-only transactions, which make no database changes, use COMMIT rather than ROLLBACK. The effect is the same, but the performance of subsequent transactions is better and the system resources used by them are reduced.

Important The RELEASE argument is only available for compatibility with previous versions of InterBase. To detach from a database use DISCONNECT.

Examples The following isql statement makes permanent the changes to the database made by the default transaction:

```
COMMIT;
```

The next embedded SQL statement commits a named transaction:

```
EXEC SQL
   COMMIT TR1;
```

The following embedded SQL statement uses COMMIT RETAIN to commit changes while maintaining the current transaction context:

```
EXEC SQL
```

```
   COMMIT RETAIN;
```

See Also DISCONNECT, ROLLBACK

For more information about handling transactions, see the *Embedded SQL Guide*.

CONNECT

Attaches to one or more databases. Available in SQL. A subset of CONNECT options is available in isql.

Syntax isql form:

```
CONNECT 'filespec' [USER 'username'][PASSWORD 'password']
   [CACHE int] [ROLE 'rolename']
```

SQL form:

```
CONNECT [TO] {ALL | DEFAULT} <config_opts>
   | <db_specs> <config_opts> [, <db_specs> <config_opts>...];
```

```
<db_specs> = dbhandle
   | {'filespec' | :variable} AS dbhandle
```

```
<config_opts> = [USER {'username' | :variable}]
   [PASSWORD {'password' | :variable}]
   [ROLE {'rolename' | :variable}]
   [CACHE int [BUFFERS]]
```

Argument	**Description**
{ALL | DEFAULT}	Connects to all databases specified with SET DATABASE; options specified with CONNECT TO ALL affect all databases.
'*filespec*'	Database file name; can include path specification and node. The filespec must be in quotes if it includes spaces.
dbhandle	Database handle declared in a previous SET DATABASE statement; available in embedded SQL but not in isql.
:variable	Host-language variable specifying a database, user name, or password; available in embedded SQL but not in isql.
AS *dbhandle*	Attaches to a database and assigns a previously declared handle to it; available in embedded SQL but not in isql.
USER {'*username*' | :*variable*}	String or host-language variable that specifies a user name for use when attaching to the database. The server checks the user name against the security database. User names are case insensitive on the server.
PASSWORD {'*password*' | :*variable*}	String or host-language variable, up to 8 characters in size, that specifies password for use when attaching to the database. The server checks the user name and password against the security database. Case sensitivity is retained for the comparison.

ROLE {'*rolename*' | :*variable*}	String or host-language variable, up to 31 characters in size, which specifies the role that the user adopts on connection to the database. The user must have previously been granted membership in the role to gain the privileges of that role. Regardless of role memberships granted, the user has the privileges of a role at connect time only if a ROLE clause is specified in the connection. The user can adopt at most one role per connection, and cannot switch roles except by reconnecting.
CACHE *int* [BUFFERS]	Sets the number of cache buffers for a database, which determines the number of database pages a program can use at the same time. Values for *int*: ▪ Default: 256 ▪ Maximum value: system-dependent Do not use the *filespec* form of database name with cache assignments.

Description The CONNECT statement:

- Initializes database data structures.
- Determines if the database is on the originating node (a *local database*) or on another node (a *remote database*). An error message occurs if InterBase cannot locate the database.
- Optionally specifies one or more of a user name, password, or role for use when attaching to the database. PC clients must always send a valid user name and password. InterBase recognizes only the first 8 characters of a password.

  If an InterBase user has ISC_USER and ISC_PASSWORD environment variables set and the user defined by those variables is not in the isc4.gdb, the user will receive the following error when attempting to view isc4.gdb users from the local server manager connection: "undefined user name and password." This applies only to the local connection; the automatic connection made through Server Manager bypasses user security.
- Attaches to the database and verifies the header page. The database file must contain a valid database, and the on-disk structure (ODS) version number of the database must be the one recognized by the installed version of InterBase on the server, or InterBase returns an error.
- Optionally establishes a database handle declared in a SET DATABASE statement.
- Specifies a cache buffer for the process attaching to a database.

In SQL programs before a database can be opened with CONNECT, it must be declared with the SET DATABASE statement. isql does not use SET DATABASE.

In SQL programs while the same CONNECT statement can open more than one database, use separate statements to keep code easy to read.

When CONNECT attaches to a database, it uses the default character set (NONE), or one specified in a previous SET NAMES statement.

In SQL programs the CACHE option changes the database cache size count (the total number of available buffers) from the default of 75. This option can be used to:

- Sets a new default size for all databases listed in the CONNECT statement that do not already have a specific cache size.
- Specifies a cache for a program that uses a single database.
- Changes the cache for one database without changing the default for all databases used by the program.

The size of the cache persists as long as the attachment is active. If a database is already attached through a multi-client server, an increase in cache size due to a new attachment persists until all the attachments end. A decrease in cache size does not affect databases that are already attached through a server.

A subset of CONNECT features is available in isql: database file name, USER, and PASSWORD. isql can only be connected to one database at a time. Each time CONNECT is used to attach to a database, previous attachments are disconnected.

Examples The following statement opens a database for use in isql. It uses all the CONNECT options available to isql:

```
CONNECT 'employee.gdb' USER 'ACCT_REC' PASSWORD 'peanuts';
```

The next statement, from an embedded application, attaches to a database file stored in the host-language variable and assigns a previously declared database handle to it:

```
EXEC SQL
   SET DATABASE DB1 = 'employee.gdb';
EXEC SQL
   CONNECT :db_file AS DB1;
```

The following embedded SQL statement attaches to employee.gdb and allocates 150 cache buffers:

```
EXEC SQL
   CONNECT 'accounts.gdb' CACHE 150;
```

The next embedded SQL statement connects the user to all databases specified with previous SET DATABASE statements:

```
EXEC SQL
   CONNECT ALL USER 'ACCT_REC' PASSWORD 'peanuts'
   CACHE 50;
```

The following embedded SQL statement attaches to the database, employee.gdb, with 80 buffers and database employee2.gdb with the default of 75 buffers:

```
EXEC SQL
   CONNECT 'employee.gdb' CACHE 80, 'employee2.gdb';
```

The next embedded SQL statement attaches to all databases and allocates 50 buffers:

```
EXEC SQL
   CONNECT ALL CACHE 50;
```

The following embedded SQL statement connects to EMP1 and v, setting the number of buffers for each to 80:

```
EXEC SQL
   CONNECT EMP1 CACHE 80, EMP2 CACHE 80;
```

The next embedded SQL statement connects to two databases identified by variable names, setting different user names and passwords for each:

```
EXEC SQL
   CONNECT
   :orderdb AS DB1 USER 'ACCT_REC' PASSWORD 'peanuts',
   :salesdb AS DB2 USER 'ACCT_PAY' PASSWORD 'payout';
```

See Also DISCONNECT, SET DATABASE, SET NAMES

Se the *Data Definition Guide* for more information about cache buffers and the *Operations Guide* for more information about database security and isql.

COUNT( )

Calculates the number of rows that satisfy a query's search condition. Available in SQL, DSQL, and isql.

Syntax

```
COUNT ( * | [ALL] value | DISTINCT value)
```

Argument	**Description**
*	Retrieves the number of rows in a specified table, including NULL values
ALL	Counts all non-NULL values in a column
DISTINCT	Returns the number of unique, non-NULL values for the column
val	A column or expression

Description COUNT() is an aggregate function that returns the number of rows that satisfy a query's search condition. It can be used in views and joins as well as in tables.

Example The following embedded SQL statement returns the number of unique currency values it encounters in the COUNTRY table:

```
EXEC SQL
   SELECT COUNT (DISTINCT CURRENCY) INTO :cnt FROM COUNTRY;
```

See Also AVG( ), MAX( ), MIN( ) SUM( )

CREATE DATABASE

Creates a new database. Available in SQL, DSQL, and isql.

Syntax

```
CREATE {DATABASE | SCHEMA} 'filespec'
    [USER 'username' [PASSWORD 'password']]
    [PAGE_SIZE [=] int]
    [LENGTH [=] int [PAGE[S]]]
    [DEFAULT CHARACTER SET charset]
    [<secondary_file>];

<secondary_file> = FILE 'filespec' [<fileinfo>] [<secondary_file>]

<fileinfo> = [LENGTH [=] int [PAGE[S]] | STARTING [AT [PAGE]] int }
    [<fileinfo>]
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
'*filespec*'	A new database file specification; file naming conventions are platform-specific
USER '*username*'	Checks the *username* against valid user name and password combinations in the security database on the server where the database will reside ▪ Windows client applications must provide a user name on attachment to a server ▪ Any client application attaching to a database on NT or NetWare must provide a user name on attachment
PASSWORD '*password*'	Checks the *password* against valid user name and password combinations in the security database on the server where the database will reside; can be up to 8 characters ▪ Windows client applications must provide a user name and password on attachment to a server ▪ Any client application attaching to a database on NT or NetWare must provide a password on attachment
PAGE_SIZE [=] *int*	Size, in bytes, for database pages *int* can be 1024 (default), 2048, 4096, or 8192
DEFAULT CHARACTER SET *charset*	Sets default character set for a database *charset* is the name of a character set; if omitted, character set defaults to NONE
FILE '*filespec*'	Names one or more secondary files to hold database pages after the primary file is filled. For databases created on remote servers,

	secondary file specifications cannot include a node name.
STARTING [AT [PAGE]] *int*	Specifies the starting page number for a secondary file.
LENGTH [=] *int* [PAGE[S]]	Specifies the length of a primary or secondary database file. Use for primary file only if defining a secondary file in the same statement.

Description CREATE DATABASE creates a new, empty database and establishes the following characteristics for it:

- The name of the primary file that identifies the database for users.

  By default, databases are contained in single files.

- The name of any secondary files in which the database is stored.

  A database can reside in more than one disk file if additional file names are specified as secondary files. If a database is created on a remote server, secondary file specifications cannot include a node name.

- The size of database pages.

  Increasing page size can improve performance for the following reasons:

  · Indexes work faster because the depth of the index is kept to a minimum.

  · Keeping large rows on a single page is more efficient.

  · Blob data is stored and retrieved more efficiently when it fits on a single page.

  If most transactions involve only a few rows of data, a smaller page size might be appropriate, since less data needs to be passed back and forth and less memory is used by the disk cache.

- The number of pages in each database file.
- The dialect of the database.

  The initial dialect of the database is the dialect of the client that creates it. For example, if you are using isql, either start it with the -sql_dialect *n* switch or issue the SET SQL DIALECT *n* command before issuing the CREATE DATABASE command. Typically, you would create all databases in dialect 3. Dialect 1 exists to ease the migration of legacy databases.

  Note To change the dialect of a database, use gfix or the Properties dialog in IBConsole. See the Migration chapter in *Getting Started* for information about migrating databases.

- The character set used by the database.

  Choice of DEFAULT CHARACTER SET limits possible collation orders to a subset of all available collation orders. Given a specific character set, a specific collation order can be specified when data is selected, inserted, or updated in a column.

  If you do not specify a default character set, the character set defaults to NONE. Using character set NONE means that there is no character set assumption for columns; data is stored and retrieved just as you originally entered it. You can load any character set into a column defined with NONE, but you cannot load that same data into another column that has been defined with a different character set. In that case, no transliteration is performed between the source and destination character sets, and transliteration errors may occur during assignment.

- System tables that describe the structure of the database.

After creating the database, you define its tables, views, indexes, and system views as well as any triggers, generators, stored procedures, and UDFs that you need.

Important In DSQL, you must execute CREATE DATABASE EXECUTE IMMEDIATE. The database handle and transaction name, if present, must be initialized to zero prior to use.

Read-only databases

Databases are always created in *read-write mode*. You can change a table to *read-only mode* in wither of two ways: You can specify mode -read_only when you restore a backup or you can use gfix -mode read_only to change the mode of a table to read-only. See "Read-only databases" in Chapter 6: "Database Configuration and Maintenance" in the *Operations Guide*.

About file sizes

InterBase dynamically expands the last file in a database as needed until it reaches the 4GB limit. This applies to single-file database as well as to the last file of multifile databases. You should be aware that specifying a LENGTH for such files has no effect. InterBase database files are limited to 4GB. The total file size is the product of the number of database pages times the page size. The default page size is 1KB and the maximum page size is 8KB. However, InterBase files are small at creation time and increase in size as needed. The product of number of pages times page size represents a potential maximum size, not the size at creation.

Examples The following isql statement creates a database in the current directory using isql:

```
CREATE DATABASE 'employee.gdb';
```

The next embedded SQL statement creates a database with a page size of 2048 bytes rather than the default of 1024:

```
EXEC SQL
   CREATE DATABASE 'employee.gdb' PAGE_SIZE 2048;
```

The following embedded SQL statement creates a database stored in two files and specifies its default character set:

```
EXEC SQL
   CREATE DATABASE 'employee.gdb'
      DEFAULT CHARACTER SET ISO8859_1
      FILE 'employee2.gdb' STARTING AT PAGE 10001;
```

See Also ALTER DATABASE, DROP DATABASE

See the *Data Definition Guide* for more information about secondary files, character set specification, and collation order; see the *Operations Guide* for more information about page size.

CREATE DOMAIN

Creates a column definition that is global to the database. Available in SQL, DSQL, and isql.

Syntax

```
CREATE DOMAIN domain [AS] <datatype>
   [DEFAULT {literal | NULL | USER}]
   [NOT NULL] [CHECK (<dom_search_condition>)]
   [COLLATE collation];
```

```
<datatype> =
   {SMALLINT|INTEGER|FLOAT|DOUBLE PRECISION}[<array_dim>]
   | {DATE|TIME|TIMESTAMP}[<array_dim>]
   | {DECIMAL | NUMERIC} [(precision [, scale])] [<array_dim>]
   | {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR} [(int)]
      [<array_dim>] [CHARACTER SET charname]
   | {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
      [VARYING] [(int)] [<array_dim>]
   | BLOB [SUB_TYPE {int | subtype_name}] [SEGMENT SIZE int]
      [CHARACTER SET charname]
   | BLOB [(seglen [, subtype])]
```

```
<array_dim> = [[x:]y [, [x:]y …]]
```

```
<dom_search_condition> = {
   VALUE <operator> value
   | VALUE [NOT] BETWEEN value AND value
   | VALUE [NOT] LIKE value [ESCAPE value]
   | VALUE [NOT] IN (value [, value …])
   | VALUE IS [NOT] NULL
   | VALUE [NOT] CONTAINING value
   | VALUE [NOT] STARTING [WITH] value
   | (<dom_search_condition>)
   | NOT <dom_search_condition>
   | <dom_search_condition> OR <dom_search_condition>
   | <dom_search_condition> AND <dom_search_condition>
   }
```

```
<operator> = {= | < | > | <= | >= | !< | !> | <> | !=}
```

Note on the CREATE DOMAIN syntax

- You cannot specify a COLLATE clause for Blob columns.
- When declaring arrays, you must include the outermost brackets, shown below in bold. For example, the following statement creates a 5 by 5 two-dimensional array of strings, each of which is 6 characters long:

```
my_array = varchar(6)[5,5]
```

Use the colon (:) to specify an array with a starting point other than 1. The following example creates an array of integers that begins at 10 and ends at 20:

```
my_array = integer[20:30]
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
domain	Unique name for the domain
datatype	SQL datatype
DEFAULT	Specifies a default column value that is entered when no other entry is made; possible values are: *literal*—Inserts a specified string, numeric value, or date value NULL—Enters a NULL value USER—Enters the user name of the current user; column must be of compatible character type to use the default
NOT NULL	Specifies that the values entered in a column cannot be NULL
CHECK (*dom_search_condition*)	Creates a single CHECK constraint for the domain
VALUE	Placeholder for the name of a column eventually based on the domain
COLLATE *collation*	Specifies a collation sequence for the domain

Description CREATE DOMAIN builds an inheritable column definition that acts as a template for columns defined with CREATE TABLE or ALTER TABLE. The domain definition contains a set of characteristics, which include:

- Datatype
- An optional default value
- Optional disallowing of NULL values
- An optional CHECK constraint
- An optional collation clause

The CHECK constraint in a domain definition sets a *dom_search_condition* that must be true for data entered into columns based on the domain. The CHECK constraint cannot reference any domain or column.

Note Be careful not to create a domain with contradictory constraints, such as declaring a domain NOT NULL and assigning it a DEFAULT value of NULL.

The datatype specification for a CHAR or VARCHAR text domain definition can include a CHARACTER SET clause to specify a character set for the domain. Otherwise, the domain uses the default database character set.

If you do not specify a default character set, the character set defaults to NONE. Using character set NONE means that there is no character set assumption for columns; data is stored and retrieved just as you originally entered it. You can load any character set into a column defined with NONE, but you cannot load that same data into another column that has been defined with a different character set. In these cases, no transliteration is performed between the source and destination character sets, so errors can occur during assignment.

The COLLATE clause enables specification of a particular collation order for CHAR, VARCHAR, and BLOB text datatypes. Choice of collation order is restricted to those supported for the domain's given character set, which is either the default character set for the entire database, or a different set defined in the CHARACTER SET clause as part of the datatype definition.

Columns based on a domain definition inherit all characteristics of the domain. The domain default, collation clause, and NOT NULL setting can be overridden when defining a column based on a domain. A column based on a domain can add additional CHECK constraints to the domain CHECK constraint.

Examples The following isql statement creates a domain that must have a positive value greater than 1,000, with a default value of 9,999. The keyword VALUE substitutes for the name of a column based on this domain.

```
CREATE DOMAIN CUSTNO
   AS INTEGER
   DEFAULT 9999
   CHECK (VALUE > 1000);
```

The next isql statement limits the values entered in the domain to four specific values:

```
CREATE DOMAIN PRODTYPE
   AS VARCHAR(12)
   CHECK (VALUE IN ('software', 'hardware', 'other', 'N/A'));
```

The following isql statement creates a domain that defines an array of CHAR datatype:

```
CREATE DOMAIN DEPTARRAY AS CHAR(31) [4:5];
```

In the following isql example, the first statement creates a domain with USER as the default. The next statement creates a table that includes a column, ENTERED_BY, based on the USERNAME domain.

```
CREATE DOMAIN USERNAME AS VARCHAR(20)
   DEFAULT USER;

CREATE TABLE ORDERS (ORDER_DATE DATE, ENTERED_BY USERNAME,
   ORDER_AMT DECIMAL(8,2));

INSERT INTO ORDERS (ORDER_DATE, ORDER_AMT)
   VALUES ('1-MAY-93', 512.36);
```

The INSERT statement does not include a value for the ENTERED_BY column, so InterBase automatically inserts the user name of the current user, JSMITH:

```
SELECT * FROM ORDERS;

1-MAY-93 JSMITH 512.36
```

The next isql statement creates a BLOB domain with a TEXT subtype that has an assigned character set:

```
CREATE DOMAIN DESCRIPT AS
   BLOB SUB_TYPE TEXT SEGMENT SIZE 80
   CHARACTER SET SJIS;
```

See Also ALTER DOMAIN, ALTER TABLE, CREATE TABLE, DROP DOMAIN

For more information about character set specification and collation orders, see the *Data Definition Guide*.

CREATE EXCEPTION

Creates a used-defined error and associated message for use in stored procedures and triggers. Available in DSQL and isql.

Syntax

```
CREATE EXCEPTION name 'message';
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
name	Name associated with the exception message; must be unique among exception names in the database
'*message*'	Quoted string containing alphanumeric characters and punctuation; maximum length = 78 characters.

Description CREATE EXCEPTION creates an exception, a user-defined error with an associated message. Exceptions may be raised in triggers and stored procedures.

Exceptions are global to the database. The same message or set of messages is available to all stored procedures and triggers in an application. For example, a database can have English and French versions of the same exception messages and use the appropriate set as needed.

When raised by a trigger or a stored procedure, an exception:

- Terminates the trigger or procedure in which it was raised and undoes any actions performed (directly or indirectly) by it.
- Returns an error message to the calling application. In isql, the error message appears on the screen, unless output is redirected.

Exceptions may be trapped and handled with a WHEN statement in a stored procedure or trigger.

Examples This isql statement creates the exception, UNKNOWN_EMP_ID:

```
CREATE EXCEPTION UNKNOWN_EMP_ID 'Invalid employee number
   or project id.';
```

The following statement from a stored procedure raises the previously created exception when SQLCODE -530 is set, which is a violation of a FOREIGN KEY constraint:

```
. . .
   WHEN SQLCODE -530 DO
      EXCEPTION UNKNOWN_EMP_ID;
. . .
```

See Also ALTER EXCEPTION, ALTER PROCEDURE, ALTER TRIGGER, CREATE PROCEDURE, CREATE TRIGGER, DROP EXCEPTION

For more information on creating, raising, and handling exceptions, see the

Data Definition Guide.

CREATE GENERATOR

Declares a generator to the database. Available in SQL, DSQL, and isql.

Syntax

```
CREATE GENERATOR name;
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
name	Name for the generator

Description CREATE GENERATOR declares a generator to the database and sets its starting value to zero. A generator is a sequential number that can be automatically inserted in a column with the GEN_ID() function. A generator is often used to ensure a unique value in a PRIMARY KEY, such as an invoice number, that must uniquely identify the associated row.

