## SQL statement and function reference

This reference describes syntax and usage for each InterBase SQL function and statement. It includes the following topics:

Statement list provides an alphabetical list of all included statements.

Function list provides an alphabetical list of all included functions.

Data types lists the data types available to SQL statements in InterBase.

Error handling lists values that are returned to SQLCODE.

Using statement and function definitions explains the elements of a definition.

**Note:** Some functions and statements can be used in DSQL or *isql*. Where such extended application is available, it is noted at the bottom of the Description area in the function or statement definition topic. If not so noted, the function or statement can only be used in SQL.

## Statement list

## Function list

| Function | Type | Purpose |
|---|---|---|
| AVG() | Aggregate | Calculates the average of a set of values. |
| CAST() | Conversion | Converts a column from one data type to another. |
| COUNT() | Aggregate | Returns the number of rows that satisfy a query's search condition. |
| GEN_ID() | Numeric | Returns a system-generated value. |
| MAX() | Aggregate | Retrieves the maximum value from a set of values. |
| MIN() | Aggregate | Retrieves the minimum value from a set of values. |
| SUM() | Aggregate | Totals the values in a set of numeric values. |
| UPPER() | Conversion | Converts a string to all uppercase. |

Aggregate functions perform calculations over a series of values, such as the columns retrieved with a SELECT statement.

Conversion functions transform data types, either converting them from one type to another, or by converting CHARACTER data types to all uppercase.

The numeric function, GEN_ID(), produces a system-generated number that can be inserted into a column requiring a numeric data type.

## Data types

InterBase supports most SQL data types, but does not directly support the SQL DATE, TIME, and TIMESTAMP data types. In addition to standard SQL data types, InterBase also supports binary large object (BLOB) data types, and arrays of data types (except for BLOB data). The following table lists the data types available to SQL statements in InterBase:

**Name:** BLOB **Size:** Variable

**Range/Precision:** None

**Description:** Binary large object. Stores large data, such as graphics, text, and digitized voice. Basic structural unit: segment. BLOB subtype describes BLOB contents.

---

**Name:** CHAR(*n)* **Size:** *n* characters

**Range/Precision:** 1 to 32767 bytes. Character set character size determines the maximum number of characters that can fit in 32K.

**Description:** Fixed length CHAR or text string type. Alternate keyword: CHARACTER

---

**Name:** DATE **Size:** 64 bits

**Range/Precision:** 1 Jan 100 to 11 Jan 5941

**Description:** Also includes time information.

---

**Name:** DECIMAL (*precision, scale)* **Size:** variable

**Range/Precision:** *precision* = 1 to 15. Specifies at least *precision* digits of precision to store. *scale* = 1 to 15. Specifies number of decimal places for storage. Must be less than or equal to *precision*.

**Description:** Number with a decimal point *scale* digits from the right. For example, DECIMAL(10, 3) holds numbers accurately in the following format:

ppppppp.sss

---

**Name:** DOUBLE PRECISION **Size:** 64 bits

**Range/Precision:** 1.7X10^-308 to 1.7X10^308

**Description:** Scientific: 15 digits of precision.

**Note:** Actual size of DOUBLE is platform dependent. Most platforms support the 64-bit size.

---

**Name:** FLOAT **Size:** 32 bits

**Range/Precision:** 3.4X10^-38 to 3.4X10^38

**Description:** Single precision: 7 digits of precision.

---

**Name:** INTEGER **Size:** 32 bits

**Range/Precision: -** 2,147,483,648 to 2,147,483,648

**Description:** Signed long (longword).

---

**Name:** NUMERIC (*precision, scale)* **Size:** variable

**Range/Precision:** *precision* = 1 to 15. Specifies exactly *precision* digits of precision to store. *scale* = 1 to 15. Specifies number of decimal places for storage. Must be less than or equal to *precision*.

**Description:** Number with a decimal point *scale* digits from the right. For example, NUMERIC(10,3) holds numbers accurately in the following format:

ppppppp.sss

---

**Name:** SMALLINT **Size:** 16 bits

**Range/Precision:** -32768 to 32767

**Description:** Signed short (word).

---

**Name:** VARCHAR (*n)* **Size:** *n* characters

**Range/Precision:** 1 to 32767 bytes. Character set character size determines the maximum number of characters that can fit in 32K.

**Description:** Variable length CHAR or text string type. Alternate keywords: VARYING CHAR, VARYING CHARACTER

## Error handling

Every time an executable SQL statement is executed, the SQLCODE variable is set to indicate its success or failure.

The following table lists values that are returned to SQLCODE:

| SQLCODE | Message | Meaning |
|---|---|---|
| < 0 | SQLERROR | Error occurred. Statement did not execute. |
| 0 | SUCCESS | Successful execution. |
| +1-99 | SQLWARNING | System warning or informational message. |
| +100 | NOT FOUND | No qualifying rows found, or end of current active set of rows reached. |

In ISQL when an error occurs, an error number and message is displayed.

## Using statement and function definitions

Each statement and function definition includes the following elements:

| Element | Description |
|---|---|
| Title | Statement name |
| Definition | The statement’s main purpose |
| Syntax | Diagram of the statement and its parameters |
| Arguments | Arguments available for use with the statement |
| Description | Information about using the statement |
| Examples | Examples of using the statement |
| See also | Where to find more information about the statement or others related to it |

## ALTER DATABASE

Syntax Example See also

### Description

ALTER DATABASE adds secondary files to an existing database. Secondary files are useful for controlling the growth and location of a database. They permit database files to be spread across storage devices, but must remain on the same node as the primary database file.

A database can be altered by its creator and the SYSDBA user.

ALTER DATABASE requires exclusive access to the database.

On NetWare only, ALTER DATABASE also enables you to modify the write-ahead log (WAL) protocol.

**See also**

CREATE DATABASE

DROP DATABASE

## ALTER DATABASE syntax

Example Definition

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

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

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

| 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. |

## ALTER DATABASE example

Syntax Definition

The following statement adds secondary files to an existing database:

```
ALTER DATABASE
    ADD FILE "employee.gd1"
    STARTING AT PAGE 10001
      LENGTH 10000
    ADD FILE "employee.gd2"
      LENGTH 10000;
```

## ALTER DOMAIN

Syntax Example See also

### Description

ALTER DOMAIN changes any aspect of an existing domain except its data type and 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 data type 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 and the SYSDBA user.

**See also**

CREATE DOMAIN

CREATE TABLE

DROP DOMAIN

## ALTER DOMAIN syntax

Example Definition

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

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

| 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:<br>*literal*–Inserts a specified string, numeric value, or date value.<br>NULL–Enters a NULL value.<br>USER–Enters the user name of the current user. Column must be of compatible text type to use the default.<br>Defaults set at column level override defaults set at the domain level. |
| DROP DEFAULT | Drops existing default. |
| ADD [CONSTRAINT] CHECK *<dom_search_cond>* | 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. |

## ALTER DOMAIN example

Syntax Definition

The following statements create a domain that must have a value > 1000, then alter it by setting a default of 9999:

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

## ALTER EXCEPTION

Syntax Example See also

### Description

ALTER EXCEPTION changes the text of an exception error message.

An exception can be altered by its creator and the SYSDBA user.

**See also**
ALTER PROCEDURE
ALTER TRIGGER
CREATE EXCEPTION
CREATE PROCEDURE
CREATE TRIGGER
DROP EXCEPTION

## ALTER EXCEPTION syntax

Example Definition

```
ALTER EXCEPTION name "<message>";
```

| Argument | Description |
|---|---|
| *name* | Name of an existing exception message. |
| "*<message>*" | Quoted string containing ASCII values. |

## ALTER EXCEPTION example

Syntax Definition

This statement alters the message of an exception:

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

## ALTER INDEX

Syntax Examples See also

**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, then reactivate it when done.

To rebuild and restore its balance, deactivate the index then reactivate it. This method recreates a balanced 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 and the SYSDBA user.

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

**See also**

ALTER TABLE

CREATE INDEX

DROP INDEX

SET STATISTICS

## ALTER INDEX syntax

Examples Definition

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

| 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. |

## ALTER INDEX examples

Syntax Definition

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

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

## ALTER PROCEDURE

Syntax Example See also

### 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.

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

A procedure can be altered by its creator and the SYSDBA user.

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.

**See also**

CREATE PROCEDURE

DROP PROCEDURE

EXECUTE PROCEDURE

SET TERM

## ALTER PROCEDURE syntax

Example Definition

```
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 data types are listed under CREATE PROCEDURE. |
| RETURNS *param* <*datatype*> | Output parameters used by the procedure. Legal data types are listed under CREATE PROCEDURE. |
| <*procedure_body*> | Procedure body. Includes:<br>Local variable declarations.<br>A block of statements in procedure and trigger language.<br>See CREATE PROCEDURE for a complete description. |
| *terminator* | Terminator defined by the ISQL SET TERM command to signify the end of the procedure body. |

## ALTER PROCEDURE example

Syntax Definition

This statement alters the GET_EMP_PROJ procedure, changing the return parameter to have a data type 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 ; !!
```

## ALTER TABLE

Syntax Examples See also

### Description

ALTER TABLE enables the structure of an existing table to be modified. A single ALTER TABLE can perform multiple adds and drops.