A database can contain any number of generators. Generators are global to the database, and can be used and updated in any transaction. InterBase does not assign duplicate generator values across transactions.

You can use SET GENERATOR to set or change the value of an existing generator when writing triggers, procedures, or SQL statements that call GEN_ID().

Note There is no "drop generator" statement. To remove a generator, delete it from the system table. For example:

```
DELETE FROM RDB$GENERATOR WHERE RDB$GENERATOR_NAME = 'EMPNO_GEN';
```

Example The following isql script fragment creates the generator, EMPNO_GEN, and the trigger, CREATE_EMPNO. The trigger uses the generator to produce sequential numeric keys, incremented by 1, for the NEW.EMPNO column:

```
CREATE GENERATOR EMPNO_GEN;
COMMIT;
SET TERM !! ;
CREATE TRIGGER CREATE_EMPNO FOR EMPLOYEES
   BEFORE INSERT POSITION 0
   AS BEGIN
      NEW.EMPNO = GEN_ID(EMPNO_GEN, 1);
   END
SET TERM ; !!
```

Important Because each statement in a stored procedure body must be terminated by a semicolon, you must define a different symbol to terminate the CREATE TRIGGER in isql. Use SET TERM before CREATE TRIGGER to specify a terminator other than a semicolon. After CREATE TRIGGER, include another SET TERM to change the terminator back to a semicolon.

See Also GEN_ID( ), SET GENERATOR

CREATE INDEX

Creates an index on one or more columns in a table. Available in SQL, DSQL, and isql.

Syntax

```
CREATE [UNIQUE] [ASC[ENDING] | DESC[ENDING]] INDEX index
   ON table (col [, col …]);
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
UNIQUE	Prevents insertion or updating of duplicate values into indexed columns
ASC[ENDING]	Sorts columns in ascending order, the default order if none is specified
DESC[ENDING]	Sorts columns in descending order
index	Unique name for the index
table	Name of the table on which the index is defined
col	Column in *table* to index

Description Creates an index on one or more columns in a table. Use CREATE INDEX to improve speed of data access. Using an index for columns that appear in a WHERE clause may improve search performance.

Important You cannot index Blob columns or arrays.

A UNIQUE index cannot be created on a column or set of columns that already contains duplicate or NULL values.

ASC and DESC specify the order in which an index is sorted. For faster response to queries that require sorted values, use the index order that matches the query's ORDER BY clause. Both an ASC and a DESC index can be created on the same column or set of columns to access data in different orders.

Tip To improve index performance, use SET STATISTICS to recompute index selectivity, or rebuild the index by making it inactive, then active with sequential calls to ALTER INDEX.

Examples The following isql statement creates a unique index:

```
CREATE UNIQUE INDEX PRODTYPEX ON PROJECT (PRODUCT, PROJ_NAME);
```

The next isql statement creates a descending index:

```
CREATE DESCENDING INDEX CHANGEX ON SALARY_HISTORY (CHANGE_DATE);
```

The following isql statement creates a two-column index:

```
CREATE INDEX NAMEX ON EMPLOYEE (LAST_NAME, FIRST_NAME);
```

See Also ALTER INDEX, DROP INDEX, SELECT, SET STATISTICS

CREATE PROCEDURE

Creates a stored procedure, its input and output parameters, and its actions. Available in DSQL, and isql.

Syntax

```
CREATE PROCEDURE name
   [(param <datatype> [, param <datatype> …])]
   [RETURNS <datatype> [, param <datatype> …])]
   AS <procedure_body> [terminator]

<procedure_body> =
   [<variable_declaration_list>]
   <block>

<variable_declaration_list> =
   DECLARE VARIABLE var <datatype>;
   [DECLARE VARIABLE var <datatype>; …]

<block> =
BEGIN
   <compound_statement>
   [<compound_statement> …]
END

<compound_statement> = {<block> | statement;}

<datatype> = SMALLINT
   | INTEGER
   | FLOAT
   | DOUBLE PRECISION
   | {DECIMAL | NUMERIC} [(precision [, scale])]
   | {DATE | TIME | TIMESTAMP)
   | {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
      [(int)] [CHARACTER SET charname]
   | {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR} [VARYING] [(int)]
```

Argument	**Description**
name	Name of the procedure. Must be unique among procedure, table, and view names in the database
param datatype	Input parameters that the calling program uses to pass values to the procedure: *param*: Name of the input parameter, unique for variables in the procedure *datatype*: An InterBase datatype
RETURNS *param datatype*	Output parameters that the procedure uses to return values to the calling program: *param*: Name of the output parameter, unique for variables within the procedure *datatype*: An InterBase datatype The procedure returns the values of output parameters when it reaches a SUSPEND statement in the procedure body

AS	Keyword that separates the procedure header and the procedure body
DECLARE VARIABLE *var datatype*	Declares local variables used only in the procedure; must be preceded by DECLARE VARIABLE and followed by a semicolon (;). *var* is the name of the local variable, unique for variables in the procedure.
statement	Any single statement in InterBase procedure and trigger language; must be followed by a semicolon (;) except for BEGIN and END statements
terminator	Terminator defined by SET TERM ▪ Signifies the end of the procedure body; ▪ Used only in isql

Description CREATE PROCEDURE defines a new stored procedure to a database. A stored procedure is a self-contained program written in InterBase procedure and trigger language, and stored as part of a database's metadata. Stored procedures can receive input parameters from and return values to applications.

InterBase procedure and trigger language includes all SQL data manipulation statements and some powerful extensions, including IF … THEN … ELSE, WHILE … DO, FOR SELECT … DO, exceptions, and error handling.

There are two types of procedures:

- *Select* procedures that an application can use in place of a table or view in a SELECT statement. A select procedure must be defined to return one or more values, or an error will result.
- *Executable* procedures that an application can call directly, with the EXECUTE PROCEDURE statement. An executable procedure need not return values to the calling program.

A stored procedure is composed of a *header* and a *body*.

The procedure header contains:

- The *name* of the stored procedure, which must be unique among procedure and table names in the database.
- An optional list of *input parameters* and their datatypes that a procedure receives from the calling program.
- RETURNS followed by a list of *output parameters* and their datatypes if the procedure returns values to the calling program.

The procedure body contains:

- An optional list of *local variables* and their datatypes.
- A *block* of statements in InterBase procedure and trigger language, bracketed by BEGIN and END. A block can itself include other blocks, so that there may be many levels of nesting.

Important Because each statement in a stored procedure body must be terminated by a semicolon, you must define a different symbol to terminate the CREATE PROCEDURE statement in isql. Use SET TERM before CREATE PROCEDURE to specify a terminator other than a semicolon. After the CREATE PROCEDURE statement, include another SET TERM to change the terminator back to a semicolon.

InterBase does not allow database changes that affect the behavior of an existing stored procedure (for example, DROP TABLE or DROP EXCEPTION). To see all procedures defined for the current database or the text and parameters of a named procedure, use the isql internal commands SHOW PROCEDURES or SHOW PROCEDURE *procedure*.

InterBase procedure and trigger language is a complete programming language for stored procedures and triggers. It includes:

- SQL data manipulation statements: INSERT, UPDATE, DELETE, and singleton SELECT.
- SQL operators and expressions, including generators and UDFs that are linked with the database.
- Extensions to SQL, including assignment statements, control-flow statements, context variables (for triggers), event-posting statements, exceptions, and error-handling statements.

The following table summarizes language extensions for stored procedures.

TABLE 3 Procedure and trigger language extensions

Statement	**Description**
BEGIN … END	Defines a block of statements that executes as one ▪ The BEGIN keyword starts the block; the END keyword terminates it ▪ Neither should end with a semicolon
variable = *expression*	Assignment statement: assigns the value of *expression* to *variable*, a local variable, input parameter, or output parameter
/* *comment_text* */	Programmer's comment, where *comment_text* can be any number of lines of text
EXCEPTION *exception_name*	Raises the named exception: an exception is a user-defined error that returns an error message to the calling application unless handled by a WHEN statement
EXECUTE PROCEDURE *proc_name* [*var* [, *var* …]] [RETURNING_VALUES *var* [, *var* …]]	Executes stored procedure, *proc_name,* with the listed input arguments, returning values in the listed output arguments following RETURNING_VALUES; input and output arguments must be local variables
EXIT	Jumps to the final END statement in the

	procedure
FOR *select_statement* DO *compound_statement*	Repeats the statement or block following DO for every qualifying row retrieved by *select_statement* *select_statement* is like a normal SELECT statement, except that there is an INTO clause that is required and which must come last
compound_statement	Either a single statement in procedure and trigger language or a block of statements bracketed by BEGIN and END
IF (*condition*) THEN *compound_statement* [ELSE *compound_statement*]	Tests *condition*, and if it is TRUE, performs the statement or block following THEN; otherwise, performs the statement or block following ELSE, if present *condition*: a Boolean expression (TRUE, FALSE, or UNKNOWN), generally two expressions as operands of a comparison operator
NEW.*column*	New context variable that indicates a new column value in an INSERT or UPDATE operation
OLD.*column*	Old context variable that indicates a column value before an UPDATE or DELETE operation
POST_EVENT *event_name* | *col*	Posts the event, *event_name*, or uses the value in *col* as an event name
SUSPEND	In a SELECT procedure: ▪ Suspends execution of procedure until next FETCH is issued by the calling application ▪ Returns output values, if any, to the calling application ▪ Not recommended for executable procedures
WHILE (*condition*) DO *compound_statement*	While *condition* is TRUE, keep performing *compound_statement* ▪ *Tests condition*, andperforms *compound_statement* if *condition* is TRUE ▪ Repeats this sequence until *condition* is no longer TRUE
WHEN {*error* [, *error* …] | ANY} DO *compound_statement*	Error-handling statement: when one of the specified errors occurs, performs *compound_statement* ▪ WHEN statements, if present, must come at the end of a block, just before END ▪ *error*: EXCEPTION *exception_name,* SQLCODE *errcode* or GDSCODE *number* ▪ ANY: Handles any errors

Examples The following procedure, SUB_TOT_BUDGET, takes a department number as its input parameter, and returns the total, average, minimum, and maximum budgets of departments with the specified HEAD_DEPT.

```
/* Compute total, average, smallest, and largest department budget.
*Parameters:
* department id
*
*Returns:
* total budget
* average budget
* min budget
* max budget */
SET TERM !! ;
CREATE PROCEDURE SUB_TOT_BUDGET (HEAD_DEPT CHAR(3))
   RETURNS (tot_budget DECIMAL(12, 2), avg_budget DECIMAL(12, 2),
      min_budget DECIMAL(12, 2), max_budget DECIMAL(12, 2))
   AS
   BEGIN
      SELECT SUM(BUDGET), AVG(BUDGET), MIN(BUDGET), MAX(BUDGET)
         FROM DEPARTMENT
         WHERE HEAD_DEPT = :head_dept
         INTO :tot_budget, :avg_budget, :min_budget, :max_budget;
      EXIT;
   END !!
SET TERM ; !!
```

The following select procedure, ORG_CHART, displays an organizational chart:

```
/* Display an org-chart.
*
*  Parameters:
*      --
*  Returns:
*      parent department
*      department name
*      department manager
*      manager's job title
*      number of employees in the department */
CREATE PROCEDURE ORG_CHART
   RETURNS (HEAD_DEPT CHAR(25), DEPARTMENT CHAR(25),
      MNGR_NAME CHAR(20), TITLE CHAR(5), EMP_CNT INTEGER)
   AS
      DECLARE VARIABLE mngr_no INTEGER;
      DECLARE VARIABLE dno CHAR(3);
   BEGIN
      FOR SELECT H.DEPARTMENT, D.DEPARTMENT, D.MNGR_NO, D.DEPT_NO
         FROM DEPARTMENT D
         LEFT OUTER JOIN DEPARTMENT H ON D.HEAD_DEPT = H.DEPT_NO
         ORDER BY D.DEPT_NO
         INTO :head_dept, :department, :mngr_no, :dno
      DO
         BEGIN
            IF (:mngr_no IS NULL) THEN
            BEGIN
               MNGR_NAME = '--TBH--';
               TITLE = '';
            END
         ELSE
            SELECT FULL_NAME, JOB_CODE
               FROM EMPLOYEE
               WHERE EMP_NO = :mngr_no
               INTO :mngr_name, :title;
            SELECT COUNT(EMP_NO)
               FROM EMPLOYEE
               WHERE DEPT_NO = :dno
               INTO :emp_cnt;
            SUSPEND;
      END
END !!
```

When ORG_CHART is invoked, for example in the following isql statement:

```
SELECT * FROM ORG_CHART
```

it displays the department name for each department, which department it is in, the department manager's name and title, and the number of employees in the department.

```
HEAD_DEPT             DEPARTMENT           MGR_NAME           TITLE  EMP_CNT
===================== ===================  ================   ====   =======
                      Corporate            Bender, Oliver H.  CEO        2
                      Headquarters
Corporate HeadquartersSales and Marketing  MacDonald, Mary S.VP          2
Sales and Marketing   Pacific Rim          Baldwin, Janet     Sales      2
                      Headquarters
Pacific Rim           Field Office: Japan  Yamamoto, Takashi  SRep       2
Headquarters
Pacific Rim           Field Office:        ——TBH——                       0
Headquarters          Singapore
```

ORG_CHART must be used as a select procedure to display the full organization. If called with EXECUTE PROCEDURE, the first time it encounters the SUSPEND statement, it terminates, returning the information for Corporate Headquarters only.

See Also ALTER EXCEPTION, ALTER PROCEDURE, CREATE EXCEPTION, DROP EXCEPTION, DROP PROCEDURE, EXECUTE PROCEDURE, SELECT

For more information on creating and using procedures, see the *Data Definition Guide*.

CREATE ROLE

Creates a role.

Syntax

```
CREATE ROLE rolename;
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
rolename	Name associated with the role; must be unique among role names in the database

Description Roles created with CREATE ROLE can be granted privileges just as users can. These roles can be granted to users, who then inherit the privilege list that has been granted to the role. Users must specify the role at connect time. Use GRANT to grant privileges (ALL, SELECT, INSERT, UPDATE, DELETE, EXECUTE, REFERENCES) to a role and to grant a role to users. Use REVOKE to revoke them.

Example The following statement creates a role called "administrator."

```
CREATE ROLE administrator;
```

See Also GRANT, REVOKE, DROP ROLE

CREATE SHADOW

Creates one or more duplicate, in-sync copies of a database. Available in SQL, DSQL, and isql.

Syntax

```
CREATE SHADOW set_num [AUTO | MANUAL] [CONDITIONAL]
   'filespec' [LENGTH [=] int [PAGE[S]]]
   [<secondary_file>];

<secondary_file> = FILE 'filespec' [<fileinfo>] [<secondary_file>]

<fileinfo> = LENGTH [=] int [PAGE[S]] | STARTING [AT [PAGE]] int
   [<fileinfo>]
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
set_num	Positive integer that designates a shadow set to which all subsequent files listed in the statement belong
AUTO	Specifies the default access behavior for databases in the event no shadow is available ▪ All attachments and accesses succeed ▪ Deletes all references to the shadow and detaches the shadow file
MANUAL	Specifies that database attachments and accesses fail until a shadow becomes available, *or* until all references to the shadow are removed from the database
CONDITIONAL	Creates a new shadow, allowing shadowing to continue if the primary shadow becomes unavailable or if the shadow replaces the database due to disk failure
'*filespec*'	Explicit path name and file name for the shadow file; must be a local filesystem and must not include a node name or be on a neworked filesystem
LENGTH [=] *int* [PAGE[S]]	Length in database pages of an additional shadow file; page size is determined by the page size of the database itself
secondary_file	Specifies the length of a primary or secondary shadow file; use for primary file only if defining a secondary file in the same statement
STARTING [AT [PAGE]] *int*	Starting page number at which a secondary shadow file begins

Description CREATE SHADOW is used to guard against loss of access to a database by establishing one or more copies of the database on secondary storage devices. Each copy of the database consists of one or more shadow files, referred to as a *shadow set*. Each shadow set is designated by a unique positive integer.

Disk shadowing has three components:

- A database to shadow.
- The RDB$FILES system table, which lists shadow files and other information about the database.
- A shadow set, consisting of one or more shadow files.

When CREATE SHADOW is issued, a shadow is established for the database most recently attached by an application. A shadow set can consist of one or multiple files. In case of disk failure, the database administrator (DBA) activates the disk shadow so that it can take the place of the database. If CONDITIONAL is specified, then when the DBA activates the disk shadow to replace an actual database, a new shadow is established for the database.

If a database is larger than the space available for a shadow on one disk, use the *secondary_file* option to define multiple shadow files. Multiple shadow files can be spread over several disks.

Tip To add a secondary file to an existing disk shadow, drop the shadow with DROP SHADOW and use CREATE SHADOW to recreate it with the desired number of files.

Examples The following isql statement creates a single, automatic shadow file for employee.gdb:

```
CREATE SHADOW 1 AUTO 'employee.shd';
```

The next isql statement creates a conditional, single, automatic shadow file for employee.gdb:

```
CREATE SHADOW 2 CONDITIONAL 'employee.shd' LENGTH 1000;
```

The following isql statements create a multiple-file shadow set for the employee.gdb database. The first statement specifies starting pages for the shadow files; the second statement specifies the number of pages for the shadow files.

```
CREATE SHADOW 3 AUTO
       'employee.sh1'
   FILE 'employee.sh2'
          STARTING AT PAGE 1000
   FILE 'employee.sh3'
          STARTING AT PAGE 2000;
CREATE SHADOW 4 MANUAL 'employee.sdw'
   LENGTH 1000
   FILE 'employee.sh1'
      LENGTH 1000
   FILE 'employee.sh2';
```

See Also DROP SHADOW

For more information about using shadows, see the *Operations Guide* or the

Data Definition Guide.

CREATE TABLE

Creates a new table in an existing database. Available in SQL, DSQL, and isql.

Syntax

```
CREATE TABLE table [EXTERNAL [FILE] 'filespec']
    (<col_def> [, <col_def> | <tconstraint> …]);

<col_def> = col {<datatype> | COMPUTED [BY] (<expr>) | domain}
    [DEFAULT {literal | NULL | USER}]
    [NOT NULL]
    [<col_constraint>]
    [COLLATE collation]

<datatype> =
    {SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION}[<array_dim>]
    | (DATE | TIME | TIMESTAMP}[<array_dim>]
    | {DECIMAL | NUMERIC} [(precision [, scale])] [<array_dim>]
    | {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR} [(int)]
        [<array_dim>] [CHARACTER SET charname]
    | {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
        [VARYING] [(int)] [<array_dim>]
    | BLOB [SUB_TYPE {int | subtype_name}] [SEGMENT SIZE int]
        [CHARACTER SET charname]
    | BLOB [(seglen [, subtype])]<array_dim> = [[x:]y [, [x:]y …]]

<expr> = A valid SQL expression that results in a single value.

<col_constraint> = [CONSTRAINT constraint]
    { UNIQUE
    | PRIMARY KEY
    | REFERENCES other_table [(other_col [, other_col …])]
        [ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
        [ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
    | CHECK (<search_condition>)}

<tconstraint> = [CONSTRAINT constraint]
    {{PRIMARY KEY | UNIQUE} (col [, col …])
    | FOREIGN KEY (col [, col …]) REFERENCES other_table
        [ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
        [ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
    | CHECK (<search_condition>)}

<search_condition> = <val> <operator> {<val> | (<select_one>)}
    | <val> [NOT] BETWEEN <val> AND <val>
    | <val> [NOT] LIKE <val> [ESCAPE <val>]
    | <val> [NOT] IN (<val> [, <val> …] | <select_list>)
    | <val> IS [NOT] NULL
    | <val> {>= | <=}
    | <val> [NOT] {= | < | >}
    | {ALL | SOME | ANY} (<select_list>)
    | EXISTS (<select_expr>)
    | SINGULAR (<select_expr>)
    | <val> [NOT] CONTAINING <val>
    | <val> [NOT] STARTING [WITH] <val>
    | (<search_condition>)
    | NOT <search_condition>
    | <search_condition> OR <search_condition>
```

```
   | <search_condition> AND <search_condition>

<val> = { col [<array_dim>] | :variable
   | <constant> | <expr> | <function>
   | udf ([<val> [, <val> …]])
   | NULL | USER | RDB$DB_KEY | ? }
   [COLLATE collation]

<constant> = num | 'string' | charsetname 'string'

<function> = COUNT (* | [ALL] <val> | DISTINCT <val>)
   | SUM ([ALL] <val> | DISTINCT <val>)
   | AVG ([ALL] <val> | DISTINCT <val>)
   | MAX ([ALL] <val> | DISTINCT <val>)
   | MIN ([ALL] <val> | DISTINCT <val>)
   | CAST (<val> AS <datatype>)
   | UPPER (<val>)
   | GEN_ID (generator, <val>)

<operator> = {= | < | > | <= | >= | !< | !> | <> | !=}

<select_one> = SELECT on a single column; returns exactly one value.

<select_list> = SELECT on a single column; returns zero or more values.

<select_expr> = SELECT on a list of values; returns zero or more values.
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Notes on the CREATE TABLE statement

- When declaring arrays, you must include the outermost brackets, shown below in bold. For example, the following statement creates a 5 by 5 two-dimensional array of strings, each of which is 6 characters long:

  ```
  my_array VARCHAR(6)[5,5]
  ```

  Use the colon (:) to specify an array with a starting point other than 1. The following example creates an array of integers that begins at 10 and ends at 20:

  ```
  my_array INTEGER[10:20]
  ```

- In SQL and isql, you cannot use *val* as a parameter placeholder (like "?").
- In DSQL and isql, *val* cannot be a variable.
- You cannot specify a COLLATE clause for Blob columns.
- *expr* is any complex SQL statement or equation that produces a single value.

Argument	**Description**
table	Name for the table; must be unique among table and procedure names in the database
EXTERNAL [FILE] '*filespec*'	Declares that data for the table under creation resides in a table or file outside the database; *filespec* is the complete file specification of the external file or table

col

Name for the table column; unique among column names in the table

datatype

SQL datatype for the column

COMPUTED [BY] (*expr*)

Specifies that the value of the column's data is calculated from *expr* at runtime and is therefore not allocated storage space in the database

- *expr* can be any arithmetic expression valid for the datatypes in the expression
- Any columns referenced in *expr* must exist before they can be used in *expr*
- *expr* cannot reference Blob columns
- *expr* must return a single value, and cannot return an array

domain

Name of an existing domain

DEFAULT

Specifies a default column value that is entered when no other entry is made; possible values are:

- *literal*: Inserts a specified string, numeric value, or date value
- NULL: Enters a NULL value
- USER: Enters the user name of the current user. Column must be of compatible text type to use the default

Defaults set at column level override defaults set at the domain level.

CONSTRAINT *constraint*

Name of a column or table constraint; the constraint name must be unique within the table

constraint_def

Specifies the kind of column constraint; valid options are UNIQUE, PRIMARY KEY, CHECK, and REFERENCES

REFERENCES

Specifies that the column values are derived from column values in another table; if you do not specify column names, InterBase looks for a column with the same name as the referencing column in the referenced table

ON DELETE | ON UPDATE

Used with REFERENCES: Changes a foreign key whenever the referenced primary key changes; valid options are:

- [Default] NO ACTION: Does not change the foreign key; may cause the primary key update to fail due to referential integrity checks
- CASCADE: For ON DELETE, deletes the corresponding foreign key; for ON UPDATE, updates the corresponding foreign key to the

	new value of the primary key ▪ SET NULL: Sets all the columns of the corresponding foreign key to NULL ▪ SET DEFAULT: Sets every column of the corresponding foreign key is set to its default value in effect when the referential integrity constraint is defined; when the default for a foreign column changes after the referential integrity constraint is defined, the change does not have an effect on the default value used in the referential integrity constraint
CHECK *search_condition*	An attempt to enter a new value in the column fails if the value does not meet the *search_condition*
COLLATE *collation*	Establishes a default sorting behavior for the column

Description CREATE TABLE establishes a new table, its columns, and integrity constraints in an existing database. The user who creates a table is the table's owner and has all privileges for it, including the ability to GRANT privileges to other users, triggers, and stored procedures.

- CREATE TABLE supports several options for defining columns:
  - Local columns specify the name and datatype for data entered into the column.
  - Computed columns are based on an expression. Column values are computed each time the table is accessed. If the datatype is not specified, InterBase calculates an appropriate one. Columns referenced in the expression must exist before the column can be defined.
  - Domain-based columns inherit all the characteristics of a domain, but the column definition can include a new default value, a NOT NULL attribute, additional CHECK constraints, or a collation clause that overrides the domain definition. It can also include additional column constraints.
  - The datatype specification for a CHAR, VARCHAR, or Blob text column definition can include a CHARACTER SET clause to specify a particular character set for the single column. Otherwise, the column uses the default database character set. If the database character set is changed, all columns subsequently defined have the new character set, but existing columns are not affected.
  - If you do not specify a default character set, the character set defaults to NONE. Using character set NONE means that there is no character set assumption for columns; data is stored and retrieved just as you originally entered it. You can load any character set into a column defined with NONE, but you cannot load that same data into another column that has been defined with a different character set. In this case, no transliteration is performed between the source and destination character sets, and errors may occur during assignment.
  - The COLLATE clause enables specification of a particular collation order for CHAR, VARCHAR, and Blob text datatypes. Choice of collation order is

restricted to those supported for the column's given character set, which is either the default character set for the entire database, or a different set defined in the CHARACTER SET clause as part of the datatype definition.

- NOT NULL is an attribute that prevents the entry of NULL or unknown values in column. NOT NULL affects all INSERT and UPDATE operations on a column.

Important A DECLARE TABLE must precede CREATE TABLE in embedded applications if the same SQL program both creates a table and inserts data in the table.

- The EXTERNAL FILE option creates a table whose data resides in an external file, rather than in the InterBase database. Use this option to:

  · Define an InterBase table composed of data from an external source, such as data in files managed by other operating systems or in non-database applications.

  · Transfer data to an existing InterBase table from an external file.

Referential integrity constraints

- You can define integrity constraints at the time you create a table. These constraints are rules that validate data entries by enforcing column-to-table and table-to-table relationships. They span all transactions that access the database and are automatically maintained by the system. CREATE TABLE supports the following integrity constraints:

- A PRIMARY KEY is one or more columns whose collective contents are guaranteed to be unique. A PRIMARY KEY column must also define the NOT NULL attribute. A table can have only one primary key.

- UNIQUE keys ensure that no two rows have the same value for a specified column or ordered set of columns. A unique column must also define the NOT NULL attribute. A table can have one or more UNIQUE keys. A UNIQUE key can be referenced by a FOREIGN KEY in another table.

- Referential constraints (REFERENCES) ensure that values in the specified columns (known as the *foreign key*) are the same as values in the referenced UNIQUE or PRIMARY KEY columns in another table. The UNIQUE or PRIMARY KEY columns in the referenced table must be defined before the REFERENCES constraint is added to the secondary table. REFERENCES has ON DELETE and ON UPDATE clauses that define the action on the foreign key when the referenced primary key is updated or deleted. The values for ON UPDATE and ON DELETE are as follows:

Action specified	**Effect on foreign key**
NO ACTION	[Default] The foreign key does not change. This may cause the primary key update or delete to fail due to referential integrity checks.
CASCADE	The corresponding foreign key is updated or deleted as appropriate to the new value of the primary key.
SET DEFAULT	Every column of the corresponding foreign key is set to its default value. If the default value of the foreign key is not found in the primary key, the update or delete

	on the primary key fails. The default value is the one in effect when the referential integrity constraint was defined. When the default for a foreign key column is changed after the referential integrity constraint is set up, the change does not have an effect on the default value used in the referential integrity constraint.
SET NULL	Every column of the corresponding foreign key is set to NULL.

- You can create a FOREIGN KEY reference to a table that is owned by someone else only if that owner has explicitly granted you REFERENCES privilege on that table. Any user who updates your foreign key table must have REFERENCES or SELECT privileges on the referenced primary key table.
- CHECK constraints enforce a *search_condition* that must be true for inserts or updates to the specified table. *search_condition* can require a combination or range of values or can compare the value entered with data in other columns.

  Note Specifying USER as the value for a *search_condition* references the login of the user who is attempting to write to the referenced table.
- Creating PRIMARY KEY and FOREIGN KEY constraints requires exclusive access to the database.
- For unnamed constraints, the system assigns a unique constraint name stored in the RDB$RELATION_CONSTRAINTS system table.

  Note Constraints are not enforced on expressions.

Examples The following isql statement creates a simple table with a PRIMARY KEY:

```
CREATE TABLE COUNTRY (COUNTRY COUNTRYNAME NOT NULL PRIMARY KEY,
   CURRENCY VARCHAR(10) NOT NULL);
```

The next isql statement creates both a column-level and a table-level UNIQUE constraint:

```
CREATE TABLE STOCK (
   MODEL SMALLINT NOT NULL UNIQUE,
   MODELNAME CHAR(10) NOT NULL,
   ITEMID INTEGER NOT NULL,
   CONSTRAINT MOD_UNIQUE UNIQUE (MODELNAME, ITEMID));
```

The following isql statemnent illustrates table-level PRIMARY KEY, FOREIGN KEY, and CHECK constraints. The PRIMARY KEY constraint is based on three columns. This example also illustrates creating an array column of VARCHAR.

```
CREATE TABLE JOB (
   JOB_CODE JOBCODE NOT NULL,
   JOB_GRADE JOBGRADE NOT NULL,
   JOB_COUNTRY COUNTRYNAME NOT NULL,
   JOB_TITLE VARCHAR(25) NOT NULL,
   MIN_SALARY SALARY NOT NULL,
   MAX_SALARY SALARY NOT NULL,
   JOB_REQUIREMENT BLOB(400,1),
   LANGUAGE_REQ VARCHAR(15) [5],
   PRIMARY KEY (JOB_CODE, JOB_GRADE, JOB_COUNTRY),
   FOREIGN KEY (JOB_COUNTRY) REFERENCES COUNTRY (COUNTRY),
   CHECK (MIN_SALARY < MAX_SALARY));
```

In the next example, the F2 column in table T2 is a foreign key that references table T1 through T1's primary key P1. When a row in T1 changes, that change propagates to all affected rows in table T2. When a row in T1 is deleted, all affected rows in the F2 column of table T2 are set to NULL.

```
CREATE TABLE T1 (P1 INTEGER NOT NULL PRIMARY KEY);

CREATE TABLE T2 (F2 INTEGER FOREIGN KEY REFERENCES T1.P1
   ON UPDATE CASCADE
   ON DELETE SET NULL);
```

The next isql statement creates a table with a calculated column:

```
CREATE TABLE SALARY_HISTORY (
   EMP_NO EMPNO NOT NULL,
   CHANGE_DATE DATE DEFAULT 'NOW' NOT NULL,
   UPDATER_ID VARCHAR(20) NOT NULL,
   OLD_SALARY SALARY NOT NULL,
   PERCENT_CHANGE DOUBLE PRECISION
      DEFAULT 0
      NOT NULL
      CHECK (PERCENT_CHANGE BETWEEN -50 AND 50),
   NEW_SALARY COMPUTED BY
      (OLD_SALARY + OLD_SALARY * PERCENT_CHANGE / 100),
   PRIMARY KEY (EMP_NO, CHANGE_DATE, UPDATER_ID),
   FOREIGN KEY (EMP_NO) REFERENCES EMPLOYEE (EMP_NO));
```

In the following isql statement the first column retains the default collating order for the database's default character set. The second column has a different collating order, and the third column definition includes a character set and a collating order.

```
CREATE TABLE BOOKADVANCE (
   BOOKNO CHAR(6),
   TITLE CHAR(50) COLLATE ISO8859_1,
   EUROPUB CHAR(50) CHARACTER SET ISO8859_1 COLLATE FR_FR);
```

See Also CREATE DOMAIN, DECLARE TABLE, GRANT, REVOKE

For more information on creating metadata, using integrity constraints, external tables, datatypes, collation order, and character sets, see the *Data Definition Guide*.

CREATE TRIGGER

Creates a trigger, including when it fires, and what actions it performs. Available in DSQL, and isql.

Syntax

```
CREATE TRIGGER name FOR table
   [ACTIVE | INACTIVE]
   {BEFORE | AFTER}
   {DELETE | INSERT | UPDATE}
   [POSITION number]
   AS <trigger_body> terminator

<trigger_body> = [<variable_declaration_list>] <block>

<variable_declaration_list> =
   DECLARE VARIABLE variable <datatype>;
   [DECLARE VARIABLE variable <datatype>; …]

<block> =
BEGIN
   <compound_statement>
   [<compound_statement> …]
END

<datatype> = SMALLINT
   | INTEGER
   | FLOAT
   | DOUBLE PRECISION
   | {DECIMAL | NUMERIC} [(precision [, scale])]
   | {DATE | TIME | TIMESTAMP)
   | {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
      [(int)] [CHARACTER SET charname]
   | {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR} [VARYING] [(int)]

<compound_statement> = {<block> | statement;}
```

Argument	**Description**
name	Name of the trigger; must be unique in the database
table	Name of the table or view that causes the trigger to fire when the specified operation occurs on the table or view
ACTIVE|INACTIVE	Optional. Specifies trigger action at transaction end: ▪ ACTIVE: [Default] Trigger takes effect ▪ INACTIVE: Trigger does not take effect
BEFORE|AFTER	Required. Specifies whether the trigger fires: ▪ BEFORE: Before associated operation ▪ AFTER: After associated operation Associated operations are DELETE, INSERT, or UPDATE
DELETE|INSERT |UPDATE	Specifies the table operation that causes the trigger to fire

POSITION *number*	Specifies firing order for triggers before the same action or after the same action; *number* must be an integer between 0 and 32,767, inclusive. ▪ Lower-number triggers fire first ▪ Default: 0 = first trigger to fire ▪ Triggers for a table need not be consecutive; triggers on the same action with the same position number will fire in random order.
DECLARE VARIABLE *var datatype*	Declares local variables used only in the trigger. Each declaration must be preceded by DECLARE VARIABLE and followed by a semicolon (;). ▪ *var*: Local variable name, unique in the trigger ▪ *datatype*: The datatype of the local variable
statement	Any single statement in InterBase procedure and trigger language; each statement except BEGIN and END must be followed by a semicolon (;)
terminator	Terminator defined by the SET TERM statement; signifies the end of the trigger body. Used in isql only.

Description CREATE TRIGGER defines a new trigger to a database. A trigger is a self-contained program associated with a table or view that automatically performs an action when a row in the table or view is inserted, updated, or deleted.

A trigger is never called directly. Instead, when an application or user attempts to INSERT, UPDATE, or DELETE a row in a table, any triggers associated with that table and operation automatically execute, or *fire*. Triggers defined for UPDATE on non-updatable views fire even if no update occurs.

A trigger is composed of a *header* and a *body*.

The trigger header contains:

- A *trigger name*, unique within the database, that distinguishes the trigger from all others.
- A *table name*, identifying the table with which to associate the trigger.
- *Statements* that determine when the trigger fires.

The trigger body contains:

- An optional list of *local variables* and their datatypes.
- A *block* of statements in InterBase procedure and trigger language, bracketed by BEGIN and END. These statements are performed when the trigger fires. A block can itself include other blocks, so that there may be many levels of nesting.

Important Because each statement in the trigger body must be terminated by a semicolon, you must define a different symbol to terminate the trigger body itself. In isql, include a SET TERM statement before CREATE TRIGGER to specify a terminator other than a semicolon. After the body of the trigger, include another SET TERM to change the terminator back to a semicolon.

A trigger is associated with a table. The table owner and any user granted privileges to the table automatically have rights to execute associated triggers.

Triggers can be granted privileges on tables, just as users or procedures can be granted privileges. Use the GRANT statement, but instead of using TO *username*, use TO TRIGGER *trigger_nam*e. Triggers' privileges can be revoked similarly using REVOKE.

When a user performs an action that fires a trigger, the trigger will have privileges to perform its actions if one of the following conditions is true:

- The trigger has privileges for the action.
- The user has privileges for the action.

InterBase procedure and trigger language is a complete programming language for stored procedures and triggers. It includes:

- SQL data manipulation statements: INSERT, UPDATE, DELETE, and singleton SELECT.
- SQL operators and expressions, including generators and UDFs that are linked with the calling application.
- Powerful extensions to SQL, including assignment statements, control-flow statements, context variables, event-posting statements, exceptions, and error-handling statements.

The following table summarizes language extensions for triggers.

TABLE 4 Procedure and trigger language extensions

Statement	**Description**
BEGIN … END	Defines a block of statements that executes as one ▪ The BEGIN keyword starts the block; the END keyword terminates it ▪ Neither should be followed by a semicolon
variable = expression	Assignment statement that assigns the value of *expression* to *variable*, a local variable, input parameter, or output parameter
/* *comment_text* */	Programmer's comment, where *comment_text* can be any number of lines of text
EXCEPTION *exception_name*	Raises the named exception; an exception is a user-defined error that returns an error message to the calling

	application unless handled by a WHEN statement
EXECUTE PROCEDURE *proc_name* [*var* [, *var* …]] [RETURNING_VALUES *var* [, *var* …]]	Executes stored procedure, *proc_name,* with the listed input arguments ▪ Returns values in the listed output arguments following RETURNING_VALUES ▪ Input and output arguments must be local variables.
EXIT	Jumps to the final END statement in the procedure
FOR *select_statement* DO *compound_statement*	Repeats the statement or block following DO for every qualifying row retrieved by *select_statement*
select_statement	A normal SELECT statement, except that the INTO clause is required and must come last
compound_statement	Either a single statement in procedure and trigger language or a block of statements bracketed by BEGIN and END
IF (*condition*) THEN *compound_statement* [ELSE *compound_statement*]	Tests *condition*, and if it is TRUE, performs the statement or block following THEN; otherwise, performs the statement or block following ELSE, if present
condition	A Boolean expression (TRUE, FALSE, or UNKNOWN), generally two expressions as operands of a comparison operator
NEW.*column*	New context variable that indicates a new column value in an INSERT or UPDATE operation
OLD.*column*	Old context variable that indicates a column value before an UPDATE or DELETE operation
POST_EVENT *event_name* | *col*	Posts the event, *event_name*, or uses the value in *col* as an event name
WHILE (*condition*) DO *compound_statement*	While *condition* is TRUE, keep performing *compound_statement* ▪ *Tests condition*, andperforms *compound_statement* if *condition* is TRUE ▪ Repeats this sequence until *condition* is no longer TRUE
WHEN {*error* [, *error* …] | ANY}	Error-handling statement. When one of the specified errors occurs, performs

DO *compound_statement*	*compound_statement*. WHEN statements, if present, must come at the end of a block, just before END ▪ ANY: Handles any errors
error	EXCEPTION *exception_name,* SQLCODE *errcode* or GDSCODE *number*

Examples The following trigger, SAVE_SALARY_CHANGE, makes correlated updates to the SALARY_HISTORY table when a change is made to an employee's salary in the EMPLOYEE table:

```
SET TERM !! ;
CREATE TRIGGER SAVE_SALARY_CHANGE FOR EMPLOYEE
   AFTER UPDATE AS
   BEGIN
      IF (OLD.SALARY <> NEW.SALARY) THEN
      INSERT INTO SALARY_HISTORY
      (EMP_NO, CHANGE_DATE, UPDATER_ID, OLD_SALARY, PERCENT_CHANGE)
         VALUES (OLD.EMP_NO, 'now', USER,OLD.SALARY,
         (NEW.SALARY - OLD.SALARY) * 100 / OLD.SALARY);
   END !!
SET TERM ; !!
```

The following trigger, SET_CUST_NO, uses a generator to create unique customer numbers when a new customer record is inserted in the CUSTOMER table:

```
SET TERM !! ;
CREATE TRIGGER SET_CUST_NO FOR CUSTOMER
   BEFORE INSERT AS
   BEGIN
      NEW.CUST_NO = GEN_ID(CUST_NO_GEN, 1);
   END !!
SET TERM ; !!
```

The following trigger, POST_NEW_ORDER, posts an event named "new_order" whenever a new record is inserted in the SALES table:

```
SET TERM !! ;
CREATE TRIGGER POST_NEW_ORDER FOR SALES
   AFTER INSERT AS
   BEGIN
      POST_EVENT 'new_order';
   END !!
SET TERM ; !!
```

The following four fragments of trigger headers demonstrate how the POSITION option determines trigger firing order:

```
CREATE TRIGGER A FOR accounts
   BEFORE UPDATE
   POSITION 5 … /*Trigger body follows*/
```

```
CREATE TRIGGER B FOR accounts
   BEFORE UPDATE
   POSITION 0 … /*Trigger body follows*/
```

```
CREATE TRIGGER C FOR accounts
   AFTER UPDATE
   POSITION 5 … /*Trigger body follows*/
```

```
CREATE TRIGGER D FOR accounts
   AFTER UPDATE
   POSITION 3 … /*Trigger body follows*/
```

When this update takes place:

```
UPDATE accounts SET account_status = 'on_hold'
   WHERE account_balance <0;
```

The triggers fire in this order:

1. Trigger B fires.
2. Trigger A fires.
3. The update occurs.
4. Trigger D fires.
5. Trigger C fires.

See Also ALTER EXCEPTION, ALTER TRIGGER, CREATE EXCEPTION, CREATE PROCEDURE, DROP EXCEPTION, DROP TRIGGER, EXECUTE PROCEDURE

For more information on creating and using triggers, see the *Data Definition Guide*.