Naming column constraints is optional. If a name is not specified, 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.

A table can be altered by its creator and the SYSDBA user.

ALTER TABLE fails if current data in a table violates a PRIMARY KEY or UNIQUE constraint definition added to the table. It also fails if a column to be dropped is:

- Part of a UNIQUE, PRIMARY, or FOREIGN KEY constraint, or is used in a CHECK constraint.
- Used in the value expression of a computed column.

Drop the constraint or computed column before dropping the table column. PRIMARY KEY and UNIQUE constraints cannot be dropped if referenced by FOREIGN KEY constraints. In these cases, drop the FOREIGN KEY constraint before dropping the PRIMARY KEY or UNIQUE key it references.

When altering a column based on a domain, an additional CHECK constraint can be supplied for the column. Changes to tables that contain CHECK constraints with subqueries may cause constraint violations.

**Caution:** When a column is altered or dropped any data stored in it is lost.

**See also**

ALTER DOMAIN

CREATE DOMAIN

CREATE TABLE

## ALTER TABLE syntax

Examples Definition

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

<operation> = {ADD <col_def> | ADD <table_constraint> | DROP col
| DROP CONSTRAINT constraint}

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

**Note:** The COLLATE clause cannot be specified for BLOB columns.

```
<col_constraint> = [CONSTRAINT constraint] <constraint_def>
[<col_constraint>]

<constraint_def> = {PRIMARY KEY | UNIQUE
| CHECK (<search_condition>)
  | REFERENCES other_table [(other_col [, other_col ...])]}

<datatype> = {
{SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION} [<array_dim>]
  | {DECIMAL | NUMERIC} [(precision [, scale])] [<array_dim>]
  | DATE [<array_dim>]
  | {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
    [(1...32767)] [<array_dim>] [CHARACTER SET charname]
  | {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
    [VARYING] [(1...32767)] [<array_dim>]
  | BLOB [SUB_TYPE {int | subtype_name}] [SEGMENT SIZE int]
    [CHARACTER SET charname]
  | BLOB [(seglen [, subtype])]
  }

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

**Note:** Outermost brackets (boldface) must be included when declaring arrays.

```
<table_constraint> = CONSTRAINT constraint <tconstraint_opt>
[<table_constraint>]
<tconstraint_opt> = {
{PRIMARY KEY | UNIQUE} (col [, col ...])
| FOREIGN KEY (col [, col ...]) REFERENCES other_table
| CHECK (<search_condition>)
}
```

**Note:** For the full syntax of <*search_condition*>, see CREATE TABLE.

| 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 data type. Can include default values, column constraints, and a specific collation order. |
| *<table_constraint>* | Description of a new table constraint to add. Only one table constraint can be added to a table. |
| *col* | Name of column to add or drop. Column name must be unique within the table. |
| *<constraint>* | Name of constraint to add or drop. Constraint name must be unique within the table. |
| COLLATE *collation* | Adds a collation order to the specified table. |
| *<datatype>* | Data type of column to add. |
| *domain* | Name of a domain upon which a column definition should be based. |
| COMPUTED [BY] *<expr>* | Specifies a column calculated from *<expr>* and which is not therefore allocated storage space in the database. *<expr>* can be any arithmetic expression valid for the data types in the expression. Other columns referenced in *<expr>* must exist before they can be used. BLOB columns cannot be referenced. *<expr>* must return a single value, and cannot return an array. |
| 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 only a column attribute. |
| DEFAULT | Specifies a default column value that is entered when no other entry is made. Values:<br>*literal*–Inserts a specified string, numeric value, or date value.<br>NULL–Enters a NULL value.<br>USER–Enters the user name of the current user. Column must be of compatible text type to use the default.<br>Defaults set at column level override defaults set at the domain level. |
| *<constraint_def>* | Column constraint definition. |
| CONSTRAINT | Adds a named constraint to a column. |
| DROP CONSTRAINT | Drops the specified table constraint. |

## ALTER TABLE examples

Syntax Definition

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

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

**Note:** This statement results in the loss of any data in the dropped column.

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

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

## ALTER TRIGGER

Syntax Examples See also

**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 and the SYSDBA user.

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

**See also**

CREATE TRIGGER

DROP TRIGGER

SET TERM

## ALTER TRIGGER syntax

Examples Definition

```
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 | Specifies that a trigger action takes effect when fired (default). |
| 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 32767, inclusive. Lower-number triggers fire first. Triggers for a table need not be consecutive. Triggers on the same action with the same position number will 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. |

## ALTER TRIGGER examples

Syntax Definition

The following statement modifies the trigger SET_CUST_NO to be inactive:

```
ALTER TRIGGER SET_CUST_NO INACTIVE;
```

The next 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 ; !!
```

## AVG( )

Syntax Examples See also

### Description

AVG() is an aggregate function that returns the average of the values in a specified column or an expression. Only numeric data types 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.

**See also**

COUNT()

MAX()

MIN()

SUM()

## AVG( ) syntax

Examples Definition

```
AVG ([ALL] <val> | DISTINCT <val>);
```

| Argument | Description |
|---|---|
| ALL | Returns the average of all values. |
| DISTINCT | Eliminates duplicate values before calculating the average. |
| *<val>* | A column or expression that evaluates to a numeric data type. |

## AVG( ) example

Syntax Definition

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

```
SELECT SUM (BUDGET), AVG (BUDGET), MIN (BUDGET), MAX (BUDGET)
  FROM DEPARTMENT
  WHERE HEAD_DEPT = 100;
```

## CAST( )

Syntax Example See also

### Description

CAST() allows mixing of numerics and characters in a single expression by converting *<val>* to a specified data type.

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

Data types can be converted as shown in the following table:

| From Data Type Class | To Data Type Class |
|---|---|
| NUMERIC | CHARACTER, VARYING CHARACTER, DATE |
| CHARACTER, VARYING CHARACTER | NUMERIC, DATE |
| DATE | CHARACTER, VARYING CHARACTER, DATE |

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

**See also**
UPPER()

## CAST( ) syntax

Example Definition

```
CAST (<val> AS <datatype>);
```

| Argument | Description |
|---|---|
| *<val>* | A column, constant, or expression that evaluates to a character data type. |
| *<datatype>* | Data type to which to convert. |

## CAST( ) example

Syntax Definition

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

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

## COMMIT

Syntax Examples See also

**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 (that make no changes), use COMMIT rather than ROLLBACK. The effect is the same, but the time and space required by subsequent transactions is reduced if COMMIT is used.

**Important:** The RELEASE argument is only available for compatibility with previous versions of InterBase.

**See also**

ROLLBACK

## COMMIT syntax

Examples Definition

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

| Argument | Description |
|---|---|
| WORK | An optional word used for compatibility with other relational databases that require it. |
| TRANSACTION *name* | Commit 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. |

## COMMIT examples

Syntax Definition

The following statement makes permanent the changes to the database:

```
COMMIT;
```

The next statement commits a named transaction:

```
COMMIT TR1;
```

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

```
COMMIT RETAIN;
```

## CONNECT

Syntax Example See also

### 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*). Databases used by Windows 3.1 clients are always on remote servers. An error message occurs if InterBase cannot locate the database.
- Optionally specifies a user name and password for use when attaching to the database. Windows 3.1 clients must always send a valid user name and password.
- 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.

When CONNECT attaches to a database, it uses the default character set (NONE), or one specified in a previous SET NAMES statement. Each time CONNECT is used to attach to a database, previous attachments are disconnected.

**See also**

SET NAMES

## CONNECT syntax

Example Definition

```
CONNECT ["]<filespec>["] [USER "username" [PASSWORD "password"]];
```

| Argument | Description |
|---|---|
| "*<filespec>*" | Database file name. May include path specification and node. |
| USER "*username*" | String that specifies a user name for use when attaching to the database. The server checks the user name against the *isc.gdb* security database. User names are case insensitive on the server. |
| PASSWORD "*password*" | String that specifies password for use when attaching to the database. The server checks the user name and password against the *isc.gdb* security database. Case sensitivity is retained for the comparison. |

## CONNECT example

Syntax Definition

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

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

## COUNT( )

Syntax Example See also

### Description

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

**See also**

AVG()

MAX()

MIN()

SUM()

## COUNT( ) syntax

Example Definition

```
COUNT ( * | [ALL] <val> | DISTINCT <val>);
```

| 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 that evaluates to a numeric data type. |

## COUNT( ) example

Syntax Definition

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

```
SELECT COUNT (DISTINCT CURRENCY) FROM COUNTRY;
```

## CREATE DATABASE

Syntax Examples See also

**Description**

CREATE DATABASE creates a 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 may 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 character set used by the database.
- System tables that describe the structure of the database.

After creating the database, a user may define its tables, views, indexes, and system views.

On NetWare only, CREATE DATABASE also enables you to define the Write-ahead Log (WAL) Protocol.

**See also**

ALTER DATABASE

DROP DATABASE

InterBase character sets and collation orders

## CREATE DATABASE syntax

Examples Definition