CREATE VIEW

Creates a new view of data from one or more tables. Available in SQL, DSQL, and isql.

Syntax

```
CREATE VIEW name [(view_col [, view_col …])]
   AS <select> [WITH CHECK OPTION];
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
name	Name for the view; must be unique among all view, table, and procedure names in the database
view_col	Names the columns for the view ▪ Column names must be unique among all column names in the view ▪ Required if the view includes columns based on expressions; otherwise optional ▪ Default: Column name from the underlying table
select	Specifies the selection criteria for rows to be included in the view
WITH CHECK OPTION	Prevents INSERT or UPDATE operations on an updatable view if the INSERT or UPDATE violates the search condition specified in the WHERE clause of the view's SELECT clause

Description CREATE VIEW describes a view of data based on one or more underlying tables in the database. The rows to return are defined by a SELECT statement that lists columns from the source tables. Only the view definition is stored in the database; a view does not directly represent physically stored data. It is possible to perform select, project, join, and union operations on views as if they were tables.

The user who creates a view is its owner and has all privileges for it, including the ability to GRANT privileges to other users, roles, triggers, views, and stored procedures. A user may have privileges to a view without having access to its base tables. When creating views:

- A read-only view requires SELECT privileges for any underlying tables.
- An updatable view requires ALL privileges to the underlying tables.

The *view_col* option ensures that the view always contains the same columns and that the columns always have the same view-defined names.

View column names correspond in order and number to the columns listed in the SELECT clause, so specify *all* view column names or none.

A *view_col* definition can contain one or more columns based on an expression that combines the outcome of two columns. The expression must return a single value, and cannot return an array or array element. If the view includes an expression, the *view-column* option is required.

Note Any columns used in the value expression must exist before the expression can be defined.

A SELECT statement clause cannot include the ORDER BY clause.

When SELECT * is used rather than a column list, order of display is based on the order in which columns are stored in the base table.

WITH CHECK OPTION enables InterBase to verify that a row added to or updated in a view is able to be seen through the view before allowing the operation to succeed. Do not use WITH CHECK OPTION for read-only views.

Note You cannot select from a view that is based on the result set of a stored procedure.

Note DSQL does not support view definitions containing UNION clauses. To create such a view, use embedded SQL.

A view is updatable if:

- It is a subset of a single table or another updatable view.
- All base table columns excluded from the view definition allow NULL values.
- The view's SELECT statement does not contain subqueries, a DISTINCT predicate, a HAVING clause, aggregate functions, joined tables, user-defined functions, or stored procedures.

If the view definition does not meet these conditions, it is considered read-only.

Note Read-only views can be updated by using a combination of user-defined referential constraints, triggers, and unique indexes.

Examples The following isql statement creates an updatable view:

```
CREATE VIEW SNOW_LINE (CITY, STATE, SNOW_ALTITUDE) AS
   SELECT CITY, STATE, ALTITUDE
      FROM CITIES
      WHERE ALTITUDE > 5000;
```

The next isql statement uses a nested query to create a view:

```
CREATE VIEW RECENT_CITIES AS
   SELECT STATE, CITY, POPULATION
      FROM CITIES WHERE STATE IN
         (SELECT STATE FROM STATES WHERE STATEHOOD > '1-JAN-1850');
```

In an updatable view, the WITH CHECK OPTION prevents any inserts or updates through the view that do not satisfy the WHERE clause of the CREATE VIEW SELECT statement:

```
CREATE VIEW HALF_MILE_CITIES AS
   SELECT CITY, STATE, ALTITUDE
      FROM CITIES
      WHERE ALTITUDE > 2500
      WITH CHECK OPTION;
```

The WITH CHECK OPTION clause in the view would prevent the following insertion:

```
INSERT INTO HALF_MILE_CITIES (CITY, STATE, ALTITUDE)
```

```
   VALUES ('Chicago', 'Illinois', 250);
```

On the other hand, the following UPDATE would be permitted:

```
INSERT INTO HALF_MILE_CITIES (CITY, STATE, ALTITUDE)
   VALUES ('Truckee', 'California', 2736);
```

The WITH CHECK OPTION clause does not allow updates through the view which change the value of a row so that the view cannot retrieve it. For example, the WITH CHECK OPTION in the HALF_MILE_CITIES view prevents the following update:

```
UPDATE HALF_MILE_CITIES
   SET ALTITUDE = 2000
   WHERE STATE = 'NY';
```

The next isql statement creates a view that joins two tables, and so is read-only:

```
CREATE VIEW PHONE_LIST AS
   SELECT EMP_NO, FIRST_NAME, LAST_NAME, PHONE_EXT, LOCATION, PHONE_NO
      FROM EMPLOYEE, DEPARTMENT
      WHERE EMPLOYEE.DEPT_NO = DEPARTMENT.DEPT_NO;
```

See Also CREATE TABLE, DROP VIEW, GRANT, INSERT, REVOKE, SELECT, UPDATE

For a complete discussion of views, see the *Data Definition Guide*.

DECLARE CURSOR

Defines a cursor for a table by associating a name with the set of rows specified in a SELECT statement. Available in SQL and DSQL.

Syntax SQL form:

```
DECLARE cursor CURSOR FOR <select> [FOR UPDATE OF <col> [, <col>…]];
```

DSQL form:

```
DECLARE cursor CURSOR FOR <statement_id>
```

Blob form: See DECLARE CURSOR (BLOB)

Argument	**Description**
cursor	Name for the cursor
select	Determines which rows to retrieve. SQL only
FOR UPDATE OF *col* [, *col …*]	Enables UPDATE and DELETE of specified column for retrieved rows
statement_id	SQL statement name of a previously prepared statement, which in this case must be a SELECT statement. DSQL only

Description DECLARE CURSOR defines the set of rows that can be retrieved using the cursor it names. It is the first member of a group of table cursor statements that must be used in sequence.

select specifies a SELECT statement that determines which rows to retrieve. The SELECT statement cannot include INTO or ORDER BY clauses.

The FOR UPDATE OF clause is necessary for updating or deleting rows using the WHERE CURRENT OF clause with UPDATE and DELETE.

A cursor is a one-way pointer into the ordered set of rows retrieved by the select expression in the DECLARE CURSOR statement. It enables sequential access to retrieved rows in turn. There are four related cursor statements:

Stage	**Statement**	**Purpose**
1	DECLARE CURSOR	Declares the cursor; the SELECT statement determines rows retrieved for the cursor
2	OPEN	Retrieves the rows specified for retrieval with DECLARE CURSOR; the resulting rows become the cursor's *active set*
3	FETCH	Retrieves the current row from the active set, starting with the first row; subsequent FETCH statements advance the cursor through the set
4	CLOSE	Closes the cursor and releases system resources

Examples The following embedded SQL statement declares a cursor with a search condition:

```
EXEC SQL
   DECLARE C CURSOR FOR
   SELECT CUST_NO, ORDER_STATUS
      FROM SALES
      WHERE ORDER_STATUS IN ('open', 'shipping');
```

The next DSQL statement declares a cursor for a previously prepared statement, QUERY1:

```
DECLARE Q CURSOR FOR QUERY1
```

See Also CLOSE, DECLARE CURSOR (BLOB), FETCH, OPEN, PREPARE, SELECT

DECLARE CURSOR (BLOB)

Declares a Blob cursor for read or insert. Available in SQL.

Syntax

```
DECLARE cursor CURSOR FOR
   {READ BLOB column FROM table
   | INSERT BLOB column INTO table}
   [FILTER [FROM subtype] TO subtype]
   [MAXIMUM_SEGMENT length];
```

Argument	**Description**
cursor	Name for the Blob cursor
column	Name of the Blob column
table	Table name
READ BLOB	Declares a read operation on the Blob
INSERT BLOB	Declares a write operation on the Blob
[FILTER [FROM *subtype*] TO *subtype*]	Specifies optional Blob filters used to translate a Blob from one user-specified format to another; *subtype* determines which filters are used for translation
MAXIMUM_ SEGMENT *length*	Length of the local variable to receive the Blob data after a FETCH operation

Description Declares a cursor for reading or inserting Blob data. A Blob cursor can be associated with only one Blob column.

To read partial Blob segments when a host-language variable is smaller than the segment length of a Blob, declare the Blob cursor with the MAXIMUM_SEGMENT clause. If *length* is less than the Blob segment, FETCH returns *length* bytes. If the same or greater, it returns a full segment (the default).

Examples The following embedded SQL statement declares a READ BLOB cursor and uses the MAXIMUM_SEGMENT option:

```
EXEC SQL
   DECLARE BC CURSOR FOR
   READ BLOB JOB_REQUIREMENT FROM JOB MAXIMUM_SEGMENT 40;
```

The next embedded SQL statement declares an INSERT BLOB cursor:

```
EXEC SQL
   DECLARE BC CURSOR FOR
   INSERT BLOB JOB_REQUIREMENt INTO JOB;
```

See Also CLOSE (BLOB), FETCH (BLOB), INSERT CURSOR (BLOB), OPEN (BLOB)

DECLARE EXTERNAL FUNCTION

Declares an existing user-defined function (UDF) to a database. Available in SQL, DSQL, and isql.

Syntax

```
DECLARE EXTERNAL FUNCTION name [datatype | CSTRING (int)
      [, datatype | CSTRING (int) …]]
   RETURNS {datatype [BY VALUE] | CSTRING (int)} [FREE_IT]
   ENTRY_POINT 'entryname'
   MODULE_NAME 'modulename';
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Note Whenever a UDF returns a value by reference to dynamically allocated memory, you must declare it using the FREE_IT keyword in order to free the allocated memory.

Argument	**Description**
name	Name of the UDF to use in SQL statements; can be different from the name of the function specified after the ENTRY_POINT keyword
datatype	Datatype of an input or return parameter ▪ All input parameters are passed to a UDF by reference ▪ Return parameters can be passed by value ▪ Cannot be an array element
CSTRING (*int*)	Specifies a UDF that returns a null-terminated string *int* bytes in length
RETURNS	Specifies the return value of a function
BY VALUE	Specifies that a return value should be passed by value rather than by reference
FREE_IT	Frees memory of the return value after the UDF finishes running ▪ Use only if the memory is allocated dynamically in the UDF ▪ See also *Language Reference*, Chapter 5
'*entryname*'	Quoted string specifying the name of the UDF in the source code as stored in the UDF library
'*modulename*'	Quoted file name identifying the library that contains the UDF; the placement of the library in the file system must meet one of the following criteria: ▪ The library is in *ib_install_dir*/UDF ▪ The library in a directory other than *ib_install_dir*/UDF and the complete pathname to the directory, including a drive letter in the case of a

Windows server, is listed in the InterBase configuration file.

- See the UDF chapter of the *Developer's Guide* for more about how InterBase finds the library

Description DECLARE EXTERNAL FUNCTION provides information about a UDF to a database: where to find it, its name, the input parameters it requires, and the single value it returns. Each UDF in a library must be declared once to each database where it will be used. As long as the entry point and module name do not change, there is no need to redelcare a UDF, even if the function itself is modified.

entryname is the actual name of the function as stored in the UDF library. It does not have to match the name of the UDF as stored in the database.

The module name does not need to include a path. However, the module must either be placed in ib_install_dir/UDF or must be listed in the InterBase configuration file.

To specify a location for UDF libraries in a configuration file, enter a line of the following form for Windows platforms:

```
EXTERNAL_FUNCTION_DIRECTORY D:\Mylibraries\InterBase
```

For UNIX and NetWare, the line does not include a drive letter:

```
EXTERNAL_FUNCTION_DIRECTORY \Mylibraries\InterBase
```

Note that beginning with InterBase 6, it is no longer sufficient to include a complete path name for the module in the DECLARE EXTERNAL FUNCTION statement. You *must* list the path in the InterBase configuration file if it is other than *ib_install_dir*/UDF. A path name is no longer useful in the DECLARE EXTERNAL FUNCTION statement.

The InterBase configuration file is called ibconfig on Windows machines, isc_config on UNIX machines, and isc_conf on Netware.

FOR NETWARE SERVERS Beginning with InterBase 5.6, the UDF library is statically linked to NetWare servers so that the UDFs are available as external functions. To make them available to a database on a Netware server, follow htese steps:

1. On a Windows client, connect to the database where you need the functions.
2. Run the ib_udf.sql script that is located in the *ib_install_dir*/Examples/UDF directory (InterBase 6 and later) or Examples/API directory (InterBase 5.6) using ISQL with the -i switch:

Examples The following isql statement declares the TOPS() UDF to a database:

```
DECLARE EXTERNAL FUNCTION TOPS
    CHAR(256), INTEGER, BLOB
    RETURNS INTEGER BY VALUE
    ENTRY_POINT 'te1' MODULE_NAME 'tm1.dll';
```

This example does not need the FREE_IT keyword because only cstrings, CHAR, and VARCHAR return types require memory allocation.

The next example declares the LOWERS() UDF and frees the memory allocated for the return value:

```
DECLARE EXTERNAL FUNCTION LOWERS VARCHAR(256)
   RETURNS CSTRING(256) FREE_IT
   ENTRY POINT 'fn_lower' MODULE_NAME 'udflib.dll';
```

See Also DROP EXTERNAL FUNCTION

For more information about writing UDFs, see "Working with UDFs" in the *Developer's Guide*.

DECLARE FILTER

Declares an existing Blob filter to a database. Available in SQL, DSQL, and isql.

Syntax

```
DECLARE FILTER filter
   INPUT_TYPE subtype OUTPUT_TYPE subtype
   ENTRY_POINT 'entryname' MODULE_NAME 'modulename';
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
filter	Name of the filter; must be unique among filter names in the database
INPUT_TYPE *subtype*	Specifies the Blob subtype from which data is to be converted
OUTPUT_TYPE *subtype*	Specifies the Blob subtype into which data is to be converted
'*entryname*'	Quoted string specifying the name of the Blob filter as stored in a linked library
'*modulename*'	Quoted file specification identifying the object module in which the filter is stored

Description DECLARE FILTER provides information about an existing Blob filter to the database: where to find it, its name, and the Blob subtypes it works with. A Blob filter is a user-written program that converts data stored in Blob columns from one subtype to another.

INPUT_TYPE and OUTPUT_TYPE together determine the behavior of the Blob filter. Each filter declared to the database should have a unique combination of INPUT_TYPE and OUTPUT_TYPE integer values. InterBase provides a built-in type of 1, for handling text. User-defined types must be expressed as negative values.

entryname is the name of the Blob filter stored in the library. When an application uses a Blob filter, it calls the filter function with this name.

Important Do not use DECLARE FILTER when creating a database on a NetWare server. Blob filters cannot be created or used on NetWare servers.

Example The following isql statement declares a Blob filter:

```
DECLARE FILTER DESC_FILTER
   INPUT_TYPE 1
   OUTPUT_TYPE -4
   ENTRY_POINT 'desc_filter'
   MODULE_NAME 'FILTERLIB';
```

See Also DROP FILTER

For instructions on writing Blob filters, see the *Embedded SQL Guide*.

For more information about Blob subtypes, see the *Data Definition Guide*.

DECLARE STATEMENT

Identifies dynamic SQL statements before they are prepared and executed in an embedded program. Available in SQL.

Syntax

```
DECLARE <statement> STATEMENT;
```

Argument	Description
statement	Name of an SQL variable for a user-supplied SQL statement to prepare and execute at runtime

Description DECLARE STATEMENT names an SQL variable for a user-supplied SQL statement to prepare and execute at run time. DECLARE STATEMENT is not executed, so it does not produce run-time errors. The statement provides internal documentation.

Example The following embedded SQL statement declares Q1 to be the name of a string for preparation and execution.

```
EXEC SQL
   DECLARE Q1 STATEMENT;
```

See Also EXECUTE, EXECUTE IMMEDIATE, PREPARE

DECLARE TABLE

Describes the structure of a table to the preprocessor, gpre, before it is created with CREATE TABLE. Available in SQL.

Syntax

```
DECLARE table TABLE (<table_def>);
```

Argument	Description
table	Name of the table; table names must be unique within the database
table_def	Definition of the table; for complete table definition syntax, see CREATE TABLE

Description DECLARE TABLE causes gpre to store a table description. You must use it if you both create and populate a table with data in the same program. If the declared table already exists in the database or if the declaration contains syntax errors, gpre returns an error.

When a table is referenced at run time, the column descriptions and datatypes are checked against the description stored in the database. If the table description is not in the database and the table is not declared, or if column descriptions and datatypes do not match, the application returns an error.

DECLARE TABLE can include an existing domain in a column definition, but must give the complete column description if the domain is not defined at compile time.

DECLARE TABLE cannot include integrity constraints and column attributes, even if they are present in a subsequent CREATE TABLE statement.

Important DECLARE TABLE cannot appear in a program that accesses multiple databases.

Example The following embedded SQL statements declare and create a table:

```
EXEC SQL
   DECLARE STOCK TABLE
   (MODEL SMALLINT,
   MODELNAME CHAR(10),
   ITEMID INTEGER);
EXEC SQL
   CREATE TABLE STOCK
   (MODEL SMALLINT NOT NULL UNIQUE,
   MODELNAME CHAR(10) NOT NULL,
   ITEMID INTEGER NOT NULL,
      CONSTRAINT MOD_UNIQUE UNIQUE (MODELNAME, ITEMID));
```

See Also CREATE DOMAIN, CREATE TABLE

DELETE

Removes rows in a table or in the active set of a cursor. Available in SQL, DSQL, and isql.

Syntax SQL and DSQL form:

Important Omit the terminating semicolon for DSQL.

```
DELETE [TRANSACTION transaction] FROM table
   {[WHERE <search_condition>] | WHERE CURRENT OF cursor};
```

```
<search_condition> = Search condition as specified in SELECT.
```

isql form:

```
DELETE FROM TABLE [WHERE <search_condition>];
```

Argument	**Description**
TRANSACTION *transaction*	Name of the transaction under control of which the statement is executed; SQL only
table	Name of the table from which to delete rows
WHERE *search_condition*	Search condition that specifies the rows to delete; without this clause, DELETE affects all rows in the specified table or view
WHERE CURRENT OF *cursor*	Specifies that the current row in the active set of *cursor* is to be deleted

DELETE specifies one or more rows to delete from a table or updatable view. DELETE is one of the database privileges controlled by the GRANT and REVOKE statements.

The TRANSACTION clause can be used in multiple transaction SQL applications to specify which transaction controls the DELETE operation. The TRANSACTION clause is not available in DSQL or isql.

For searched deletions, the optional WHERE clause can be used to restrict deletions to a subset of rows in the table.

Important Without a WHERE clause, a searched delete removes all rows from a table.

When performing a positioned delete with a cursor, the WHERE CURRENT OF clause must be specified to delete one row at a time from the active set.

Examples The following isql statement deletes all rows in a table:

```
DELETE FROM EMPLOYEE_PROJECT;
```

The next embedded SQL statement is a searched delete in an embedded application. It deletes all rows where a host-language variable equals a column value.

```
EXEC SQL
   DELETE FROM SALARY_HISTORY
   WHERE EMP_NO = :emp_num;
```

The following embedded SQL statements use a cursor and the WHERE CURRENT OF option to delete rows from CITIES with a population less than the host variable, *min_pop*. They declare and open a cursor that finds qualifying cities, fetch rows into the cursor, and delete the current row pointed to by the cursor.

```
EXEC SQL
   DECLARE SMALL_CITIES CURSOR FOR
   SELECT CITY, STATE
   FROM CITIES
   WHERE POPULATION < :min_pop;

EXEC SQL
   OPEN SMALL_CITIES;

EXEC SQL
   FETCH SMALL_CITIES INTO :cityname, :statecode;
   WHILE (!SQLCODE)
      {EXEC SQL
         DELETE FROM CITIES
         WHERE CURRENT OF SMALL_CITIES;
      EXEC SQL
         FETCH SMALL_CITIES INTO :cityname, :statecode;}
EXEC SQL
   CLOSE SMALL_CITIES;
```

See Also DECLARE CURSOR, FETCH, GRANT, OPEN, REVOKE, SELECT

For more information about using cursors, see the *Embedded SQL Guide*.

DESCRIBE

Provides information about columns that are retrieved by a dynamic SQL (DSQL) statement, or information about dynamic parameters that statement passes. Available in SQL.

Syntax

```
DESCRIBE [OUTPUT | INPUT] statement
   {INTO | USING} SQL DESCRIPTOR xsqlda;
```

Argument	Description
OUTPUT	[Default] Indicates that column information should be returned in the XSQLDA
INPUT	Indicates that dynamic parameter information should be stored in the XSQLDA
statement	A previously defined alias for the statement to DESCRIBE. ▪ Use PREPARE to define aliases
{INTO | USING} SQL DESCRIPTOR *xsqlda*	Specifies the XSQLDA to use for the DESCRIBE statement

Description DESCRIBE has two uses:

- As a *describe output* statement, DESCRIBE stores into an XSQLDA a description of the columns that make up the select list of a previously prepared statement. If the PREPARE statement included an INTO clause, it is unnecessary to use DESCRIBE as an output statement.
- As a *describe input* statement, DESCRIBE stores into an XSQLDA a description of the dynamic parameters that are in a previously prepared statement.

DESCRIBE is one of a group of statements that process DSQL statements.

Statement	Purpose
PREPARE	Readies a DSQL statement for execution
DESCRIBE	Fills in the XSQLDA with information about the statement
EXECUTE	Executes a previously prepared statement
EXECUTE IMMEDIATE	Prepares a DSQL statement, executes it once, and discards it

Separate DESCRIBE statements must be issued for input and output operations. The INPUT keyword must be used to store dynamic parameter information.

Important When using DESCRIBE for output, if the value returned in the *sqld* field in the XSQLDA is larger than the *sqln* field, you must:

- Allocate more storage space for XSQLVAR structures.
- Reissue the DESCRIBE statement.

Note The same XSQLDA structure can be used for input and output if desired.

Example The following embedded SQL statement retrieves information about the output of a SELECT statement:

```
EXEC SQL
   DESCRIBE Q INTO xsqlda
```

The next embedded SQL statement stores information about the dynamic parameters passed with a statement to be executed:

```
EXEC SQL
   DESCRIBE INPUT Q2 USING SQL DESCRIPTOR xsqlda;
```

See Also EXECUTE, EXECUTE IMMEDIATE, PREPARE

For more information about DSQL programming and the XSQLDA, see the *Embedded SQL Guide*.

DISCONNECT

Detaches an application from a database. Available in SQL.

Syntax

```
DISCONNECT {{ALL | DEFAULT} | dbhandle [, dbhandle] …]};
```

Argument	**Description**
ALL|DEFAULT	Either keyword detaches all open databases
dbhandle	Previously declared database handle specifying a database to detach

Description DISCONNECT closes a specific database identified by a database handle or all databases, releases resources used by the attached database, zeroes database handles, commits the default transaction if the gpre -manual option is not in effect, and returns an error if any non-default transaction is not committed.

Before using DISCONNECT, commit or roll back the transactions affecting the database to be detached.

To reattach to a database closed with DISCONNECT, reopen it with a CONNECT statement.

Examples The following embedded SQL statements close all databases:

```
EXEC SQL
   DISCONNECT DEFAULT;
```

```
EXEC SQL
   DISCONNECT ALL;
```

The next embedded SQL statements close the databases identified by their handles:

```
EXEC SQL
   DISCONNECT DB1;
```

```
EXEC SQL
   DISCONNECT DB1, DB2;
```

See Also COMMIT, CONNECT, ROLLBACK, SET DATABASE

DROP DATABASE

Deletes the currently attached database. Available in isql.

Syntax

```
DROP DATABASE;
```

Description DROP DATABASE deletes the currently attached database, including any associated secondary, shadow, and log files. Dropping a database deletes any data it contains.

A database can be dropped by its creator, the SYSDBA user, and any users with operating system root privileges.

Example The following isql statement deletes the current database:

```
DROP DATABASE;
```

See Also ALTER DATABASE, CREATE DATABASE

DROP DOMAIN

Deletes a domain from a database. Available in SQL, DSQL, and isql.

Syntax

```
DROP DOMAIN name;
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
name	Name of an existing domain

Description DROP DOMAIN removes an existing domain definition from a database.

If a domain is currently used in any column definition in the database, the DROP operation fails. To prevent failure, use ALTER TABLE to delete the columns based on the domain before executing DROP DOMAIN.

A domain may be dropped by its creator, the SYSDBA, and any users with operating system root privileges.

Example The following isql statement deletes a domain:

```
DROP DOMAIN COUNTRYNAME;
```

See Also ALTER DOMAIN, ALTER TABLE, CREATE DOMAIN

DROP EXCEPTION

Deletes an exception from a database. Available in DSQL and isql.

Syntax

```
DROP EXCEPTION name
```

Argument	Description
name	Name of an existing exception message

Description DROP EXCEPTION removes an exception from a database.

Exceptions used in existing procedures and triggers cannot be dropped.

Tip In isql, SHOW EXCEPTION displays a list of exceptions' *dependencies*, the procedures and triggers that use the exceptions.

An exception can be dropped by its creator, the SYSDBA user, and any user with operating system root privileges.

Example This isql statement drops an exception:

```
DROP EXCEPTION UNKNOWN_EMP_ID;
```

See Also ALTER EXCEPTION, ALTER PROCEDURE, ALTER TRIGGER, CREATE EXCEPTION, CREATE PROCEDURE, CREATE TRIGGER

DROP EXTERNAL FUNCTION

Removes a user-defined function (UDF) declaration from a database. Available in SQL, DSQL, and isql.

Syntax

```
DROP EXTERNAL FUNCTION name;
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
name	Name of an existing UDF

Description DROP EXTERNAL FUNCTION deletes a UDF declaration from a database. Dropping a UDF declaration from a database does *not* remove it from the corresponding UDF library, but it does make the UDF inaccessible from the database. Once the definition is dropped, any applications that depend on the UDF will return run-time errors.

A UDF can be dropped by its declarer, the SYSDBA user, or any users with operating system root privileges.

Important UDFs are not available for databases on NetWare servers. If a UDF is accidentally declared for a database on a NetWare server, DROP EXTERNAL FUNCTION should be used to remove the declaration.

Example This isql statement drops a UDF:

```
DROP EXTERNAL FUNCTION TOPS;
```

See Also DECLARE EXTERNAL FUNCTION

DROP FILTER

Removes a Blob filter declaration from a database. Available in SQL, DSQL, and isql.

Syntax

```
DROP FILTER name;
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
name	Name of an existing Blob filter

Description DROP FILTER removes a Blob filter declaration from a database. Dropping a Blob filter declaration from a database does *not* remove it from the corresponding Blob filter library, but it does make the filter inaccessible from the database. Once the definition is dropped, any applications that depend on the filter will return run-time errors.

DROP FILTER fails and returns an error if any processes are using the filter.

A filter can be dropped by its creator, the SYSDBA user, or any user with operating system root privileges.

Important Blob filters are not available for databases on NetWare servers. If a Blob filter is accidentally declared for a database on a NetWare server, DROP FILTER should be used to remove the declaration.

Example This isql statement drops a Blob filter:

```
DROP FILTER DESC_FILTER;
```

See Also DECLARE FILTER

DROP INDEX

Removes an index from a database. Available in SQL, DSQL, and isql.

Syntax

```
DROP INDEX name;
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
name	Name of an existing index

Description DROP INDEX removes a user-defined index from a database.

An index can be dropped by its creator, the SYSDBA user, or any user with operating system root privileges.

Important You cannot drop system-defined indexes, such as those for UNIQUE, PRIMARY KEY, and FOREIGN KEY.

An index in use is not dropped until it is no longer in use.

Example The following isql statement deletes an index:

```
DROP INDEX MINSALX;
```

See Also ALTER INDEX, CREATE INDEX

For more information about integrity constraints and system-defined indexes, see the *Data Definition Guide*.

DROP PROCEDURE

Deletes an existing stored procedure from a database. Available in DSQL, and isql.

Syntax

```
DROP PROCEDURE name
```

Argument	Description
name	Name of an existing stored procedure

Description DROP PROCEDURE removes an existing stored procedure definition from a database.

Procedures used by other procedures, triggers, or views cannot be dropped. Procedures currently in use cannot be dropped.

Tip In isql, SHOW PROCEDURE displays a list of procedures' *dependencies*, the procedures, triggers, exceptions, and tables that use the procedures.

A procedure can be dropped by its creator, the SYSDBA user, or any user with operating system root privileges.

Example The following isql statement deletes a procedure:

```
DROP PROCEDURE GET_EMP_PROJ;
```

See Also ALTER PROCEDURE, CREATE PROCEDURE, EXECUTE PROCEDURE

DROP ROLE

Deletes a role from a database. Available in SQL, DSQL, and isql.

Syntax

```
DROP ROLE rolename;
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
rolename	Name of an existing role

Description DROP ROLE deletes a role that was previously created using CREATE ROLE. Any privileges that users acquired or granted through their membership in the role are revoked.

A role can be dropped by its creator, the SYSDBA user, or any user with superuser privileges.

Example The following isql statement deletes a role from its database:

```
DROP ROLE administrator;
```

See Also CREATE ROLE, GRANT, REVOKE

DROP SHADOW

Deletes a shadow from a database. Available in SQL, DSQL, and isql.

Syntax

```
DROP SHADOW set_num;
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
set_num	Positive integer to identify an existing shadow set

Description DROP SHADOW deletes a shadow set and detaches from the shadowing process. The isql SHOW DATABASE command can be used to see shadow set numbers for a database.

A shadow can be dropped by its creator, the SYSDBA user, or any user with operating system root privileges.

Example The following isql statement deletes a shadow set from its database:

```
DROP SHADOW 1;
```

See Also CREATE SHADOW

DROP TABLE

Removes a table from a database. Available in SQL, DSQL, and isql.

Syntax

```
DROP TABLE name;
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
name	Name of an existing table

Description DROP TABLE removes a table's data, metadata, and indexes from a database. It also drops any triggers that reference the table.

A table referenced in an SQL expression, a view, integrity constraint, or stored procedure cannot be dropped. A table used by an active transaction is not dropped until it is free.

Note When used to drop an external table, DROP TABLE only removes the table definition from the database. The external file is not deleted.

A table can be dropped by its creator, the SYSDBA user, or any user with operating system root privileges.

Example The following embedded SQL statement drops a table:

```
EXEC SQL
   DROP TABLE COUNTRY;
```

See Also ALTER TABLE, CREATE TABLE

DROP TRIGGER

Deletes an existing user-defined trigger from a database. Available in DSQL and isql.

Syntax

```
DROP TRIGGER name
```

Argument	**Description**
name	Name of an existing trigger

Description DROP TRIGGER removes a user-defined trigger definition from the database. System-defined triggers, such as those created for CHECK constraints, cannot be dropped. Use ALTER TABLE to drop the CHECK clause that defines the trigger.

Triggers used by an active transaction cannot be dropped until the transaction is terminated.

A trigger can be dropped by its creator, the SYSDBA user, or any user with operating system root privileges.

Tip To inactivate a trigger temporarily, use ALTER TRIGGER and specify INACTIVE in the header.

Example The following isql statement drops a trigger:

```
DROP TRIGGER POST_NEW_ORDER;
```

See Also ALTER TRIGGER, CREATE TRIGGER

DROP VIEW

Removes a view definition from the database. Available in SQL, DSQL, and isql.

Syntax

```
DROP VIEW name;
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
name	Name of an existing view definition to drop

Description DROP VIEW enables a view's creator to remove a view definition from the database if the view is not used in another view, stored procedure, or CHECK constraint definition.

A view can be dropped by its creator, the SYSDBA user, or any user with operating system root privileges.

Example The following isql statement removes a view definition:

```
DROP VIEW PHONE_LIST;
```

See Also CREATE VIEW

END DECLARE SECTION

Identifies the end of a host-language variable declaration section. Available in SQL.

Syntax

```
END DECLARE SECTION;
```

Description END DECLARE SECTION is used in embedded SQL applications to identify the end of host-language variable declarations for variables used in subsequent SQL statements.

Example The following embedded SQL statements declare a section, and single host-language variable:

```
EXEC SQL
   BEGIN DECLARE SECTION;
      BASED_ON EMPLOYEE.SALARY salary;
EXEC SQL
   END DECLARE SECTION;
```

See Also BASED ON, BEGIN DECLARE SECTION

EVENT INIT

Registers interest in one or more events with the InterBase event manager. Available in SQL.

Syntax

```
EVENT INIT request_name [dbhandle]
   [('string' | :variable [, 'string' | :variable …]);
```

Argument	**Description**
request_name	Application event handle
dbhandle	Specifies the database to examine for occurrences of the events; if omitted, *dbhandle* defaults to the database named in the most recent SET DATABASE statement
'*string*'	Unique name identifying an event associated with *event_name*
:*variable*	Host-language character array containing a list of event names to associate with

Description EVENT INIT is the first step in the InterBase two-part synchronous event mechanism:

1. EVENT INIT registers an application's interest in an event.
2. EVENT WAIT causes the application to wait until notified of the event's occurrence.

EVENT INIT registers an application's interest in a list of events in parentheses. The list should correspond to events posted by stored procedures or triggers in the database. If an application registers interest in multiple events with a single EVENT INIT, then when one of those events occurs, the application must determine which event occurred.

Events are posted by a POST_EVENT call within a stored procedure or trigger.

The event manager keeps track of events of interest. At commit time, when an event occurs, the event manager notifies interested applications.

Example The following embedded SQL statement registers interest in an event:

```
EXEC SQL
   EVENT INIT ORDER_WAIT EMPDB ('new_order');
```

See Also CREATE PROCEDURE, CREATE TRIGGER, EVENT WAIT, SET DATABASE

For more information about events, see the *Embedded SQL Guide*.

EVENT WAIT

Causes an application to wait until notified of an event's occurrence. Available in SQL.

Syntax

```
EVENT WAIT request_name;
```

Argument	**Description**
request_name	Application event handle declared in a previous EVENT INIT statement

Description EVENT WAIT is the second step in the InterBase two-part synchronous event mechanism. After a program registers interest in an event, EVENT WAIT causes the process running the application to sleep until the event of interest occurs.

Examples The following embedded SQL statements register an application event name and indicate the program is ready to receive notification when the event occurs:

```
EXEC SQL
   EVENT INIT ORDER_WAIT EMPDB ('new_order');

EXEC SQL
   EVENT WAIT ORDER_WAIT;
```

See Also EVENT INIT

For more information about events, see the *Embedded SQL Guide*.

EXECUTE

Executes a previously prepared dynamic SQL (DSQL) statement. Available in SQL.

Syntax

```
EXECUTE [TRANSACTION transaction] statement
   [USING SQL DESCRIPTOR xsqlda] [INTO SQL DESCRIPTOR xsqlda];
```

Argument	**Description**
TRANSACTION *transaction*	Specifies the transaction under which execution occurs
statement	Alias of a previously prepared statement to execute
USING SQL DESCRIPTOR	Specifies that values corresponding to the prepared statement's parameters should be taken from the specified XSQLDA
INTO SQL DESCRIPTOR	Specifies that return values from the executed statement should be stored in the specified XSQLDA
xsqlda	XSQLDA host-language variable

Description EXECUTE carries out a previously prepared DSQL statement. It is one of a group of statements that process DSQL statements.

Statement	**Purpose**
PREPARE	Readies a DSQL statement for execution
DESCRIBE	Fills in the XSQLDA with information about the statement
EXECUTE	Executes a previously prepared statement
EXECUTE IMMEDIATE	Prepares a DSQL statement, executes it once, and discards it

Before a statement can be executed, it must be prepared using the PREPARE statement. The statement can be any SQL data definition, manipulation, or transaction management statement. Once it is prepared, a statement can be executed any number of times.

The TRANSACTION clause can be used in SQL applications running multiple, simultaneous transactions to specify which transaction controls the EXECUTE operation.

USING DESCRIPTOR enables EXECUTE to extract a statement's parameters from an XSQLDA structure previously loaded with values by the application. It need only be used for statements that have dynamic parameters.

INTO DESCRIPTOR enables EXECUTE to store return values from statement execution in a specified XSQLDA structure for application retrieval. It need only be used for DSQL statements that return values.

Note If an EXECUTE statement provides both a USING DESCRIPTOR clause and an INTO DESCRIPTOR clause, then two XSQLDA structures must be provided.

Example The following embedded SQL statement executes a previously prepared DSQL statement:

```
EXEC SQL
   EXECUTE DOUBLE_SMALL_BUDGET;
```

The next embedded SQL statement executes a previously prepared statement with parameters stored in an XSQLDA:

```
EXEC SQL
   EXECUTE Q USING DESCRIPTOR xsqlda;
```

The following embedded SQL statement executes a previously prepared statement with parameters in one XSQLDA, and produces results stored in a second XSQLDA:

```
EXEC SQL
   EXECUTE Q USING DESCRIPTOR xsqlda_1 INTO DESCRIPTOR xsqlda_2;
```

See Also DESCRIBE, EXECUTE IMMEDIATE, PREPARE

For more information about DSQL programming and the XSQLDA, see the *Embedded SQL Guide*.

EXECUTE IMMEDIATE

Prepares a dynamic SQL (DSQL) statement, executes it once, and discards it. Available in SQL.

Syntax

```
EXECUTE IMMEDIATE [TRANSACTION transaction]
   {:variable | 'string'} [USING SQL DESCRIPTOR xsqlda];
```

Argument	**Description**
TRANSACTION *transaction*	Specifies the transaction under which execution occurs
:*variable*	Host variable containing the SQL statement to execute
'*string*'	A string literal containing the SQL statement to execute
USING SQL DESCRIPTOR	Specifies that values corresponding to the statement's parameters should be taken from the specified XSQLDA
xsqlda	XSQLDA host-language variable

Description EXECUTE IMMEDIATE prepares a DSQL statement stored in a host-language variable or in a literal string, executes it once, and discards it. To prepare and execute a DSQL statement for repeated use, use PREPARE and EXECUTE instead of EXECUTE IMMEDIATE.

The TRANSACTION clause can be used in SQL applications running multiple, simultaneous transactions to specify which transaction controls the EXECUTE IMMEDIATE operation.

The SQL statement to execute must be stored in a host variable or be a string literal. It can contain any SQL data definition statement or data manipulation statement that does not return output.

USING DESCRIPTOR enables EXECUTE IMMEDIATE to extract the values of a statement's parameters from an XSQLDA structure previously loaded with appropriate values.

Example The following embedded SQL statement prepares and executes a statement in a host variable:

```
EXEC SQL
   EXECUTE IMMEDIATE :insert_date;
```

See Also DESCRIBE, EXECUTE IMMEDIATE, PREPARE

For more information about DSQL programming and the XSQLDA, see the *Embedded SQL Guide*.

EXECUTE PROCEDURE

Calls a stored procedure. Available in SQL, DSQL, and isql.

Syntax SQL form:

```
EXECUTE PROCEDURE [TRANSACTION transaction]
   name [:param [[INDICATOR]:indicator]]
      [, :param [[INDICATOR]:indicator] …]
   [RETURNING_VALUES :param [[INDICATOR]:indicator]
      [, :param [[INDICATOR]:indicator] …]];
```

DSQL form:

```
EXECUTE PROCEDURE name [param [, param …]]
   [RETURNING_VALUES param [, param …]]
```

isql form:

```
EXECUTE PROCEDURE name [param [, param …]]
```

Argument	**Description**
TRANSACTION *transaction*	Specifies the transaction under which execution occurs
name	Name of an existing stored procedure in the database
param	Input or output parameter; can be a host variable or a constant
RETURNING_VALUES : *param*	Host variable which takes the values of an output parameter
[INDICATOR] :*indicator*	Host variable for indicating NULL or unknown values

Description EXECUTE PROCEDURE calls the specified stored procedure. If the procedure requires input parameters, they are passed as host-language variables or as constants. If a procedure returns output parameters to an SQL program, host variables must be supplied in the RETURNING_VALUES clause to hold the values returned.

In isql, do not use the RETURN clause or specify output parameters. isql will automatically display return values.

Note In DSQL, an EXECUTE PROCEDURE statement requires an input descriptor area if it has input parameters and an output descriptor area if it has output parameters.

In embedded SQL, input parameters and return values may have associated indicator variables for tracking NULL values. Indicator variables are integer values that indicate unknown or NULL values of return values.

An indicator variable that is less than zero indicates that the parameter is unknown or NULL. An indicator variable that is zero or greater indicates that the associated parameter is known and not NULL.

Examples The following embedded SQL statement demonstrates how the executable procedure, DEPT_BUDGET, is called from embedded SQL with literal parameters:

```
EXEC SQL
   EXECUTE PROCEDURE DEPT_BUDGET 100 RETURNING_VALUES :sumb;
```

The next embedded SQL statement calls the same procedure using a host variable instead of a literal as the input parameter:

```
EXEC SQL
   EXECUTE PROCEDURE DEPT_BUDGET :rdno RETURNING_VALUES :sumb;
```

See Also ALTER PROCEDURE, CREATE PROCEDURE, DROP PROCEDURE

For more information about indicator variables, see the *Embedded SQL Guide*.

FETCH

Retrieves the next available row from the active set of an opened cursor. Available in SQL and DSQL.

Syntax SQL form:

```
FETCH cursor
   [INTO :hostvar [[INDICATOR] :indvar]
   [, :hostvar [[INDICATOR] :indvar] …]];
```

DSQL form:

```
FETCH cursor {INTO | USING} SQL DESCRIPTOR xsqlda
```

Blob form: See FETCH (BLOB).

Argument	Description
cursor	Name of the opened cursor from which to fetch rows
:*hostvar*	A host-language variable for holding values retrieved with the FETCH ▪ Optional if FETCH gets rows for DELETE or UPDATE ▪ Required if row is displayed before DELETE or UPDATE
:*indvar*	Indicator variable for reporting that a column contains an unknown or NULL value
[INTO|USING] SQL DESCRIPTOR	Specifies that values should be returned in the specified XSQLDA
xsqlda	XSQLDA host-language variable

Description FETCH retrieves one row at a time into a program from the active set of a cursor. The first FETCH operates on the first row of the active set. Subsequent FETCH statements advance the cursor sequentially through the active set one row at a time until no more rows are found and SQLCODE is set to 100.

A cursor is a one-way pointer into the ordered set of rows retrieved by the select expression in the DECLARE CURSOR statement. A cursor enables sequential access to retrieved rows. There are four related cursor statements:

Stage	Statement	Purpose
1	DECLARE CURSOR	Declare the cursor; the SELECT statement determines rows retrieved for the cursor
2	OPEN	Retrieve the rows specified for retrieval with DECLARE CURSOR; the resulting rows become the cursor's *active set*

3	FETCH	Retrieve the current row from the active set, starting with the first row; subsequent FETCH statements advance the cursor through the set
4	CLOSE	Close the cursor and release system resources

The number, size, datatype, and order of columns in a FETCH must be the same as those listed in the query expression of its matching DECLARE CURSOR statement. If they are not, the wrong values can be assigned.

Examples The following embedded SQL statement fetches a column from the active set of a cursor:

```
EXEC SQL
   FETCH PROJ_CNT INTO :department, :hcnt;
```

See Also CLOSE, DECLARE CURSOR, DELETE, FETCH (BLOB), OPEN

For more information about cursors and XSQLDA, see the *Embedded SQL Guide*.

FETCH (BLOB)

Retrieves the next available segment of a Blob column and places it in the specified local buffer. Available in SQL.

Syntax

```
FETCH cursor INTO
   [:<buffer> [[INDICATOR] :segment_length];
```

Argument	**Description**
cursor	Name of an open Blob cursor from which to retrieve segments
:*buffer*	Host-language variable for holding segments fetched from the Blob column; user must declare the buffer before fetching segments into it
INDICATOR	Optional keyword indicating that a host-language variable for indicating the number of bytes returned by the FETCH follows
:*segment_length*	Host-language variable used to indicate he number of bytes returned by the FETCH

Description FETCH retrieves the next segment from a Blob and places it into the specified buffer.

The host variable, *segment_length,* indicates the number of bytes fetched. This is useful when the number of bytes fetched is smaller than the host variable, for example, when fetching the last portion of a Blob.

FETCH can return two SQLCODE values:

- SQLCODE = 100 indicates that there are no more Blob segments to retrieve.
- SQLCODE = 101 indicates that a partial segment was retrieved and placed in the local buffer variable.

Note To ensure that a host variable buffer is large enough to hold a Blob segment buffer during FETCH operations, use the SEGMENT option of the BASED ON statement.

To ensure that a host variable buffer is large enough to hold a Blob segment buffer during FETCH operations, use the SEGMENT option of the BASED ON statement.

Example The following code, from an embedded SQL application, performs a BLOB FETCH:

```
while (SQLCODE != 100)
{
   EXEC SQL
      OPEN BLOB_CUR USING :blob_id;
   EXEC SQL
      FETCH BLOB_CUR INTO :blob_segment :blob_seg_len;
   while (SQLCODE !=100 || SQLCODE == 101)
   {
      blob_segment{blob_seg_len + 1] = '\0';
      printf("%*.*s",blob_seg_len,blob_seg_len,blob_segment);
         blob_segment{blob_seg_len + 1] = ' ';
      EXEC SQL
         FETCH BLOB_CUR INTO :blob_segment :blob_seg_len;
   }
   . . .
}
```

See Also BASED ON, CLOSE (BLOB), DECLARE CURSOR (BLOB), INSERT CURSOR (BLOB), OPEN (BLOB)

GEN_ID( )

Produces a system-generated integer value. Available in SQL, DSQL, and isql.

Syntax

```
gen_id (generator, step)
```

Argument	Description
generator	Name of an existing generator
step	Integer or expression specifying the increment for increasing or decreasing the current generator value. Values can range from –(2^{31}) to 2^{31} – 1

Description The GEN_ID() function:

1. Increments the current value of the specified generator by *step*.
2. Returns the new value of the specified generator.

GEN_ID() is useful for automatically producing unique values that can be inserted into a UNIQUE or PRIMARY KEY column. To insert a generated number in a column, write a trigger, procedure, or SQL statement that calls GEN_ID().

Note A generator is initially created with CREATE GENERATOR. By default, the value of a generator begins at zero. It can be set to a different value with SET GENERATOR.

Examples The following isql trigger definition includes a call to GEN_ID():

```
SET TERM !! ;
CREATE TRIGGER CREATE_EMPNO FOR EMPLOYEES
   BEFORE INSERT
   POSITION 0
   AS BEGIN
      NEW.EMPNO = GEN_ID (EMPNO_GEN, 1);
   END
SET TERM ; !!
```

The first time the trigger fires, NEW.EMPNO is set to 1. Each subsequent firing increments NEW.EMPNO by 1.

See Also CREATE GENERATOR, SET GENERATOR

GRANT

Assigns privileges to users for specified database objects. Available in SQL, DSQL, and isql.

```
GRANT <privileges> ON [TABLE] {tablename | viewname}
      TO {<object> | <userlist> | GROUP UNIX_group}
   | EXECUTE ON PROCEDURE procname TO {<object> | <userlist>}
   | <role_granted> TO {PUBLIC | <role_grantee_list>};

<privileges> = {ALL [PRIVILEGES] | <privilege_list>}

<privilege_list> = SELECT
   | DELETE
   | INSERT
   | UPDATE [(col [, col …])]
   | REFERENCES [(col [, col …])]
   [, <privilege_list> …]

<object> = PROCEDURE procname
   | TRIGGER trigname
   | VIEW viewname
   | PUBLIC
   [, <object> …]

<userlist> = [USER] username
   | rolename
   | Unix_user}
   [, <userlist> …]
   [WITH GRANT OPTION]

<role_granted> = rolename [, rolename …]

<role_grantee_list> = [USER] username [, [USER] username …]
   [WITH ADMIN OPTION]
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
privilege_list	Name of privilege to be granted; valid options are SELECT, DELETE, INSERT, UPDATE, and REFERENCES
col	Column to which the granted privileges apply
tablename	Name of an existing table for which granted privileges apply
viewname	Name of an existing view for which granted privileges apply
GROUP *unix_group*	On a UNIX system, the name of a group defined in /etc/group
object	Name of an existing procedure, trigger, or view; PUBLIC is also a permitted value

userlist	A user in isc4.gdb or a rolename created with CREATE ROLE
WITH GRANT OPTION	Passes GRANT authority for privileges listed in the GRANT statement to *userlist*
rolename	An existing role created with the CREATE ROLE statement
role_grantee_list	A list of users to whom *rolename* is granted; users must be in isc4.gdb
WITH ADMIN OPTION	Passes grant authority for roles listed to *role_grantee_list*

Description GRANT assigns privileges and roles for database objects to users, roles, or other database objects. When an object is first created, only its creator has privileges to it and only its creator can GRANT privileges for it to other users or objects.

- The following table summarizes available privileges:

Privilege	**Enables users to …**
ALL	Perform SELECT, DELETE, INSERT, UPDATE, and REFERENCES
SELECT	Retrieve rows from a table or view
DELETE	Remove rows from a table or view
INSERT	Store new rows in a table or view
UPDATE	Change the current value in one or more columns in a table or view; can be restricted to a specified subset of columns
EXECUTE	Execute a stored procedure
REFERENCES	Reference the specified columns with a foreign key; at a minimum, this must be granted to all the columns of the primary key if it is granted at all

Note ALL does not include REFERENCES in code written for InterBase 4.0 or earlier.

- To access a table or view, a user or object needs the appropriate SELECT, INSERT, UPDATE, DELETE, or REFERENCES privileges for that table or view. SELECT, INSERT, UPDATE, DELETE, and REFERENCES privileges can be assigned as a unit with ALL.
- A user or object must have EXECUTE privilege to call a stored procedure in an application.
- To grant privileges to a group of users, create a role using CREATE ROLE. Then use GRANT *privilege* TO *rolename* to assign the desired privileges to that role and use GRANT *rolename* TO *user* to assign that role to users. Users can be

added or removed from a role on a case-by-case basis using GRANT and REVOKE. A user must specify the role at connection time to actually have those privileges. See SQL roles on page 102 of the *Operations Guide* for more on invoking a role when connecting to a database.

- On UNIX systems, privileges can be granted to groups listed in /etc/groups and to any UNIX user listed in /etc/passwd on both the client and server, as well as to individual users and to roles.
- To allow another user to reference a columns from a foreign key, grant REFERENCES privileges on the primary key table or on the table's primary key columns to the owner of the foreign key table. You must also grant REFERENCES or SELECT privileges on the primary key table to any user who needs to write to the foreign key table.

Tip Make it easy: if read security is not an issue, GRANT REFERENCES on the primary key table to PUBLIC.

- If you grant the REFERENCES privilege, it must, at a minimum, be granted to all columns of the primary key. When REFERENCES is granted to the entire table, columns that are not part of the primary key are not affected in any way.
- When a user defines a foreign key constraint on a table owned by someone else, InterBase checks that that user has REFERENCES privileges on the referenced table.
- The privilege is used at runtime to verify that a value entered in a foreign key field is contained in the primary key table.
- You can grant REFERENCES privileges to roles.
- To give users permission to grant privileges to other users, provide a *userlist* that includes the WITH GRANT OPTION. Users can grant to others only the privileges that they themselves possess.
- To grant privileges to all users, specify PUBLIC in place of a list of user names. Specifying PUBLIC grants privileges only to users, not to database objects.

Privileges can be removed only by the user who assigned them, using REVOKE. If ALL privileges are assigned, then ALL privileges must be revoked. If privileges are granted to PUBLIC, they can be removed only for PUBLIC.

Examples The following isql statement grants SELECT and DELETE privileges to a user. The WITH GRANT OPTION gives the user GRANT authority.

```
GRANT SELECT, DELETE ON COUNTRY TO CHLOE WITH GRANT OPTION;
```

The next embedded SQL statement, from an embedded program, grants SELECT and UPDATE privileges to a procedure for a table:

```
EXEC SQL
   GRANT SELECT, UPDATE ON JOB TO PROCEDURE GET_EMP_PROJ;
```

This embedded SQL statement grants EXECUTE privileges for a procedure to another procedure and to a user:

```
EXEC SQL
   GRANT EXECUTE ON PROCEDURE GET_EMP_PROJ
   TO PROCEDURE ADD_EMP_PROJ, LUIS;
```

The following example creates a role called "administrator", grants UPDATE privileges on table1 to that role, and then grants the role to user1, user2, and user3. These users then have UPDATE and REFERENCES privileges on table1.

```
CREATE ROLE administrator;
GRANT UPDATE ON table1 TO administrator;
GRANT administrator TO user1, user2, user3;
```

See Also REVOKE

For more information about privileges, see the *Data Definition Guide*.

INSERT

Adds one or more new rows to a specified table. Available in SQL, DSQL, and isql.

Syntax

```
INSERT [TRANSACTION transaction] INTO <object> [(col [, col …])]
   {VALUES (<val> [, <val> …]) | <select_expr>};

<object> = tablename | viewname

<val> = {:variable | <constant> | <expr>
   | <function> | udf ([<val> [, <val> …]])
   | NULL | USER | RDB$DB_KEY | ?
   } [COLLATE collation]

<constant> = num | 'string' | charsetname 'string'

<function> = CAST (<val> AS <datatype>)
   | UPPER (<val>)
   | GEN_ID (generator, <val>)
```

Argument	**Description**
expr	A valid SQL expression that results in a single column value
select_expr	A SELECT that returns zero or more rows and where the number of columns in each row is the same as the number of items to be inserted

Notes on the INSERT statement

- In SQL and isql, you cannot use *val* as a parameter placeholder (like "?").
- In DSQL and isql, *val* cannot be a variable.
- You cannot specify a COLLATE clause for Blob columns.

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
TRANSACTION *transaction*	Name of the transaction that controls the execution of the INSERT
INTO *object*	Name of an existing table or view into which to insert data
col	Name of an existing column in a table or view into which to insert values
VALUES (*val* [, *val* …])	Lists values to insert into the table or view; values must be listed in the same order as the target columns
select_expr	Query that returns row values to insert into

target columns

Description INSERT stores one or more new rows of data in an existing table or view. INSERT is one of the database privileges controlled by the GRANT and REVOKE statements.

Values are inserted into a row in column order unless an optional list of target columns is provided. If the target list of columns is a subset of available columns, default or NULL values are automatically stored in all unlisted columns.

If the optional list of target columns is omitted, the VALUES clause must provide values to insert into all columns in the table.

To insert a single row of data, the VALUES clause should include a specific list of values to insert.

To insert multiple rows of data, specify a *select_expr* that retrieves existing data from another table to insert into this one. The selected columns must correspond to the columns listed for insert.

Important It is legal to select from the same table into which insertions are made, but this practice is not advised because it may result in infinite row insertions.

The TRANSACTION clause can be used in multiple transaction SQL applications to specify which transaction controls the INSERT operation. The TRANSACTION clause is not available in DSQL or isql.

Examples The following statement, from an embedded SQL application, adds a row to a table, assigning values from host-language variables to two columns:

```
EXEC SQL
   INSERT INTO EMPLOYEE_PROJECT (EMP_NO, PROJ_ID)
      VALUES (:emp_no, :proj_id);
```

The next isql statement specifies values to insert into a table with a SELECT statement:

```
INSERT INTO PROJECTS
   SELECT * FROM NEW_PROJECTS
   WHERE NEW_PROJECTS.START_DATE > '6-JUN-1994';
```

See Also GRANT, REVOKE, SET TRANSACTION, UPDATE

INSERT CURSOR (BLOB)

Inserts data into a Blob cursor in units of a Blob segment-length or less in size. Available in SQL.

Syntax

```
INSERT CURSOR cursor
   VALUES (:buffer [INDICATOR] :bufferlen);
```

Argument	**Description**
cursor	Name of the Blob cursor
VALUES	Clause containing the name and length of the buffer variable to insert
:*buffer*	Name of host-variable buffer containing information to insert
INDICATOR	Indicates that the length of data placed in the buffer follows
:*bufferlen*	Length, in bytes, of the buffer to insert

Description INSERT CURSOR writes Blob data into a column. Data is written in units equal to or less than the segment size for the Blob. Before inserting data into a Blob cursor:

- Declare a local variable, *buffer*, to contain the data to be inserted.
- Declare the length of the variable, *bufferlen*.
- Declare a Blob cursor for INSERT and open it.

Each INSERT into the Blob column inserts the current contents of *buffer*. Between statements fill *buffer* with new data. Repeat the INSERT until each existing *buffer* is inserted into the Blob.

Important INSERT CURSOR requires the INSERT privilege, a table privilege controlled by the GRANT and REVOKE statements.

Example The following embedded SQL statement shows an insert into the Blob cursor:

```
EXEC SQL
   INSERT CURSOR BC VALUES (:line INDICATOR :len);
```

See Also CLOSE (BLOB), DECLARE CURSOR (BLOB), FETCH (BLOB), OPEN (BLOB)

MAX( )

Retrieves the maximum value in a column. Available in SQL, DSQL, and isql.

Syntax

```
MAX ([ALL] <val> | DISTINCT <val>)
```

Argument	**Description**
ALL	Searches all values in a column
DISTINCT	Eliminates duplicate values before finding the largest
val	A column, constant, host-language variable, expression, non-aggregate function, or UDF

Description MAX() is an aggregate function that returns the largest value in a specified column, excluding NULL values. If the number of qualifying rows is zero, MAX() returns a NULL value.

When MAX() is used on a CHAR, VARCHAR, or Blob text column, the largest value returned varies depending on the character set and collation in use for the column. A default character set can be specified for an entire database with the DEFAULT CHARACTER SET clause in CREATE DATABASE, or specified at the column level with the COLLATE clause in CREATE TABLE.

Example The following embedded SQL statement demonstrates the use of SUM(), AVG(), MIN(), and MAX():

```
EXEC SQL
   SELECT SUM (BUDGET), AVG (BUDGET), MIN (BUDGET), MAX (BUDGET)
   FROM DEPARTMENT
   WHERE HEAD_DEPT = :head_dept
   INTO :tot_budget, :avg_budget, :min_budget, :max_budget;
```

See Also AVG( ), COUNT( ), CREATE DATABASE, CREATE TABLE, MIN( ), SUM( )

MIN( )

Retrieves the minimum value in a column. Available in SQL, DSQL, and isql.

Syntax

```
MIN ([ALL] <val> | DISTINCT <val>)
```

Argument	**Description**
ALL	Searches all values in a column
DISTINCT	Eliminates duplicate values before finding the smallest
val	A column, constant, host-language variable, expression, non-aggregate function, or UDF

Description MIN() is an aggregate function that returns the smallest value in a specified column, excluding NULL values. If the number of qualifying rows is zero, MIN() returns a NULL value.

When MIN() is used on a CHAR, VARCHAR, or Blob text column, the smallest value returned varies depending on the character set and collation in use for the column. Use the DEFAULT CHARACTER SET clause in CREATE DATABASE to specify a default character set for an entire database, or the COLLATE clause in CREATE TABLE to specify a character set at the column level.

Example The following embedded SQL statement demonstrates the use of SUM(), AVG(), MIN(), and MAX():

```
EXEC SQL
   SELECT SUM (BUDGET), AVG (BUDGET), MIN (BUDGET), MAX (BUDGET)
   FROM DEPARTMENT
   WHERE HEAD_DEPT = :head_dept
   INTO :tot_budget, :avg_budget, :min_budget, :max_budget;
```

See Also AVG( ), COUNT( ), CREATE DATABASE, CREATE TABLE, MAX( ), SUM( )

OPEN

Retrieve specified rows from a cursor declaration. Available in SQL and DSQL.