```
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>]
```

| Argument | Description |
|---|---|
| "*<filespec>*" | A new database file specification. File naming conventions are platform-specific. |
| USER "*username*" | User name is checked 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. |
| PASSWORD "*password*" | Password is checked 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 and password on attachment to a server. |
| 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 quoted 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 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. |

## CREATE DATABASE examples

Syntax Definition

The following statement creates a database in the current directory:

```
CREATE DATABASE "employee.gdb";
```

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

```
CREATE DATABASE "employee.gdb" PAGE_SIZE 2048;
```

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

```
CREATE DATABASE "employee.gdb"
  DEFAULT CHARACTER SET "ISO8859_1"
  FILE "employee.gd1" STARTING AT PAGE 10001 LENGTH 10000 PAGES;
```

## CREATE DOMAIN

Syntax Examples See also

### 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:

- Data type
- 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 when creating domains. It is possible to create a domain with contradictory constraints, such as declaring a domain NOT NULL, and assigning it a DEFAULT value of NULL.

The data type specification for a CHAR, VARCHAR, or BLOB 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.

The COLLATE clause enables specification of a particular collation order for CHAR, VARCHAR, and BLOB text data types. 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 data type definition. See the *Language Reference* for a complete list of collation orders.

Columns based on a domain definition inherit all characteristics of the domain. The domain default, collation clause, and NOT NULL setting may 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.

**See also**

ALTER DOMAIN

ALTER TABLE

CREATE TABLE

DROP DOMAIN

InterBase character sets and collation orders

## CREATE DOMAIN syntax

Examples    Definition

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

**Note:** The COLLATE clause cannot be specified for BLOB columns.

```
<datatype> = {
{SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION} [<array_dim>]
  | {DECIMAL | NUMERIC} [(precision [, scale])] [<array_dim>]
  | DATE [<array_dim>]
  | {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
     [(1...32767)] [<array_dim>] [CHARACTER SET charname]
  | {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
     [VARYING] [(1...32767)] [<array_dim>]
  | BLOB [SUB_TYPE {int | subtype_name}] [SEGMENT SIZE n]
     [CHARACTER SET charname]
  | BLOB [(seglen [, subtype])]
  }

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

**Note:** Outermost brackets (boldface) must be included when declaring arrays.

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

| Argument | Description |
|---|---|
| *domain* | Unique name for the domain. |
| *<datatype>* | SQL data type. |
| DEFAULT | Specifies a default column value that is entered when no other entry is made. Values:<br>*literal*: Inserts a specified string, numeric value, or date value.<br>NULL: Enters a NULL value.<br>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_cond*>) | 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. |

## CREATE DOMAIN examples

Syntax Definition

The following statement creates a domain that must have a positive value greater than 1000, with a default value of 9999. 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 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 statement creates a domain that defines an array of CHARACTER data type:

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

In the following 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 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;
```

## CREATE EXCEPTION

Syntax Examples See also

### 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.
- Displays an error message.

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

**See also**

ALTER EXCEPTION

ALTER PROCEDURE

ALTER TRIGGER

CREATE PROCEDURE

CREATE TRIGGER

DROP EXCEPTION

## CREATE EXCEPTION syntax

Examples Definition

```
CREATE EXCEPTION name "<message>";
```

| 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. |

## CREATE EXCEPTION examples

Syntax Definition

This 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 set exception when SQLCODE -530 is set, which is a violation of a FOREIGN KEY constraint.

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

## CREATE GENERATOR

Syntax Example See also

### 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.

After a generator is created, SET GENERATOR can set or change its current value. The generator can be used by writing a trigger, procedure, or SQL statement that calls GEN_ID().

**See also**

GEN_ID()

SET GENERATOR

SET TERM

## CREATE GENERATOR syntax

Example Definition

```
CREATE GENERATOR name;
```

| Argument | Description |
|---|---|
| *name* | Name for the generator. |

## CREATE GENERATOR example

Syntax Definition

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

## CREATE INDEX

Syntax Examples See also

**Description**

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:** BLOB columns and arrays cannot be indexed.

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.

**See also**

ALTER INDEX

DROP INDEX

SELECT

SET STATISTICS

## CREATE INDEX syntax

Examples Definition

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

| 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. |

## CREATE INDEX examples

Syntax Definition

The following statement creates a unique index:

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

The next statement creates a descending index:

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

The following statement creates a two-column index:

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

## CREATE PROCEDURE

Syntax Examples See also

### 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 data types that a procedure receives from the calling program.
- RETURNS followed by a list of output parameters and their data types if the procedure returns values to the calling program.

The procedure body contains:

- An optional list of local variables and their data types.
- 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 (e.g., DROP TABLE, 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 UDFs linked with the database and generators.
- Powerful 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.

| 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 which 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, which can be handled with WHEN. |
| EXECUTE PROCEDURE *proc_name* [*var* [, *var* ...]] [RETURNING_VALUES *var* [, *var* ...]] | Executes stored procedure *proc_name* with the input arguments listed following the procedure name, returning values in the output arguments listed following RETURNING_VALUES. Input and output parameters must be variables defined within the procedure. Enables nested procedures and recursion. |
| 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*>.<br><br><*select_statement*>: a normal SELECT statement, except 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.<br><br><*condition*>: a Boolean expression (TRUE, FALSE, or UNKNOWN), generally two expressions as operands of a comparison operator. |
| POST_EVENT *event_name* | Posts the event *event_name.* |
| SUSPEND | In a SELECT procedure SUSPEND 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*>. First <*condition*> is tested, and if it is TRUE, then <*compound_statement*> is performed. This sequence is repeated 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.<br><br><*error*>: EXCEPTION *exception_name*, SQLCODE errcode or GDSCODE number.<br><br>ANY: handles any types of errors. |

**See also**

ALTER EXCEPTION

ALTER PROCEDURE

CREATE EXCEPTION

DROP EXCEPTION

DROP PROCEDURE

EXECUTE PROCEDURE

SELECT

SET TERM

## CREATE PROCEDURE syntax

Examples Definition

```
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
  | {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:<br>*param*: Name of the input parameter, unique for variables in the procedure.<br>*<datatype>* Any InterBase data types. |
| RETURNS *param <datatype>* | Output parameters that the procedure uses to return values to the calling program.<br>*param*: Name of the output parameter, unique for variables within the procedure.<br>*<datatype>*: Any InterBase data types.<br>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. Each declaration 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. *<datatype>* Any InterBase data types. |

| | |
|---|---|
| *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 SET TERM which signifies the end of the procedure body. Used in ISQL only. |

## CREATE PROCEDURE examples

Syntax Definition

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 statement:

```
SELECT * FROM ORG_CHART
```

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

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.

## CREATE SHADOW

Syntax Examples See also

**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 activates the disk shadow so that it can take the place of the database. If CONDITIONAL is specified, then when the database administrator 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.

**See also**

DROP SHADOW

## CREATE SHADOW syntax

Examples Definition

```
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>]
```

| 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 of shadow unavailability. All attachments and accesses succeed; references to the shadow are deleted and the shadow file is detached. |
| MANUAL | Specifies that database attachments and accesses fail until a shadow becomes available, or 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. Shadow file specifications cannot include a node name. |
| LENGTH [=] *int* [PAGE[S]] | Length in database pages of an additional shadow file. Page size is determined by 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. |

## CREATE SHADOW examples

Syntax Definition

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

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

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

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

The following 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";
```

## CREATE TABLE

Syntax Examples See also

**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 data type 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 data type 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 data type 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.

The COLLATE clause enables specification of a particular collation order for CHAR, VARCHAR, and BLOB text data types. 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 data type definition. See the *Language Reference* for a complete list of collation orders.

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.

Integrity constraints can be defined for a table when it is created. Integrity constraints are rules that control the database and its contents, enforcing column-to-table and table-to-table relationships, and validating data entries. They span all transactions that access the database and are automatically maintained by the system. CREATE TABLE can create the following types of integrity constraints:

- PRIMARY KEY constraints are unique identifiers for each row in a table. Values in this column or ordered set of columns cannot occur in more than one row. A PRIMARY KEY column must also define the NOT NULL attribute. A table can have only one PRIMARY KEY that can be defined on one or more columns.
- 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 ensure that values in a set of columns that define a FOREIGN KEY are the same as values in UNIQUE or PRIMARY KEY columns in a referenced table. Before the REFERENCES constraint can be added, the UNIQUE or PRIMARY KEY columns it references must already be defined in the referenced table.
- CHECK constraints enforce a <*search_condition*> that must be true for inserts or updates to the specified table. <*search_condition*> can require a certain combination or range of values or compare the value entered with data in other columns.

For unnamed constraints, the system assigns a unique constraint name stored in the system table RDB$RELATION_CONSTRAINTS.

**Note:** Constraints are not enforced on expressions.

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

- Define an InterBase table composed of columns and 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.

**See also**

CREATE DOMAIN

GRANT

REVOKE

InterBase character sets and collation orders

## CREATE TABLE syntax

Examples            Definition

```
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]
```

**Note:** The COLLATE clause cannot be specified for BLOB columns.