Syntax SQL form:

```
OPEN [TRANSACTION transaction] cursor;
```

DSQL form:

```
OPEN [TRANSACTION transaction] cursor [USING SQL DESCRIPTOR xsqlda]
```

Blob form: See OPEN (BLOB).

Argument	**Description**
TRANSACTION *transaction*	Name of the transaction that controls execution of OPEN
cursor	Name of a previously declared cursor to open
USING DESCRIPTOR *xsqlda*	Passes the values corresponding to the prepared statement's parameters through the extended descriptor area (XSQLDA)

Description OPEN evaluates the search condition specified in a cursor's DECLARE CURSOR statement. The selected rows become the *active set* for the cursor.

A cursor is a one-way pointer into the ordered set of rows retrieved by the SELECT in a DECLARE CURSOR statement. It enables sequential access to retrieved rows in turn. There are four related cursor statements:

Stage	**Statement**	**Purpose**
1	DECLARE CURSOR	Declares the cursor; the SELECT statement determines rows retrieved for the cursor
2	OPEN	Retrieves the rows specified for retrieval with DECLARE CURSOR; the resulting rows become the cursor's *active set*
3	FETCH	Retrieves the current row from the active set, starting with the first row ▪ Subsequent FETCH statements advance the cursor through the set
4	CLOSE	Closes the cursor and release system resources

Examples The following embedded SQL statement opens a cursor:

```
EXEC SQL
   OPEN C;
```

See Also CLOSE, DECLARE CURSOR, FETCH

OPEN (BLOB)

Opens a previously declared Blob cursor and prepares it for read or insert. Available in SQL.

Syntax

```
OPEN [TRANSACTION name] cursor
   {INTO | USING} :blob_id;
```

Argument	Description
TRANSACTION *name*	Specifies the transaction under which the cursor is opened Default: The default transaction
cursor	Name of the Blob cursor
INTO | USING	Depending on Blob cursor type, use one of these: INTO: For INSERT BLOB USING: For READ BLOB
blob_id	Identifier for the Blob column

Description OPEN prepares a previously declared cursor for reading or inserting Blob data. Depending on whether the DECLARE CURSOR statement declares a READ or INSERT BLOB cursor, OPEN obtains the value for Blob ID differently:

- For a READ BLOB, the *blob_id* comes from the outer TABLE cursor.
- For an INSERT BLOB, the *blob_id* is returned by the system.

Examples The following embedded SQL statements declare and open a Blob cursor:

```
EXEC SQL
   DECLARE BC CURSOR FOR
   INSERT BLOB PROJ_DESC INTO PRJOECT;

EXEC SQL
   OPEN BC INTO :blob_id;
```

See Also CLOSE (BLOB), DECLARE CURSOR (BLOB), FETCH (BLOB),INSERT CURSOR (BLOB)

PREPARE

Prepares a dynamic SQL (DSQL) statement for execution. Available in SQL.

Syntax

```
PREPARE [TRANSACTION transaction] statement
   [INTO SQL DESCRIPTOR xsqlda] FROM {:variable | 'string'};
```

Argument	**Description**
TRANSACTION *transaction*	Name of the transaction under control of which the statement is executed
statement	Establishes an alias for the prepared statement that can be used by subsequent DESCRIBE and EXCUTE statements
INTO *xsqlda*	Specifies an XSQLDA to be filled in with the description of the select-list columns in the prepared statement
:*variable* | '*string*'	DSQL statement to PREPARE; can be a host-language variable or a string literal

Description PREPARE readies a DSQL statement for repeated execution by:

- Checking the statement for syntax errors.
- Determining datatypes of optionally specified dynamic parameters.
- Optimizing statement execution.
- Compiling the statement for execution by EXECUTE.

PREPARE is part of a group of statements that prepare DSQL statements for execution.

Statement	**Purpose**
PREPARE	Readies a DSQL statement for execution
DESCRIBE	Fills in the XSQLDA with information about the statement
EXECUTE	Executes a previously prepared statement
EXECUTE IMMEDIATE	Prepares a DSQL statement, executes it once, and discards it

After a statement is prepared, it is available for execution as many times as necessary during the current session. To prepare and execute a statement only once, use EXECUTE IMMEDIATE.

statement establishes a symbolic name for the actual DSQL statement to prepare. It is *not* declared as a host-language variable. Except for C programs, gpre does not distinguish between uppercase and lowercase in *statement*, treating "B" and "b" as the same character. For C programs, use the gpre -either_case switch to activate case sensitivity during preprocessing.

If the optional INTO clause is used, PREPARE also fills in the extended SQL descriptor area (XSQLDA) with information about the datatype, length, and name of select-list columns in the prepared statement. This clause is useful only when the statement to prepare is a SELECT.

Note The DESCRIBE statement can be used instead of the INTO clause to fill in the XSQLDA for a select list.

The FROM clause specifies the actual DSQL statement to PREPARE. It can be a host-language variable, or a quoted string literal. The DSQL statement to PREPARE can be any SQL data definition, data manipulation, or transaction-control statement.

Examples The following embedded SQL statement prepares a DSQL statement from a host-variable statement. Because it uses the optional INTO clause, the assumption is that the DSQL statement in the host variable is a SELECT.

```
EXEC SQL
   PREPARE Q INTO xsqlda FROM :buf;
```

Note The previous statement could also be prepared and described in the following manner:

```
EXEC SQL
   PREPARE Q FROM :buf;

EXEC SQL
   DESCRIBE Q INTO SQL DESCRIPTOR xsqlda;
```

See Also DESCRIBE, EXECUTE, EXECUTE IMMEDIATE

REVOKE

Withdraws privileges from users for specified database objects. Available in SQL, DSQL, and isql.

```
REVOKE [GRANT OPTION FOR] <privileges> ON [TABLE]
          {tablename | viewname}
       FROM {<object> | <userlist> | <rolelist> | GROUP UNIX_group}
   | EXECUTE ON PROCEDURE procname
       FROM {<object> | <userlist>}
   | <role_granted> FROM {PUBLIC | <role_grantee_list>}};
```

```
<privileges> = {ALL [PRIVILEGES] | <privilege_list>}
```

```
<privilege_list> = {
   SELECT
   | DELETE
   | INSERT
   | UPDATE [(col [, col …])]
   | REFERENCES [(col [, col …])]
   [, <privilege_list> …]}}
```

```
<object> ={
   PROCEDURE procname
   | TRIGGER trigname
   | VIEW viewname
   | PUBLIC
   [, <object>]}
```

```
<userlist> = [USER] username [, [USER] username …]
```

```
<rolelist> = rolename [, rolename]
```

```
<role_granted> = rolename [, rolename …]
```

```
<role_grantee_list> = [USER] username [, [USER] username …]
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
privilege_list	Name of privilege to be granted; valid options are SELECT, DELETE, INSERT, UPDATE, and REFERENCES
GRANT OPTION FOR	Removes grant authority for privileges listed in the REVOKE statement from *userlist*; cannot be used with *object*
col	Column for which the privilege is revoked
tablename	Name of an existing table for which privileges are revoked
viewname	Name of an existing view for which privileges are revoked
GROUP	On a UNIX system, the name of a group defined

unix_group	in /etc/group
object	Name of an existing database object from which privileges are to be revoked
userlist	A list of users from whom privileges are to be revoked
rolename	An existing role created with the CREATE ROLE statement
role_grantee_list	A list of users to whom *rolename* is granted; users must be in isc4.gdb

Description REVOKE removes privileges from users or other database objects. Privileges are operations for which a user has authority. The following table lists SQL privileges:

TABLE 5 SQL privileges

Privilege	**Removes a user's privilege to …**
ALL	Perform SELECT, DELETE, INSERT, UPDATE, REFERENCES, and EXECUTE
SELECT	Retrieve rows from a table or view
DELETE	Remove rows from a table or view
INSERT	Store new rows in a table or view
UPDATE	Change the current value in one or more columns in a table or view; can be restricted to a specified subset of columns
REFERENCES	Reference the specified columns with a foreign key; at a minimum, this must be granted to all the columns of the primary key if it is granted at all
EXECUTE	Execute a stored procedure

GRANT OPTION FOR revokes a user's right to GRANT privileges to other users.

The following limitations should be noted for REVOKE:

- Only the user who grants a privilege can revoke that privilege.
- A single user can be assigned the same privileges for a database object by any number of other users. A REVOKE issued by a user only removes privileges previously assigned by that particular user.
- Privileges granted to all users with PUBLIC can only be removed by revoking privileges from PUBLIC.
- When a role is revoked from a user, all privileges that granted by that user to others because of authority gained from membership in the role are also revoked.

Examples

The following isql statement takes the SELECT privilege away from a user for a table:

```
REVOKE SELECT ON COUNTRY FROM MIREILLE;
```

The following isql statement withdraws EXECUTE privileges for a procedure from another procedure and a user:

```
REVOKE EXECUTE ON PROCEDURE GET_EMP_PROJ
   FROM PROCEDURE ADD_EMP_PROJ, LUIS;
```

See Also

GRANT

ROLLBACK

Restores the database to its state prior to the start of the current transaction. Available in SQL, DSQL, and isql.

Syntax

```
ROLLBACK [TRANSACTION name] [WORK] [RELEASE];
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
TRANSACTION *name*	Specifies the transaction to roll back in a multiple-transaction application [Default: roll back the default transaction]
WORK	Optional word allowed for compatibility
RELEASE	Detaches from all databases after ending the current transaction; SQL only

Description ROLLBACK undoes changes made to a database by the current transaction, then ends the transaction. It breaks the program's connection to the database and frees system resources. Use RELEASE in the last ROLLBACK to close all open databases. Wait until a program no longer needs the database to release system resources.

The TRANSACTION clause can be used in multiple-transaction SQL applications to specify which transaction to roll back. If omitted, the default transaction is rolled back. The TRANSACTION clause is not available in DSQL.

Note RELEASE, available only in SQL, detaches from all databases after ending the current transaction. In effect, this option ends database processing. RELEASE is supported for backward compatibility with older versions of InterBase. The preferred method of detaching is with DISCONNECT.

Examples The following isql statement rolls back the default transaction:

```
ROLLBACK;
```

The next embedded SQL statement rolls back a named transaction:

```
EXEC SQL
   ROLLBACK TRANSACTION MYTRANS;
```

See Also COMMIT, DISCONNECT

For more information about controlling transactions, see the *Embedded SQL Guide*.

SELECT

Retrieves data from one or more tables. Available in SQL, DSQL, and isql.

Syntax

```
SELECT [TRANSACTION transaction]
   [DISTINCT | ALL]
   {* | <val> [, <val> …]}
   [INTO :var [, :var …]]
   FROM <tableref> [, <tableref> …]
   [WHERE <search_condition>]
   [GROUP BY col [COLLATE collation] [, col [COLLATE collation] …]
   [HAVING <search_condition>]
   [UNION <select_expr> [ALL]]
   [PLAN <plan_expr>]
   [ORDER BY <order_list>]
   [FOR UPDATE [OF col [, col …]]];

<val> = {
   col [<array_dim>] | :variable
   | <constant> | <expr> | <function>
   | udf ([<val> [, <val> …]])
   | NULL | USER | RDB$DB_KEY | ?
   } [COLLATE collation] [AS alias]

<array_dim> = [[x:]y [, [x:]y …]]

<constant> = num | 'string' | charsetname 'string'

<function> = COUNT (* | [ALL] <val> | DISTINCT <val>)
   | SUM ([ALL] <val> | DISTINCT <val>)
   | AVG ([ALL] <val> | DISTINCT <val>)
   | MAX ([ALL] <val> | DISTINCT <val>)
   | MIN ([ALL] <val> | DISTINCT <val>)
   | CAST (<val> AS <datatype>)
   | UPPER (<val>)
   | GEN_ID (generator, <val>)

<tableref> = <joined_table> | table | view | procedure
   [(<val> [, <val> …])] [alias]

<joined_table> = <tableref> <join_type> JOIN <tableref>
   ON <search_condition> | (<joined_table>)

<join_type> = [INNER] JOIN
   | {LEFT | RIGHT | FULL } [OUTER]} JOIN

<search_condition> = <val> <operator> {<val> | (<select_one>)}
   | <val> [NOT] BETWEEN <val> AND <val>
   | <val> [NOT] LIKE <val> [ESCAPE <val>]
   | <val> [NOT] IN (<val> [, <val> …] | <select_list>)
   | <val> IS [NOT] NULL
   | <val> {>= | <=}
   | <val> [NOT] {= | < | >}
   | {ALL | SOME | ANY} (<select_list>)
   | EXISTS (<select_expr>)
   | SINGULAR (<select_expr>)
   | <val> [NOT] CONTAINING <val>
   | <val> [NOT] STARTING [WITH] <val>
   | (<search_condition>)
   | NOT <search_condition>
   | <search_condition> OR <search_condition>
   | <search_condition> AND <search_condition>

<operator> = {= | < | > | <= | >= | !< | !> | <> | !=}

<plan_expr> =
   [JOIN | [SORT] [MERGE]] ({<plan_item> | <plan_expr>}
   [, {<plan_item> | <plan_expr>} …])

<plan_item> = {table | alias}
   {NATURAL | INDEX (<index> [, <index> …])| ORDER <index>}

<order_list> =
   {col | int} [COLLATE collation]
      [ASC[ENDING] | DESC[ENDING]]
      [, <order_list> …]
```

Argument	**Description**
expr	A valid SQL expression that results in a single value
select_one	A SELECT on a single column that returns exactly one value
select_list	A SELECT on a single column that returns zero or more rows
select_expr	A SELECT on a list of values that returns zero or more rows

Notes on SELECT syntax

- When declaring arrays, you must include the outermost brackets, shown below in bold. For example, the following statement creates a 5 by 5 two-dimensional array of strings, each of which is 6 characters long:

  ```
  my_array = varchar(6)[5,5]
  ```

  Use the colon (:) to specify an array with a starting point other than 1. The following example creates an array of integers that begins at 10 and ends at 20:

  ```
  my_array = integer[20:30]
  ```

- In SQL and isql, you cannot use *val* as a parameter placeholder (like "?").
- In DSQL and isql, *val* cannot be a variable.
- You cannot specify a COLLATE clause for Blob columns.

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
TRANSACTION *transaction*	Name of the transaction under control of which the statement is executed; SQL only

SELECT [DISTINCT | ALL]	Specifies data to retrieve. DISTINCT prevents duplicate values from being returned. ALL, the default, retrieves every value
{*|*val* [, *val* …]}	The asterisk (*) retrieves all columns for the specified tables *val* [, *val* …] retrieves a list of specified columns, values, and expressions
INTO :*var* [, *var* …]	Singleton select in embedded SQL only; specifies a list of host-language variables into which to retrieve values
FROM *tableref* [, *tableref* …]	List of tables, views, and stored procedures from which to retrieve data; list can include joins and joins can be nested
table	Name of an existing table in a database
view	Name of an existing view in a database
procedure	Name of an existing stored procedure that functions like a SELECT statement
alias	Brief, alternate name for a table, view, or column; after declaration in *tableref*, *alias* can stand in for subsequent references to a table or view
joined_table	A table reference consisting of a JOIN
join_type	Type of join to perform. Default: INNER
WHERE *search_condition*	Specifies a condition that limits rows retrieved to a subset of all available rows
GROUP BY *col* [, *col* …]	Partitions the results of a query into groups containing all rows with identical values based on a column list
COLLATE *collation*	Specifies the collation order for the data retrieved by the query
HAVING *search_condition*	Used with GROUP BY; specifies a condition that limits grouped rows returned
UNION [ALL]	Combines two or more tables that are fully or partially identical in structure; the ALL option keeps identical rows separate instead of folding them together into one
PLAN *plan_expr*	Specifies the access plan for the InterBase optimizer to use during retrieval
plan_item	Specifies a table and index method for a plan
ORDER BY *order_list*	Specifies columns to order, either by column name or ordinal number in the query, and the order (ASC

or DESC) in which rows to return the rows

Description SELECT retrieves data from tables, views, or stored procedures. Variations of the SELECT statement make it possible to:

- Retrieve a single row, or part of a row, from a table. This operation is referred to as a *singleton select*.

  In embedded applications, all SELECT statements that occur outside the context of a cursor must be singleton selects.

- Retrieve multiple rows, or parts of rows, from a table.

  In embedded applications, multiple row retrieval is accomplished by embedding a SELECT within a DECLARE CURSOR statement.

  In isql, SELECT can be used directly to retrieve multiple rows.

- Retrieve related rows, or parts of rows, from a join of two or more tables.
- Retrieve all rows, or parts of rows, from union of two or more tables.

All SELECT statements consist of two required clauses (SELECT, FROM), and possibly others (INTO, WHERE, GROUP BY, HAVING, UNION, PLAN, ORDER BY). The following table explains the purpose of each clause, and when they are required:

TABLE 6 SELECT statement clauses

Clause	**Purpose**	**Singleton SELECT**	**Multi-row SELECT**
SELECT	Lists columns to retrieve	Required	Required
INTO	Lists host variables for storing retrieved columns	Required	Not allowed
FROM	Identifies the tables to search for values	Required	Required
WHERE	Specifies the search conditions used to restrict retrieved rows to a subset of all available rows; a WHERE clause can contain its own SELECT statement, referred to as a *subquery*	Optional	Optional
GROUP BY	Groups related rows based on common column values; used in conjunction with HAVING	Optional	Optional
HAVING	Restricts rows generated by GROUP BY to a subset of those rows	Optional	Optional
UNION	Combines the results of two or more SELECT statements to produce a single, dynamic table without duplicate rows	Optional	Optional

ORDER BY	Specifies which columns to order, either by column name or by ordinal number in the query, and the sort order of rows returned: ascending (ASC) [default] or descending (DESC)	Optional	Optional
PLAN	Specifies the query plan that should be used by the query optimizer instead of one it would normally choose	Optional	Optional
FOR UPDATE	Specifies columns listed after the SELECT clause of a DECLARE CURSOR statement that can be updated using a WHERE CURRENT OF clause	—	Optional

Because SELECT is such a ubiquitous and complex statement, a meaningful discussion lies outside the scope of this reference. To learn how to use SELECT in isql, see the *Operations Guide*. For a complete explanation of SELECT and its clauses, see the *Embedded SQL Guide*.

Examples The following isql statement selects columns from a table:

```
SELECT JOB_GRADE, JOB_CODE, JOB_COUNTRY, MAX_SALARY FROM PROJECT;
```

The next isql statement uses the * wildcard to select all columns and rows from a table:

```
SELECT * FROM COUNTRIES;
```

The following embedded SQL statement uses an aggregate function to count all rows in a table that satisfy a search condition specified in the WHERE clause:

```
EXEC SQL
   SELECT COUNT (*) INTO :cnt FROM COUNTRY
   WHERE POPULATION > 5000000;
```

The next isql statement establishes a table alias in the SELECT clause and uses it to identify a column in the WHERE clause:

```
SELECT C.CITY FROM CITIES C
   WHERE C.POPULATION < 1000000;
```

The following isql statement selects two columns and orders the rows retrieved by the second of those columns:

```
SELECT CITY, STATE FROM CITIES
   ORDER BY STATE;
```

The next isql statement performs a left join:

```
SELECT CITY, STATE_NAME FROM CITIES C
   LEFT JOIN STATES S ON S.STATE = C.STATE
   WHERE C.CITY STARTING WITH 'San';
```

The following isql statement specifies a query optimization plan for ordered retrieval, utilizing an index for ordering:

```
SELECT * FROM CITIES
   PLAN (CITIES ORDER CITIES_1);
   ORDER BY CITY
```

The next isql statement specifies a query optimization plan based on a three-way join with two indexed column equalities:

```
SELECT * FROM CITIES C, STATES S, MAYORS M
   WHERE C.CITY = M.CITY AND C.STATE = M.STATE
   PLAN JOIN (STATE NATURAL, CITIES INDEX DUPE_CITY,
   MAYORS INDEX MAYORS_1);
```

The next example queries two of the system tables, RDB$CHARACTER_SETS and RDB$COLLATIONS to display all the available character sets, their ID numbers, number of bytes per character, and collations. Note the use of ordinal column numbers in the ORDER BY clause.

```
SELECT RDB$CHARACTER_SET_NAME, RDB$CHARACTER_SET_ID,
      RDB$BYTES_PER_CHARACTER, RDB$COLLATION_NAME
   FROM RDB$CHARACTER_SETS JOIN RDB$COLLATIONS
   ON RDB$CHARACTER_SETS.RDB$CHARACTER_SET_ID =
      RDB%COLLATIONS.RDB$CHARACTER_SET_ID
   ORDER BY 1, 4;
```

See Also DECLARE CURSOR, DELETE, INSERT, UPDATE

For an introduction to using SELECT in isql, see the *Operations Guide*.

For a full discussion of data retrieval in embedded programming using DECLARE CURSOR and SELECT, see the *Embedded SQL Guide*.

SET DATABASE

Declares a database handle for database access. Available in SQL.

Syntax

```
SET {DATABASE | SCHEMA} dbhandle =
        [GLOBAL | STATIC | EXTERN][COMPILETIME][FILENAME] 'dbname'
    [USER 'name' PASSWORD 'string']
    [RUNTIME [FILENAME]
    {'dbname' | :var}
    [USER {'name' | :var} PASSWORD {'string' |:var}]];
```

Argument	**Description**
dbhandle	An alias for a specified database ▪ Must be unique within the program ▪ Used in subsequent SQL statements that support database handles
GLOBAL	[Default] Makes this database declaration available to all modules
STATIC	Limits scope of this database declaration to the current module
EXTERN	References a database declaration in another module, rather than actually declaring a new handle
COMPILETIME	Identifies the database used to look up column references during preprocessing ▪ If only one database is specified in SET DATABASE, it is used both at runtime and compiletime
'*dbname*'	Location and path name of the database associated with *dbhandle*; platform-specific
RUNTIME	Specifies a database to use at runtime if different than the one specified for use during preprocessing
:*var*	Host-language variable containing a database specification, user name, or password
USER '*name*'	A valid user name on the server where the database resides ▪ Used with PASSWORD to gain database access on the server ▪ Required for PC client attachments, optional for all others
PASSWORD '*string*'	A valid password on the server where the database resides ▪ Used with USER to gain database access on the server ▪ Required for PC client attachments,

optional for all others.

Description SET DATABASE declares a database handle for a specified database and associates the handle with that database. It enables optional specification of different compile-time and run-time databases. Applications that access multiple databases simultaneously must use SET DATABASE statements to establish separate database handles for each database.

dbhandle is an application-defined name for the database handle. Usually handle names are abbreviations of the actual database name. Once declared, database handles can be used in subsequent CONNECT, COMMIT, and ROLLBACK statements. They can also be used within transactions to differentiate table names when two or more attached databases contain tables with the same names.

dbname is a platform-specific file specification for the database to associate with *dbhandle*. It should follow the file syntax conventions for the server where the database resides.

GLOBAL, STATIC, and EXTERN are optional parameters that determine the scope of a database declaration. The default scope, GLOBAL, means that a database handle is available to all code modules in an application. STATIC limits database handle availability to the code module where the handle is declared. EXTERN references a global database handle in another module.

The optional COMPILETIME and RUNTIME parameters enable a single database handle to refer to one database when an application is preprocessed, and to another database when an application is run by a user. If omitted, or if only a COMPILETIME database is specified, InterBase uses the same database during preprocessing and at run time.

The USER and PASSWORD parameters are required for all PC client applications, but are optional for all other remote attachments. The user name and password are verified by the server in the security database before permitting remote attachments to succeed.

Examples The following embedded SQL statement declares a handle for a database:

```
EXEC SQL
   SET DATABASE DB1 = 'employee.gdb';
```

The next embedded SQL statement declares different databases at compile time and run time. It uses a host-language variable to specify the run-time database.

```
EXEC SQL
   SET DATABASE EMDBP = 'employee.gdb' RUNTIME :db_name;
```

See Also COMMIT, CONNECT, ROLLBACK, SELECT

For more information on the security database, see the *Operations Guide*.

SET GENERATOR

Sets a new value for an existing generator. Available in SQL, DSQL, and isql.

Syntax

```
SET GENERATOR name TO int;
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
name	Name of an existing generator
int	Value to which to set the generator, an integer from -2^{31} to $2^{31} - 1$

Description SET GENERATOR initializes a starting value for a newly created generator, or resets the value of an existing generator. A generator provides a unique, sequential numeric value through the GEN_ID() function. If a newly created generator is not initialized with SET GENERATOR, its starting value defaults to zero.

int is the new value for the generator. When the GEN_ID() function inserts or updates a value in a column, that value is *int* plus the increment specified in the GEN_ID() step parameter.

Tip To force a generator's first insertion value to 1, use SET GENERATOR to specify a starting value of 0, and set the step value of the GEN_ID() function to 1.

Important When resetting a generator that supplies values to a column defined with PRIMARY KEY or UNIQUE integrity constraints, be careful that the new value does not enable duplication of existing column values, or all subsequent insertions and updates will fail.

Example The following isql statement sets a generator value to 1,000:

```
SET GENERATOR CUST_NO_GEN TO 1000;
```

If GEN_ID() now calls this generator with a step value of 1, the first number it returns is 1,001.

See Also CREATE GENERATOR, CREATE PROCEDURE, CREATE TRIGGER, GEN_ID( )

SET NAMES

Specifies an active character set to use for subsequent database attachments. Available in SQL, and isql.

Syntax

```
SET NAMES [charset | :var];
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argumen t	**Description**
charset	Name of a character set that identifies the active character set for a given process; default: NONE
:*var*	Host variable containing string identifying a known character set name ▪ Must be declared as a character set name ▪ SQL only

Description SET NAMES specifies the character set to use for subsequent database attachments in an application. It enables the server to translate between the default character set for a database on the server and the character set used by an application on the client.

SET NAMES must appear before the SET DATABASE and CONNECT statements it is to affect.

Tip Use a host-language variable with SET NAMES in an embedded application to specify a character set interactively.

Choice of character sets limits possible collation orders to a subset of all available collation orders. Given a specific character set, a specific collation order can be specified when data is selected, inserted, or updated in a column.

Important If you do not specify a default character set, the character set defaults to NONE. Using character set NONE means that there is no character set assumption for columns; data is stored and retrieved just as you originally entered it. You can load any character set into a column defined with NONE, but you cannot load that same data into another column that has been defined with a different character set. No transliteration is performed between the source and destination character sets, so in most cases, errors occur during assignment.

Example The following statements demonstrate the use of SET NAMES in an embedded SQL application:

```
EXEC SQL
   SET NAMES ISO8859_1;

EXEC SQL
   SET DATABASE DB1 = 'employee.gdb';

EXEC SQL
```

```
   CONNECT;
```

The next statements demonstrate the use of SET NAMES in isql:

```
SET NAMES LATIN1;
CONNECT 'employee.gdb';
```

See Also CONNECT, SET DATABASE

For more information about character sets and collation orders, see the *Data Definition Guide*.

SET SQL DIALECT

Declares the SQL Dialect for database access. Available in gpre, isql, wisql, and SQL.

Syntax

```
SET SQL DIALECT n;
```

Argument	Description
n	The SQL Dialect type, either 1, 2, or 3

Description SET SQL DIALECT declares the SQL Dialect for database access.

n is the SQL Dialect type, either 1, 2, or 3.   If no dialect is specified, the default dialect is set to that of the specified compile-time database. If the default dialect is different than the one specified by the user, a warning is generated and the the default dialect is set to the user-specified value

TABLE 7 SQL Dialects

SQL Dialect	Used for
1	InterBase 5.5 and earlier compatibility
2	Transitional dialect used to flag changes when migrating from dialect 1 to dialect 3
3	InterBase 6.0; allows you to use delimited identifiers, exact numerics, and DATE, TIME, and TIMESTAMP datatypes

Examples The following embedded SQL statement sets the SQL Dialect to 3:

```
EXEC SQL
   SET SQL DIALECT 3;
```

See Also SHOW SQL DIALECT

SET STATISTICS

Recomputes the selectivity of a specified index. Available in SQL, DSQL, and isql.

Syntax

```
SET STATISTICS INDEX name;
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
name	Name of an existing index for which to recompute selectivity

Description SET STATISTICS enables the selectivity of an index to be recomputed. Index selectivity is a calculation, based on the number of distinct rows in a table, that is made by the InterBase optimizer when a table is accessed. It is cached in memory, where the optimizer can access it to calculate the optimal retrieval plan for a given query. For tables where the number of duplicate values in indexed columns radically increases or decreases, periodically recomputing index selectivity can improve performance.

Only the creator of an index can use SET STATISTICS.

Note SET STATISTICS does not rebuild an index. To rebuild an index, use ALTER INDEX.

Example The following embedded SQL statement recomputes the selectivity for an index:

```
EXEC SQL
   SET STATISTICS INDEX MINSALX;
```

See Also ALTER INDEX, CREATE INDEX, DROP INDEX

SET TRANSACTION

Starts a transaction and optionally specifies its behavior. Available in SQL, DSQL, and isql.

Syntax

```
SET TRANSACTION [NAME transaction]
   [READ WRITE | READ ONLY]
   [WAIT | NO WAIT]
   [[ISOLATION LEVEL] {SNAPSHOT [TABLE STABILITY]
      | READ COMMITTED [[NO] RECORD_VERSION]}]
   [RESERVING <reserving_clause>
      | USING dbhandle [, dbhandle …]];

<reserving_clause> = table [, table …]
   [FOR [SHARED | PROTECTED] {READ | WRITE}] [, <reserving_clause>]
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	**Description**
NAME *transaction*	Specifies the name for this transaction ▪ *transaction* is a previously declared and initialized host-language variable ▪ SQL only
READ WRITE	[Default] Specifies that the transaction can read and write to tables
READ ONLY	Specifies that the transaction can only read tables
WAIT	[Default] Specifies that a transaction wait for access if it encounters a lock conflict with another transaction
NO WAIT	Specifies that a transaction immediately return an error if it encounters a lock conflict
ISOLATION LEVEL	Specifies the isolation level for this transaction when attempting to access the same tables as other simultaneous transactions; default: SNAPSHOT
RESERVING *reserving_clause*	Reserves lock for tables at transaction start
USING *dbhandle* [, *dbhandle* …]	Limits database access to a subset of available databases; SQL only

Description SET TRANSACTION starts a transaction, and optionally specifies its database access, lock conflict behavior, and level of interaction with other concurrent transactions accessing the same data. It can also reserve locks for tables. As an alternative to reserving tables, multiple database SQL applications can restrict a transaction's access to a subset of connected databases.

Important Applications preprocessed with the gpre -manual switch must explicitly start each transaction with a SET TRANSACTION statement.

SET TRANSACTION affects the default transaction unless another transaction is specified in the optional NAME clause. Named transactions enable support for multiple, simultaneous transactions in a single application. All transaction names must be declared as host-language variables at compile time. In DSQL, this restriction prevents dynamic specification of transaction names.

By default a transaction has READ WRITE access to a database. If a transaction only needs to read data, specify the READ ONLY parameter.

When simultaneous transactions attempt to update the same data in tables, only the first update succeeds. No other transaction can update or delete that data until the controlling transaction is rolled back or committed. By default, transactions WAIT until the controlling transaction ends, then attempt their own operations. To force a transaction to return immediately and report a lock conflict error without waiting, specify the NO WAIT parameter.

ISOLATION LEVEL determines how a transaction interacts with other simultaneous transactions accessing the same tables. The default ISOLATION LEVEL is SNAPSHOT. It provides a repeatable-read view of the database at the moment the transaction starts. Changes made by other simultaneous transactions are not visible.

SNAPSHOT TABLE STABILITY provides a repeatable read of the database by ensuring that transactions cannot write to tables, though they may still be able to read from them.

READ COMMITTED enables a transaction to see the most recently committed changes made by other simultaneous transactions. It can also update rows as long as no update conflict occurs. Uncommitted changes made by other transactions remain invisible until committed. READ COMMITTED also provides two optional parameters:

- NO RECORD_VERSION, the default, reads only the latest version of a row. If the WAIT lock resolution option is specified, then the transaction waits until the latest version of a row is committed or rolled back, and retries its read.
- RECORD_VERSION reads the latest committed version of a row, even if more recent uncommitted version also resides on disk.

The RESERVING clause enables a transaction to register its desired level of access for specified tables when the transaction starts instead of when the transaction attempts its operations on that table. Reserving tables at transaction start can reduce the possibility of deadlocks.

The USING clause, available only in SQL, can be used to conserve system resources by limiting the number of databases a transaction can access.

Examples The following embedded SQL statement sets up the default transaction with an isolation level of READ COMMITTED. If the transaction encounters an update conflict, it waits to get control until the first (locking) transaction is committed or rolled back.

```
EXEC SQL
   SET TRANSACTION WAIT ISOLATION LEVEL READ COMMITTED;
```

The next embedded SQL statement starts a named transaction:

```
EXEC SQL
   SET TRANSACTION NAME T1 READ COMMITTED;
```

The following embedded SQL statement reserves three tables:

```
EXEC SQL
   SET TRANSACTION NAME TR1
   ISOLATION LEVEL READ COMMITTED
   NO RECORD_VERSION WAIT
   RESERVING TABLE1, TABLE2 FOR SHARED WRITE,
      TABLE3 FOR PROTECTED WRITE;
```

See Also COMMIT, ROLLBACK, SET NAMES

For more information about transactions, see the *Embedded SQL Guide*.

SHOW SQL DIALECT

Returns the current client SQL Dialect setting and the database SQL Dialect value. Available in gpre, isql, wisql, and SQL.

Syntax

```
SHOW SQL DIALECT;
```

Description SHOW SQL DIALECT returns the current client SQL Dialect setting and the database SQL Dialect value, either 1, 2, or 3.

TABLE 8 SQL Dialects

SQL Dialect	Used for
1	InterBase 5.5 and earlier compatibility
2	Transitional dialect used to flag changes when migrating from dialect 1 to dialect 3
3	InterBase 6.0; allows you to use delimited identifiers, exact numerics, and DATE, TIME, and TIMESTAMP datatypes

Examples The following embedded SQL statement returns the SQL Dialect:

```
EXEC SQL
   SHOW SQL DIALECT;
```

See Also SET SQL DIALECT

SUM( )

Totals the numeric values in a specified column. Available in SQL, DSQL, and isql.

Syntax

```
SUM ([ALL] <val> | DISTINCT <val>)
```

Argument	**Description**
ALL	Totals all values in a column
DISTINCT	Eliminates duplicate values before calculating the total
val	A column, constant, host-language variable, expression, non-aggregate function, or UDF that evaluates to a numeric datatype

Description SUM() is an aggregate function that calculates the sum of numeric values for a column. If the number of qualifying rows is zero, SUM() returns a NULL value.

Example The following embedded SQL statement demonstrates the use of SUM(), AVG(), MIN(), and MAX():

```
EXEC SQL
   SELECT SUM (BUDGET), AVG (BUDGET), MIN (BUDGET), MAX (BUDGET)
   FROM DEPARTMENT
   WHERE HEAD_DEPT = :head_dept
   INTO :tot_budget, :avg_budget, :min_budget, :max_budget;
```

See Also AVG( ), COUNT( ), MAX( ), MIN( )

UPDATE

Changes the data in all or part of an existing row in a table, view, or active set of a cursor. Available in SQL, DSQL, and isql.

Syntax SQL form:

```
UPDATE [TRANSACTION transaction] {table | view}
   SET col = <val> [, col = <val> …]
   [WHERE <search_condition> | WHERE CURRENT OF cursor];
```

DSQL and isql form:

```
UPDATE {table | view}
   SET col = <val> [, col = <val> …]
   [WHERE <search_condition>

<val> = {
   col [<array_dim>]
   | :variable
   | <constant>
   | <expr>
   | <function>
   | udf ([<val> [, <val> …]])
   | NULL
   | USER
   | ?}
   [COLLATE collation]

<array_dim> = [[x:]y [, [x:]y …]]

<constant> = num | 'string' | charsetname 'string'

<function> = CAST (<val> AS <datatype>)
   | UPPER (<val>)
   | GEN_ID (generator, <val>)
```

<expr> = A valid SQL expression that results in a single value.

<search_condition> = See CREATE TABLE for a full description.

Notes on the UPDATE statement

- In SQL and isql, you cannot use *val* as a parameter placeholder (like "?").
- In DSQL and isql, *val* cannot be a variable.
- You cannot specify a COLLATE clause for Blob columns.

Argument	**Description**
TRANSACTION *transaction*	Name of the transaction under control of which the statement is executed
table | *view*	Name of an existing table or view to update.
SET *col* = *val*	Specifies the columns to change and the values to assign to those columns
WHERE *search_condition*	Searched update only; specifies the conditions a row must meet to be modified

WHERE CURRENT OF *cursor*	Positioned update only; specifies that the current row of a cursor's active set is to be modified - Not available in DSQL and isql

Description UPDATE modifies one or more existing rows in a table or view. UPDATE is one of the database privileges controlled by GRANT and REVOKE.

For searched updates, the optional WHERE clause can be used to restrict updates to a subset of rows in the table. Searched updates cannot update array slices.

Important Without a WHERE clause, a searched update modifies all rows in a table.

When performing a positioned update with a cursor, the WHERE CURRENT OF clause must be specified to update one row at a time in the active set.

Note When updating a Blob column, UPDATE replaces the entire Blob with a new value.

Examples The following isql statement modifies a column for all rows in a table:

```
UPDATE CITIES
   SET POPULATION = POPULATION * 1.03;
```

The next embedded SQL statement uses a WHERE clause to restrict column modification to a subset of rows:

```
EXEC SQL
   UPDATE PROJECT
   SET PROJ_DESC = :blob_id
   WHERE PROJ_ID = :proj_id;
```

See Also DELETE, GRANT, INSERT, REVOKE, SELECT

UPPER( )

Converts a string to all uppercase. Available in SQL, DSQL, and isql.

Syntax

```
UPPER (<val>)
```

Argument	Description
val	A column, constant, host-language variable, expression, function, or UDF that evaluates to a character datatype

Description UPPER() converts a specified string to all uppercase characters. If applied to character sets that have no case differentiation, UPPER() has no effect.

Examples The following isql statement changes the name, BMatthews, to BMATTHEWS:

```
UPDATE EMPLOYEE
   SET EMP_NAME = UPPER (BMatthews)
   WHERE EMP_NAME = 'BMatthews';
```

The next isql statement creates a domain called PROJNO with a CHECK constraint that requires the value of the column to be all uppercase:

```
CREATE DOMAIN PROJNO
   AS CHAR(5)
   CHECK (VALUE = UPPER (VALUE));
```

See Also CAST( )

WHENEVER

Traps SQLCODE errors and warnings. Available in SQL.

Syntax

```
WHENEVER {NOT FOUND | SQLERROR | SQLWARNING}
   {GOTO label | CONTINUE};
```

Argument	**Description**
NOT FOUND	Traps SQLCODE = 100, no qualifying rows found for the executed statement
SQLERROR	Traps SQLCODE < 0, failed statement
SQLWARNING	Traps SQLCODE > 0 AND < 100, system warning or informational message
GOTO *label*	Jumps to program location specified by *label* when a warning or error occurs
CONTINUE	Ignores the warning or error and attempts to continue processing

Description WHENEVER traps for SQLCODE errors and warnings. Every executable SQL statement returns an SQLCODE value to indicate its success or failure. If SQLCODE is zero, statement execution is successful. A non-zero value indicates an error, warning, or not found condition.

If the appropriate condition is trapped for, WHENEVER can:

- Use GOTO *label* to jump to an error-handling routine in an application.
- Use CONTINUE to ignore the condition.

WHENEVER can help limit the size of an application, because the application can use a single suite of routines for handling all errors and warnings.

WHENEVER statements should precede any SQL statement that can result in an error. Each condition to trap for requires a separate WHENEVER statement. If WHENEVER is omitted for a particular condition, it is not trapped.

Tip Precede error-handling routines with WHENEVER … CONTINUE statements to prevent the possibility of infinite looping in the error-handling routines.

Example In the following code from an embedded SQL application, three WHENEVER statements determine which label to branch to for error and warning handling:

```
EXEC SQL
   WHENEVER SQLERROR GO TO Error; /* Trap all errors. */
EXEC SQL
   WHENEVER NOT FOUND GO TO AllDone; /* Trap SQLCODE = 100 */

EXEC SQL
   WHENEVER SQLWARNING CONTINUE; /* Ignore all warnings. */
```

For a complete discussion of error-handling methods and programming, see the *Embedded SQL Guide*.