```
<datatype> = {
{SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION} [<array_dim>]
  | {DECIMAL | NUMERIC} [(precision [, scale])] [<array_dim>]
  | DATE [<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 ...]]
```

**Note:** Outermost brackets (boldface) must be included when declaring arrays.

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

<col_constraint> = [CONSTRAINT constraint] <constraint_def>
[<col_constraint>]

<constraint_def> = {UNIQUE | PRIMARY KEY
| CHECK (<search_condition>)
  | REFERENCES other_table [(other_col [, other_col ...])]}

<tconstraint> = CONSTRAINT constraint <tconstraint_def>
[<tconstraint>]

<tconstraint_def> = {{PRIMARY KEY | UNIQUE} (col [, col ...])
| FOREIGN KEY (col [, col ...]) REFERENCES other_table
  | 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> {[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>] | <constant> | <expr> | <function>
   | 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 that returns exactly one value.

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

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

| Argument | Description |
|---|---|
| *table* | Name for the table. Table name must be unique among table and procedure names in the database. |
| EXTERNAL [FILE] "<*filespec*>" | Declares that data for the table to create 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 data type for a column. |
| COMPUTED [BY] (<*expr*>) | Bases a column definition on an expression. The expression must return a single value, and cannot be of array type or return an array. <*expr*> is any arithmetic expression that is valid for the data types in the columns. |
| *domain* | Name of an existing domain. |
| COLLATE *collation* | Specifies the collation order for the column. Collation order set at column level overrides domain level collation order. |
| DEFAULT | Specifies a default column value that is entered when no other entry is made. Values:<br>*literal*: Inserts a specified string, numeric value, or date value.<br>NULL: Enters a NULL value.<br>USER: Enters the user name of the current user. Column must be of compatible text type to use the default.<br>Defaults set at column level override defaults set at the |

| | |
|---|---|
| | domain level. |
| CONSTRAINT *constraint* | Places a named constraint on a table or column. A constraint is a rule applied to a table's structure or contents. If this clause is omitted, InterBase generates a system name for the constraint. |

## CREATE TABLE examples

Syntax Definition

The following 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 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 statement 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));
```

The next 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 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);
```

## CREATE TRIGGER

Syntax Examples See also

**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 data types.
- 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_name*. 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 UDFs linked with the calling application and generators.
- 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.

| 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 which assigns the value of expression to 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, which 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. Input and output arguments must be local variables. |
| FOR <*select_statement*> DO <*compound_statement*> | Repeats the statement or block following DO for every qualifying row retrieved by <*select_statement*>.<br><*select_statement*>: a normal SELECT statement, except 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.<br><*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* | Posts the event *event_name.* |
| WHILE (<*condition*>) DO <*compound_statement*> | While <*condition*> is TRUE, keep performing <*compound_statement*>. First <*condition*> is tested, and if it is TRUE, then <*compound_statement*> is performed. This sequence is repeated 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.<br><*error*>: EXCEPTION *exception_name*, SQLCODE errcode or GDSCODE number.<br>ANY: handles any types of errors. |

**See also**

ALTER EXCEPTION

ALTER TRIGGER

CREATE EXCEPTION

CREATE PROCEDURE

DROP EXCEPTION

DROP TRIGGER

EXECUTE PROCEDURE

SET TERM

## CREATE TRIGGER syntax

Examples Definition

```
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

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

<datatype> = {
{SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION}
  | {DECIMAL | NUMERIC} [(precision [, scale])]
  | DATE
  | {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
     [(1...32767)] [CHARACTER SET charname]
  | {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
     [VARYING] [(1...32767)]}
```

| Argument | Description |
|---|---|
| name | Name of the trigger. The name 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 | (Default.) Optionally specifies that trigger action takes effect when fired. |
| INACTIVE | Optionally specifies that trigger action does not take effect. |
| BEFORE | Specifies that the trigger fires before the associated operation. |
| AFTER | Specifies that the trigger fires after the associated operation. |
| 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 32767, inclusive. Lower-number triggers fire first. Default: 0 = first trigger to fire.<br>Triggers for a table need not be consecutive. Triggers on the same action with the same position number will fire in random order. |
| DECLARE | Declares local variables used only in the trigger. Each declaration |

| | |
|---|---|
| VARIABLE var <datatype> | must be preceded by DECLARE VARIABLE and followed by a semicolon (;). var: Local variable name, unique in the trigger. <datatype>: The data type 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 which signifies the end of the trigger body. Used in ISQL only. |

## CREATE TRIGGER examples

Syntax Definition

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.

## CREATE VIEW

Syntax Examples See also

### 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, triggers, 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*>, 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.

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.

**See also**

CREATE TABLE

DROP VIEW

GRANT

INSERT

REVOKE

SELECT

UPDATE

## CREATE VIEW syntax

Examples Definition

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

| 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*>. |

## CREATE VIEW examples

Syntax Definition

The following 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 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 insert:

```
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 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;
```

## DECLARE EXTERNAL FUNCTION

Syntax Example See also

**Description**

DECLARE EXTERNAL FUNCTION provides information about a UDF to the database: where to find it, its name, the input parameters it requires, and the single value it returns. A UDF exists in a library outside the database.

The function being called must be compiled using the CDECL calling convention.

"<*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.

**Important:** Do not use DECLARE EXTERNAL FUNCTION when creating a database on a Netware server. UDF libraries cannot be created or used on NetWare servers.

**See also**
DROP EXTERNAL FUNCTION

## DECLARE EXTERNAL FUNCTION syntax

Example Definition

```
DECLARE EXTERNAL FUNCTION name [<datatype> | CSTRING (int)
  [, <datatype> | CSTRING (int) ...]]
  RETURNS {<datatype> [BY VALUE] | CSTRING (int)}
  ENTRY_POINT "<entryname>"
  MODULE_NAME "<modulename>";
```

| Argument | Description |
|---|---|
| *name* | Name of the UDF. |
| *<datatype>* | Data type of an input or return parameter. All input parameters are passed to a UDF by reference. Return parameters can be passed by value. |
| RETURNS | Specifies the return value of a function. |
| BY VALUE | Specifies that a return value should be passed by value, rather than by reference. |
| CSTRING (*int)* | Specifies a UDF that returns a null-terminated string *int* bytes in length. |
| "*<entryname>*" | Quoted string specifying the name of the UDF as stored in the UDF library. |
| "*<modulename>*" | Quoted file specification identifying the object module in which the UDF resides. |

## DECLARE EXTERNAL FUNCTION example

Syntax Definition

The following statement declares the function TOPS() to a database:

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

## DECLARE FILTER

Syntax Example See also

### 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.

**See also**
DROP FILTER

## DECLARE FILTER syntax

Example Definition

```
DECLARE FILTER filter
INPUT_TYPE subtype OUTPUT_TYPE subtype
  ENTRY_POINT "<entryname>" MODULE_NAME "<modulename>";
```

| 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. |

## DECLARE FILTER example

Syntax Definition

The following statement declares a BLOB filter:

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

## DELETE

Syntax Examples See also

**Description**

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.

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

**Caution:** Without a WHERE clause, a searched delete removes all rows from a table.

**See also**

GRANT

REVOKE

SELECT

## DELETE syntax

Examples Definition

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

| Argument | Description |
|---|---|
| *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. |

## DELETE examples

Syntax Definition

The following statement deletes all rows in a table:

```
DELETE FROM EMPLOYEE_PROJECT;
```

The next statement deletes the row for employee #141:

```
DELETE FROM SALARY_HISTORY
WHERE EMP_NO = 141;
```

**Note:** EMP_NO is a PRIMARY KEY of the EMPLOYEE table and therefore is guaranteed to uniquely identify a row.

## DROP DATABASE

Syntax Example See also

**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 and the SYSDBA user.

**See also**

ALTER DATABASE

CREATE DATABASE

## DROP DATABASE syntax

Example Definition

```
DROP DATABASE;
```

## DROP DATABASE example

Syntax Definition

The following statement deletes the current database:

```
DROP DATABASE;
```

## DROP DOMAIN

Syntax Example See also

**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 can be dropped by its creator and the SYSDBA user.

**See also**
ALTER DOMAIN
ALTER TABLE
CREATE DOMAIN

## DROP DOMAIN syntax

Example Definition

```
DROP DOMAIN name;
```

| Argument | Description |
|---|---|
| *name* | Name of an existing domain. |

## DROP DOMAIN example

Syntax Definition

The following statement deletes a domain:

```
DROP DOMAIN COUNTRYNAME;
```

## DROP EXCEPTION

Syntax Example See also

### Description

DROP EXCEPTION removes an exception from a database.

Exceptions used in existing procedures and triggers cannot be dropped.

**Tip:** Use SHOW EXCEPTION to display a list of exceptions' dependencies, the procedures and triggers that use the exceptions.

An exception can be dropped by its creator and the SYSDBA user.

**See also**

ALTER EXCEPTION

ALTER PROCEDURE

ALTER TRIGGER

CREATE EXCEPTION

CREATE PROCEDURE

CREATE TRIGGER

## DROP EXCEPTION syntax

Example    Definition

```
DROP EXCEPTION name;
```

| Argument | Description |
|---|---|
| *name* | Name of an existing exception message. |

## DROP EXCEPTION example

Syntax    Definition

This statement drops an exception:

```
DROP EXCEPTION UNKNOWN_EMP_ID;
```

## DROP EXTERNAL FUNCTION

Syntax Example See also

### Description

DROP EXTERNAL FUNCTION deletes a UDF declaration from a database. Dropping a UDF declaration from a database does not remove it from the corresonding 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 creator and the SYSDBA user.

**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.

**See also**

DECLARE EXTERNAL FUNCTION

## DROP EXTERNAL FUNCTION syntax

Example Definition

```
DROP EXTERNAL FUNCTION name;
```

| Argument | Description |
|---|---|
| *name* | Name of an existing UDF. |

## DROP EXTERNAL FUNCTION example

Syntax Definition

This statement drops a UDF:

```
DROP EXTERNAL FUNCTION TOPS;
```

## DROP FILTER

Syntax Example See also

**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 corresonding 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 and the SYSDBA user.

**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.

**See also**

DECLARE FILTER

## DROP FILTER syntax

Example Definition

```
DROP FILTER name;
```

| Argument | Description |
|---|---|
| *name* | Name of an existing BLOB filter. |

## DROP FILTER example

Syntax Definition

This statement drops a BLOB filter:

```
DROP FILTER DESC_FILTER;
```

## DROP INDEX

Syntax Example See also

**Description**

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

An index can be dropped by its creator and the SYSDBA user.

**Important:** System-defined indexes, such as those for UNIQUE, PRIMARY KEY, and FOREIGN KEY integrity constraints, cannot be dropped.

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

**See also**

ALTER INDEX

CREATE INDEX

## DROP INDEX syntax

Example Definition

```
DROP INDEX name;
```

| Argument | Description |
|---|---|
| *name* | Name of an existing index. |

## DROP INDEX example

Syntax Definition

The following statement deletes an index:

```
DROP INDEX MINSALX;
```

## DROP PROCEDURE

Syntax Example See also

**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:** Use SHOW PROCEDURE to display a list of procedures' dependencies, the procedures, triggers, exceptions, and tables that use the procedures.

A procedure can be dropped by its creator and the SYSDBA user.

**See also**

ALTER PROCEDURE

CREATE PROCEDURE

EXECUTE PROCEDURE

## DROP PROCEDURE syntax

Example Definition

```
DROP PROCEDURE name;
```

| Argument | Description |
|---|---|
| *name* | Name of an existing stored procedure. |

## DROP PROCEDURE example

Syntax Definition

The following statement deletes a procedure:

```
DROP PROCEDURE GET_EMP_PROJ;
```

## DROP SHADOW

Syntax Example See also

**Description**

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

A shadow can be dropped by its creator and the SYSDBA user.

**See also**
CREATE SHADOW

## DROP SHADOW syntax

Example Definition

```
DROP SHADOW set_num;
```

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

## DROP SHADOW example

Syntax Definition

The following statement deletes a shadow set from its database:

```
DROP SHADOW 1;
```

## DROP TABLE

Syntax Example See also

### 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 and the SYSDBA user.

**See also**

ALTER TABLE

CREATE TABLE

## DROP TABLE syntax

Example Definition

```
DROP TABLE name;
```

| Argument | Description |
|---|---|
| *name* | Name of an existing table. |

## DROP TABLE example

Syntax Definition

The following statement drops a table:

```
DROP TABLE COUNTRY;
```

## DROP TRIGGER

Syntax Example See also

**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 and the SYSDBA user.

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

**See also**

ALTER TRIGGER

CREATE TRIGGER

## DROP TRIGGER syntax

Example        Definition

```
DROP TRIGGER name;
```

| Argument | Description |
|---|---|
| *name* | Name of an existing trigger. |

## DROP TRIGGER example

Syntax        Definition

The following statement drops a trigger:

```
DROP TRIGGER POST_NEW_ORDER;
```

## DROP VIEW

Syntax Example See also

### 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 and the SYSDBA user.

**See also**
CREATE VIEW

## DROP VIEW syntax

Example Definition

```
DROP VIEW name;
```

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

## DROP VIEW example

Syntax Definition

The following statement removes a view definition:

```
DROP VIEW PHONE_LIST;
```

## EXECUTE PROCEDURE

Syntax Example See also

**Description**

EXECUTE PROCEDURE calls the specified stored procedure. If the procedure requires input parameters, they are passed as constants.

ISQL automatically displays return values.

**See also**

ALTER PROCEDURE

CREATE PROCEDURE

DROP PROCEDURE

## EXECUTE PROCEDURE syntax

Example Definition

```
EXECUTE PROCEDURE name [param [, param ...]];
```

| Argument | Description |
|---|---|
| *name* | Name of an existing stored procedure in the database. |
| *param* | Input parameter. Must be a constant. |

## EXECUTE PROCEDURE example

Syntax Definition

The following statement demonstrates how the executable procedure, DEPT_BUDGET, is called:

```
EXECUTE PROCEDURE DEPT_BUDGET 100;
```

## GEN_ID( )

Syntax Example See also

### Description

The GEN_ID() function:

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

GEN_ID() is useful for automatically producing unique values to insert in 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 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.

**See also**

CREATE GENERATOR

SET GENERATOR

## GEN_ID( ) syntax

Example Definition

```
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. |

## GEN_ID( ) example

Syntax Definition

The following 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 will be set to 1. The next time, it will be set to 2, and so on.

## GRANT

Syntax Examples See also

### Description

GRANT assigns privileges for database objects to users 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.

To access a table or view, a user or object needs SELECT, INSERT, UPDATE, or DELETE privileges for that table or view. SELECT, INSERT, UPDATE, and DELETE privileges may be assigned as a unit with ALL.

To call a stored procedure in an application, a user or object needs EXECUTE privilege for it.

Users can be given permission to grant privileges to other users by providing a *<userlist>* that includes the WITH GRANT OPTION. Users can only grant to others the privileges that they, themselves, are assigned.

Privileges may be assigned to all users by specifying PUBLIC in place of a list of user names. Specifying PUBLIC grants privileges only to users, not to database objects.

The following table summarizes available privileges:

| Privilege | Lets User . . . |
|---|---|
| ALL | Perform SELECT, DELETE, INSERT, UPDATE, and EXECUTE. |
| SELECT | Retrieve rows from a table or view. |
| DELETE | Eliminate rows from a table or view. |
| INSERT | Store new rows in a table or view. |
| UPDATE | Change the current value of one or more columns in a table or view. Can be restricted to a specified subset of columns. |
| EXECUTE | Execute a stored procedure. |

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

**See also**

REVOKE

## GRANT syntax

Examples          Definition

```
GRANT{
{ALL [PRIVILEGES] | SELECT | DELETE | INSERT
  | UPDATE [(col [, col ...])]}
    ON [TABLE] {tablename | viewname}
    TO {<object> | <userlist>}
  | EXECUTE ON PROCEDURE procname
    TO {<object> | <userlist>}
  };

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

<userlist> = [USER] username [, [USER] username ...]
[WITH GRANT OPTION]
```

| Argument | Description |
|---|---|
| *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. |
| *<object>* | Name of a user or an existing database object to which privileges are to be granted. |
| *<userlist>* | A list of users to whom privileges are to be granted. |
| WITH GRANT OPTION | Passes GRANT authority for privileges listed in the GRANT statement to *<userlist>*. |

## GRANT examples

Syntax          Definition

The following 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;
```

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

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

## INSERT

Syntax Examples See also

**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.

**Caution:** 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.

**See also**

GRANT

REVOKE

SET TRANSACTION

UPDATE

## INSERT syntax

Examples Definition

```
INSERT INTO <object> [(col [, col ...])]
{VALUES (<val> [, <val> ...]) | <select_expr>};

<object> = tablename | viewname

<val> = {
<constant> | <expr>| <function> | NULL | USER
  } [COLLATE collation]
```

**Note:** The COLLATE clause cannot be used with BLOB values.

```
<constant> = num | "string" | charsetname "string"

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

<function> = {
CAST (<val> AS <datatype>)
  | UPPER (<val>)
  | GEN_ID (generator, <val>)
  }

<select_expr> = A SELECT returning zero or more rows and where the number of
columns in each row is the same as the number of items to be inserted.
```

| Argument | Description |
|---|---|
| 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. |

## INSERT examples

Syntax Definition

The following statement adds a row to a table, assigning values to two columns:

```
INSERT INTO EMPLOYEE_PROJECT (EMP_NO, PROJ_ID) VALUES (52, "DGPII");
```

The next 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";
```

## MAX( )

Syntax Example See also

**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.

**See also**

AVG()

COUNT()

MIN()

SUM()

InterBase character sets and collation orders

## MAX( ) syntax

Example Definition

```
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 or expression that evaluates to a numeric data type. |

## MAX( ) example

Syntax Definition

The following statement demonstrates the use of SUM(), AVG(), MIN(), and MAX():

```
SELECT SUM (BUDGET), AVG (BUDGET), MIN (BUDGET), MAX (BUDGET)
FROM DEPARTMENT
WHERE HEAD_DEPT = 100;
```

## MIN( )

Syntax Example See also

### 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.

**See also**

AVG()

COUNT()

MAX()

SUM()

InterBase character sets and collation orders

## MIN( ) syntax

Example Definition

```
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 or expression that evaluates to a numeric data type. |

## MIN( ) example

Syntax Definition

The following statement demonstrates the use of SUM(), AVG(), MIN(), and MAX():

```
SELECT SUM (BUDGET), AVG (BUDGET), MIN (BUDGET), MAX (BUDGET)
FROM DEPARTMENT
WHERE HEAD_DEPT = 100;
```

## REVOKE

Syntax Examples See also

**Description**

REVOKE removes privileges to access database objects from users or other database objects. Privileges are operations for which a user has authority. The following table defines SQL privileges:

| **Privilege** | **Removes a User’s Privilege to . . .** |
|---|---|
| ALL | Perform SELECT, DELETE, INSERT, UPDATE, 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 of one or more columns in a table or view. Can be restricted to a specified subset of columns. |
| 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 may 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.

**See also**
GRANT

## REVOKE syntax

Examples Definition

```
REVOKE [GRANT OPTION FOR]{
{ALL [PRIVILEGES] | SELECT | DELETE | INSERT
    | UPDATE [(col [, col ...])]}
    ON [TABLE] {tablename | viewname}
    FROM {<object> | <userlist>}
  | EXECUTE ON PROCEDURE procname
    FROM {<object> | <userlist>}
  };

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

<userlist> = [USER] username [, [USER] username ...]
```

| Argument | Description |
|---|---|
| GRANT OPTION FOR | Removes grant authority for privileges listed in the REVOKE statement from *<userlist>*. Cannot be used with *<object>*. |
| *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. |
| *<object>* | Name of a user or an existing database object from which privileges are to be revoked. |
| *<userlist>* | A list of users from whom privileges are to be revoked. |

## REVOKE examples

Syntax Definition

The following statement takes the SELECT privilege away from a user for a table:

```
REVOKE SELECT ON COUNTRY FROM MIREILLE;
```

The following 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;
```

## ROLLBACK

Syntax Examples See also

**Description**

ROLLBACK undoes all DML statements since the last COMMIT. Since DDL statements are auto-committed, ROLLBACK has no effect on them.

**See also**
COMMIT

## ROLLBACK syntax

Examples Definition

```
ROLLBACK [WORK];
```

| Argument | Description |
|---|---|
| WORK | Optional word allowed for compatibility. |

## ROLLBACK examples

Syntax Definition

The following statements illustrate ROLLBACK in ISQL. The first ROLLBACK undoes the INSERT but not the CREATE TABLE. The second ROLLBACK has no effect, because the INSERT has been committed.

```
CREATE TABLE T (A INT);
INSERT INTO T VALUES (5);
ROLLBACK;
INSERT INTO T VALUES (7);
COMMIT;
ROLLBACK;
SELECT * FROM T;
          A
==============
          7
```

## SELECT

Syntax Examples See also

**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.*
- Directly retrieve multiple rows, or parts of rows, from a table.
- 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 (WHERE, GROUP BY, HAVING, UNION, PLAN, ORDER BY). The SELECT and FROM clauses are required for both singleton and multi-row SELECTs; all other clauses listed below are optional The following table explains the purpose of each clause:

| Clause | Purpose |
|---|---|
| SELECT | Lists columns to retrieve. |
| FROM | Identifies the tables to search for values. |
| WHERE | Specifies the search conditions used to restrict retrieved rows to a subset of all available rows. A WHERE clause may contain its own SELECT statement, referred to as a *subquery.* |
| GROUP BY | Groups related rows based on common column values. Used in conjunction with HAVING. |
| HAVING | Restricts rows generated by GROUP BY to a subset of those rows. |
| UNION | Combines the results of two or more SELECT statements to produce a single, dynamic table without duplicate rows. |
| ORDER BY | Specifies the sort order of rows returned by a SELECT, either ascending (ASC), the default, or descending (DESC). |
| PLAN | Specifies the query plan that should be used by the query optimizer instead of one it would normally choose. |

**See also**

DELETE

INSERT

UPDATE

## SELECT syntax

Examples Definition

```
SELECT [DISTINCT | ALL] {* | <val> [, <val> ...]}
  FROM <tableref> [, <tableref> ...]
  [WHERE <search_condition>]
  [GROUP BY col [COLLATE collation] [, col [COLLATE collation] ...]
  [HAVING <search_condition>]
  [UNION <select_expr>]
  [PLAN <plan_expr>]
  [ORDER BY <order_list>]

<val> = {
col [<array_dim>] | <constant> | <expr> | <function>
  | NULL | USER | RDB$DB_KEY
  }

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

**Note:** Outermost brackets (boldface) must be included when referencing arrays.

```
<constant> = num | "string" | charsetname "string"

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

<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] | {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> {[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> = {= | < | > | <= | >= | !< | !> | <> | !=}

<select_one> = SELECT on a single column that returns exactly one row.

<select_list> = SELECT on a single column that returns zero or more rows.

<select_expr> = SELECT on a list of values that returns zero or more rows.

<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** |
|---|---|
| SELECT [DISTINCT \| ALL] | Specifies data to retrieve. DISTINCT prevents duplicate values from being returned. ALL, the default, retrieves every value. |
| {* \| <*val*> [, <*val*> ...] | * retrieves all columns for the specified tables. <*val*> [, <*val*> ...] retrieves a specific list of columns and 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 or view. 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_cond*> | 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_cond*> | Used with GROUP BY. Specifies a condition that limits grouped rows returned. |
| UNION | Combines two or more tables that are fully or partially identical in structure. |
| 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 the order in which rows are returned. |

## SELECT examples

Syntax Definition

The following statement selects columns from a table:

```
SELECT JOB_GRADE, JOB_CODE, JOB_COUNTRY, MAX_SALARY FROM PROJECT;
```

The next statement uses the * wildcard to select all columns and rows from a table:

```
SELECT * FROM COUNTRIES;
```

The following statement uses an aggregate function to count all rows in a table that satisfy a search condition specified in the WHERE clause:

```
SELECT COUNT (*) FROM COUNTRY
  WHERE POPULATION > 5000000;
```

The next 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 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 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 statement specifies a query optimization plan for ordered retrieval, utilizing an index for ordering:

```
SELECT * FROM CITIES ORDER BY CITY
  PLAN (CITIES ORDER CITIES_1);
```

The next 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);
```

## SET GENERATOR

Syntax Example See also

### 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.

**See also**

CREATE GENERATOR

CREATE PROCEDURE

CREATE TRIGGER

GEN_ID()

## SET GENERATOR syntax

Example Definition

```
SET GENERATOR name TO int;
```

| Argument | Description |
|---|---|
| *name* | Name of an existing generator. |
| *int* | Value to which to set the generator, an integer from -231 to 231 -1. |

## SET GENERATOR example

Syntax Definition

The following 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.

## SET NAMES

Syntax Example See also

### Description

SET NAMES specifies the character set to use for subsequent database attachments. It enables ISQL to override the default character set for a database.

SET NAMES must appear before the CONNECT statement it is to affect.

**See also**

CONNECT

InterBase character sets and collation orders

## SET NAMES syntax

Example Definition

```
SET NAMES [charset];
```

| Argument | Description |
|---|---|
| *charset* | Name of a character set that identifies the active character set for a given process. Default: NONE. |

## SET NAMES example

Syntax Definition

The following statements demonstrate the use of SET NAMES:

```
SET NAMES LATIN1;
CONNECT "employee.gdb";
```

## SET STATISTICS

Syntax Example See also

### 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.

**See also**

ALTER INDEX

CREATE INDEX

DROP INDEX

## SET STATISTICS syntax

Example Definition

```
SET STATISTICS INDEX name;
```

| Argument | Description |
|---|---|
| *name* | Name of an existing index for which to recompute selectivity. |

## SET STATISTICS example

Syntax Definition

The following statement recomputes the selectivity for an index:

```
SET STATISTICS INDEX MINSALX;
```

## SET TRANSACTION

Syntax Examples See also

**Description**

SET TRANSACTION specifies the default transaction's database access, lock conflict behavior, and level of interaction with other concurrent transactions accessing the same data. It can also reserve locks for tables.

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 the default 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 the default 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.

**See also**

COMMIT

ROLLBACK

SET NAMES

## SET TRANSACTION syntax

Examples Definition

```
SET TRANSACTION [READ WRITE | READ ONLY]
  [WAIT | NO WAIT]
  [[ISOLATION LEVEL] {SNAPSHOT [TABLE STABILITY]
     | READ COMMITTED [[NO] RECORD_VERSION]}]
  [RESERVING <reserving_clause>;

<reserving_clause> = table [, table ...]
[FOR [SHARED | PROTECTED] {READ | WRITE}] [, <reserving_clause>]
```

| Argument | Description |
|---|---|
| READ WRITE | Specifies that the transaction can read and write to tables (default). |
| READ ONLY | Specifies that the transaction can only read tables. |
| WAIT | Specifies that a transaction wait for access if it encounters a lock conflict with another transaction (default). |
| 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. |

## SET TRANSACTION examples

Syntax Definition

The following statement sets up the default transaction, **gds_$trans**, 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.

```
SET TRANSACTION WAIT ISOLATION LEVEL READ COMMITTED;
```

The following statement reserves three tables:

```
SET TRANSACTION
  ISOLATION LEVEL READ COMMITTED
  NO RECORD_VERSION WAIT
  RESERVING TABLE1, TABLE2 FOR SHARED WRITE,
     TABLE3 FOR PROTECTED WRITE;
```

## SUM( )

Syntax	Example	See also

**Description**

SUM() is an aggregate function that calculates the sum of values for a column. If the number of qualifying rows is zero, SUM() returns a NULL value.

**See also**

AVG()

COUNT()

MAX()

MIN()

## SUM( ) syntax

Example Definition

```
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 or expression that evaluates to a numeric data type. |

## SUM( ) example

Syntax Definition

The following statement demonstrates the use of SUM(), AVG(), MIN(), and MAX():

```
SELECT SUM (BUDGET), AVG (BUDGET), MIN (BUDGET), MAX (BUDGET)
FROM DEPARTMENT
WHERE HEAD_DEPT = 100;
```

## UPDATE

Syntax Examples See also

**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.

The optional WHERE clause can be used to restrict updates to a subset of rows in the table. Updates cannot update array slices.

**Caution:** Without a WHERE clause, an update modifies all rows in a table.

**Note:** When updating a BLOB column, UPDATE replaces the entire BLOB with a new value.

**See also**

DELETE

GRANT

INSERT

REVOKE

SELECT

## UPDATE syntax

Examples          Definition

```
UPDATE {table | view}
SET col = <val> [, col = <val> ...]
  [WHERE <search_condition>;

<val> = {
col [<array_dim>] | <constant> | <expr> | <function>
  | NULL | USER
  }

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

**Note:** Outermost brackets (boldface) must be included when referencing arrays.

```
<constant> = num | "string" | charsetname "string"

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

<function> = {
CAST (<val> AS <datatype>)
  | UPPER (<val>)
  | GEN_ID (generator, <val>)
  }

<search_condition> = See CREATE TABLE for a full description.
```

| Argument | Description |
|---|---|
| *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_cond*> | Searched update only. Specifies the conditions a row must meet to be modified. |

## UPDATE examples

Syntax          Definition

The following statement modifies a column for all rows in a table:

```
UPDATE CITIES
  SET POPULATION = POPULATION * 1.03;
```

The next statement uses a WHERE clause to restrict column modification to a subset of rows:

```
UPDATE COUNTRY
  SET CURRENCY = "USDollar"
  WHERE COUNTRY = "USA";
```

## UPPER( )

Syntax Examples See also

### Description

UPPER() converts a specified string to all uppercase characters. If applied to character sets that have no case differentiation, UPPER() has no effect.

**See also**
CAST()

## UPPER( ) syntax

Examples     Definition

```
UPPER (<val>);
```

| Argument | Description |
|---|---|
| *<val>* | A column, constant, or expression that evaluates to a character data type. |

## UPPER( ) examples

Syntax     Definition

The following statement changes the user name, BMatthews, to BMATTHEWS:

```
UPPER (BMatthews);
```

The next 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));
```

## Write-ahead log protocol

Syntax Example

The write-ahead log (WAL) protocol is an InterBase feature that can be when creating a database on a NetWare server. The CREATE DATABASE statement defines WAL components, and the ALTER DATABASE statement modifies WAL components.

**Important:** WAL is only available on NetWare servers

The WAL protocol serves two main purposes. It:

- Improves throughput of database transactions. The WAL protocol uses buffers to group database changes and write those changes more efficiently to disk.
- Facilitates database recovery. The WAL protocol stores database changes in log files. In the event of a system crash, these log files can be applied to the database to restore it to its most recent consistent state.

Using the WAL protocol involves:

- Defining how it will operate with CREATE DATABASE. Once initialized, the WAL is started when the database is first accessed; the WAL remains active as long as any process is attached to the database.
- Maintaining it, including checking WAL performance, changing WAL parameters, and dropping the WAL.

### Components of the WAL protocol

The WAL protocol consists of the following components:

- WAL buffers.
- Log files.
- The WAL writer.
- The group commit protocol.

WAL buffers are areas of memory set aside to store database changes. When the WAL buffers fill up, or when a transaction commits the changes they contain, the changes are written to log files. You must choose one of the available log file configurations.

Log files are disk files that record the changes flushed from the WAL buffers. When a log file fills up, the next configured log file is used so that additional changes can be stored from the WAL buffers.

The WAL writer is a thread that:

- Writes WAL buffers to log files.
- Performs rollovers when log files become full. In a rollover, the WAL writer either creates a new log file or reuses an old log file.
- Manages checkpoints. A checkpoint causes the contents of the database buffers to be written to the database.

The group commit protocol combines commits from multiple transactions and writes them as a unit, thereby reducing I/O.

### WAL and forced writes

By default, the InterBase Workgroup Server for NetWare performs forced writes (also referred to as synchronous writes). When InterBase performs forced writes, it physically writes data to disk whenever it performs an (internal) write operation.

WAL performs much the same function as forced writes. If a database is using WAL, then it is not necessary to enable forced writes to ensure data integrity. Therefore, forced writes are automatically disabled when WAL is enabled for a database. Conversely, when WAL is disabled (dropped), forced writes are automatically re-enabled.

You can manually enable and disable forced writes with Server Manager. For more information about Server Manager, see the Windows Client User's Guide.

**Caution:** If neither WAL nor forced writes are enabled for a database, then the database will be subject to data loss if there is a hardware or other system failure. In general, it is best to have at least one of these features active.

**Using WAL for database recovery**

After an operating system recovers from a system failure, a database may be in an inconsistent state. When the first process attaches to a database in that state, InterBase activates automatic recovery to restore the database to a consistent state.

InterBase starts recovery by reading the log page, a database page containing the name of the current log file as well as information about the last two checkpoints. It then applies to the database all log file changes from the second-to-last checkpoint. Users can access the database as soon as the recovery is finished. The InterBase multi-generational architecture automatically finds the correct version of a row and discards unneeded or uncommitted versions.

**Note:** This recovery feature restores changes from the second-to-last checkpoint to the end of the most recent log file. Database recovery does not require any log files that contain log records generated earlier than the second-to-last checkpoint, because the information before the second-to-last checkpoint has already been saved to the actual database files.

Because log files contain records of all changes to the database, including records of changes made to the B-tree, index root, and database header pages, you do not need to perform a database backup to rebuild or recover indexes after recovery.

## Log file configurations

Three log file configurations are possible with the WAL protocol:

- The default log file configuration.
- Named series log file configuration.
- Round-robin log file configuration.

### Default configuration

With the default WAL protocol, log files reside on the same disk and directory as the database; therefore, if the disk fails, recovery is not possible. In addition, because processes are writing data both to log files and to the database, the increased disk contention can slow performance. Therefore, the default WAL configuration is recommended for prototyping only.

To use the default log file configuration, use the LOGFILE keyword with no parameters in the CREATE DATABASE statement.

In the default configuration, log files are named by using the name of the database file (without extension) as a base and appending the extension .*n*, where *n* starts at 1 and is incremented by 1 each time a new file is written. In the previous example, log files would be named STOCKS.1, STOCKS.2, and so on.

The default configuration has the following characteristics:

- The size of each log file is 100K.
- Four WAL buffers are created, each four times the database page size.
- Checkpoints occur after log files receive approximately 500K of data.
- Group commits are disabled.

### Named series configuration

The named series log file configuration creates a series of log files for the database, specifying the location of the log files and their size in kilobytes. For improved I/O, place the log files on a device other than where the database resides.

In a named series configuration, log files are named as in the default configuration, by using the name of the log file as a base and appending the extension .*n*, where *n* starts at 1 and is incremented by 1 each time a new file is written.

### Round-robin configuration

The round-robin WAL configuration creates a set of log files that are used in sequence. When the last file is filled, the WAL returns to the first log file listed in the CREATE DATABASE statement and reuses the files in the same order.

Round-robin configuration can be advantageous because it:

- Limits the number of log files.
- Specifies the directory location of log files.
- Specifies the size of log files, in kilobytes.
- Pre-allocates log files, reducing the danger of running out of log space at run time.

A round-robin log file configuration must specify an overflow log file to provide additional storage in case other log files are not available for reuse when needed. Log files will not be available if pre-allocated log files are required for recovery before two checkpoints occur. This can happen if the checkpointing interval is too large relative to the size and number of the files.

The overflow file specification follows the same rules as the named series configuration.

## Write-ahead log syntax

Example  Definition

The CREATE DATABASE syntax for defining the WAL protocol is:

```
CREATE DATABASE "dbname"
LOGFILE [configuration]
[NUM_LOG_BUFFER [=] num]
[LOG_BUFFER_SIZE [=] size]
[CHECK_POINT_LENGTH [=] length]
[GROUP_COMMIT_WAIT_TIME [=] interval];

configuration = [named_series | round-robin]

named_series = BASE_NAME "logfile" [SIZE [=] size]

round-robin =
("file_name" [SIZE [=] size] [, "file_name" [SIZE [=] size] ...])
OVERFLOW "basename" [SIZE [=] size]
```

| Argument | Description |
|---|---|
| "*dbname*" | Name of the database being created with the WAL protocol. |
| LOGFILE [*configuration*] | LOGFILE is the keyword that begins WAL protocol definition. *configuration* specifies the log file configuration to use: round-robin, or named series. If no configuration is specified, WAL uses the default log file configuration. |
| [NUM_LOG_ BUFFERS [=] *num*] | Number of WAL buffers to use.<br>Minimum = 4, Maximum = 64, Default = 4. |
| [LOG_BUFFER_SIZE [=] *size*] | Size of WAL buffers, in bytes.<br>Minimum = Four times database page size, Maximum = 64,000 bytes. Default = Four times database page size. |
| [CHECK_POINT_ LENGTH [=] *length*] | Amount of data allowed to accumulate in a log file, in kilobytes, before a checkpoint occurs.<br><br>Minimum = 100K. Default = 500K. |
| [GROUP_COMMIT_ WAIT_TIME [=] *interval*] | Number of microseconds to wait before writing committed changes from WAL buffers to log files.<br><br>Minimum = 0 (disabled).<br>Default = 0 (disabled). |
| BASE_NAME "*logfile*" [SIZE [=] *size*] | Specifies named series WAL configuration.<br><br>*logfile* - Quoted base file name of log files, at most eight characters. The WAL writer adds an extension of .n, where n starts at 1, and is incremented by 1 as each log file is created.<br><br>*size* - Size, in kilobytes, of named series log files. Default: 100K. |
| ("*file_name*" [SIZE [=] *size*] [, "*file_name*" [SIZE [=] *size*] ...]) | List of log files to use with round-robin configuration.<br><br>*file_name* - Name, including full path specification of log file. File name can be at most eight characters, with three letter extension.<br><br>*size* - Size of the preceding log file in kilobytes. Default: 100K. |
| OVERFLOW "*basename*" [SIZE [=] *size*] | Establishes temporary overflow files for use when no round-robin files are available for reuse. Required with round-robin configuration.<br><br>*basename* - Base file name of overflow file, at most eight |

characters. The WAL writer adds an extension of .n, where n starts at 1, and is incremented by 1 as each log file is created.

*size* - Size, in kilobytes, of round-robin overflow log file. Default: 100K.

## Write-ahead log example

Syntax Definition

### Default configuration

The following statement is an example of creating a database with the default log file configuration. It creates a database named USR\JOHN\STOCKS.GDB on the node, MYNODE, and the volume, VOL2:

```
CREATE DATABASE "MYNODE:VOL2:USR\JOHN\STOCKS.GDB" LOGFILE;
```

In the default configuration, log files are 100 Kb in size, and are named by using the name of the database file (without extension) as a base and appending the extension .*n*, where *n* starts at 1 and is incremented by 1 each time a new file is written. In the example, the log files would be named STOCKS.1, STOCKS.2, and so on.

### Named series configuration

The next statement is an example of creating a database with named series log file configuration. It creates NEWDB.GDB and a series of log files, each 400K in size:

```
CREATE DATABASE "NEWDB.GDB"
  LOGFILE BASE_NAME "VOL2:USR/NEWFILES"
  SIZE = 400;
```

The log file names are VOL2:USR/NEWFILES.1, VOL2:USR/NEWFILES.2, VOL2:USR/NEWFILES.3, and so forth.

### Round-robin configuration

The following statement creates NEWDB.GDB and enables a round-robin log file configuration with overflow files with base name NEWDATA:

```
CREATE DATABASE "NEWDB.GDB"
  LOGFILE ("VOL2:LOG/WAL.1" SIZE = 200,
  "VOL2:USR/LOG/WAL.2" SIZE = 200, "VOL2:USR/LOG/WAL.3" SIZE = 200)
  OVERFLOW "VOL2:USR/TEMP/NEWDATA" SIZE = 200;
```

## AVG( )

Syntax Examples See also

### Description

AVG() is an aggregate function that returns the average of the values in a specified column or expression. Only numeric data types 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.

This function may be used in SQL, DSQL, and *isql*.

## AVG( ) syntax

Example Definition

```
AVG ([ALL] <val> | DISTINCT <val>)
```

| Argument | Description |
|---|---|
| ALL | Returns the average of all values. |
| DISTINCT | Eliminates duplicate values before calculating the average. |
| <*val*> | A column or expression that evaluates to a numeric data type. |

## AVG( ) examples

Syntax Definition

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

Syntax Examples See also

### 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.

### Application

May be used in SQL.

## BASED ON syntax

Example	Definition

```
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 data type. |
| variable | Name of the host-language variable that inherits the characteristics of a database column. |

## BASED ON examples

Syntax	Definition

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

Syntax Examples See also

### 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.

### Application

May be used in SQL.

## BEGIN DECLARE SECTION syntax

Example Definition

```
BEGIN DECLARE SECTION;
```

## BEGIN DECLARE SECTION example

Syntax Definition

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

## CLOSE

Syntax Examples See also

### Description

CLOSE terminates the specified open 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.

This statement may be used in SQL.

## CLOSE syntax

Example Definition

```
CLOSE cursor;
```

| Argument | Description |
|---|---|
| cursor | Name of an open cursor. |

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.

## CLOSE example

Syntax Definition

The following embedded SQL statement closes a cursor:

```
EXEC SQL
  CLOSE BC;
```

**See also**

CLOSE (BLOB)

COMMIT

DECLARE CURSOR

FETCH

OPEN

ROLLBACK

## CLOSE (BLOB)

Syntax Examples See also

### 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.

This statement may be used in SQL.

## CLOSE (BLOB) syntax

Example Definition

```
CLOSE blob_cursor;
```

| Argument | Description |
|---|---|
| blob_cursor | Name of an open BLOB cursor. |

## CLOSE (BLOB) example

Syntax Definition

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)

## DECLARE CURSOR

Syntax Examples See also

**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. |

This statement may be used in SQL and DSQL.

## DECLARE CURSOR syntax

Example Definition

**SQL form:**

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

**DSQL form:**

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

**BLOB form:**

```
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. |

## DECLARE CURSOR examples

Syntax Definition

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)

Syntax Examples See also

**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).

This statement may be used in SQL.

## DECLARE CURSOR (BLOB) syntax

Example Definition

```
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. |

## DECLARE CURSOR (BLOB) examples

Syntax Definition

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 STATEMENT

Syntax Examples See also

**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.

This statement may be used in SQL.

## DECLARE STATEMENT syntax

Example Definition

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

| Argument | Description |
|---|---|
| *<statement>* | Name of an SQL variable for a user-supplied SQL statement to prepare and execute at run time. |

## DECLARE STATEMENT example

Syntax Definition

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

Syntax Examples See also

**Description**

DECLARE TABLE causes gpre to store a table description. A table declaration is required if a table is both created and populated 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 data types 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 data types 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.

This statement may be used in SQL.

## DECLARE TABLE syntax

Example Definition

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

| Argument | Description |
|---|---|
| table | Name of the table that will be created. Table names must be unique within the database. |
| *<table_def>* | Definition of the table. For complete table definition syntax, see CREATE TABLE. |

## DECLARE TABLE examples

Syntax Definition

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

## DESCRIBE

Syntax Examples See also

**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.

This statement may be used in SQL.

## DESCRIBE syntax

Example Definition

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

| Argument | Description |
|---|---|
| OUTPUT | Indicates that column information should be returned in the XSQLDA (default). |
| INPUT | Indicates that dynamic parameter information should be stored in the XSQLDA. |
| statement | A previously defined alias for the statement to DESCRIBE. Aliases are defined with PREPARE. |
| {INTO \| USING} SQL DESCRIPTOR xsqlda | Specifies the XSQLDA to use for the DESCRIBE statement. |

## DESCRIBE example

Syntax Definition

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

## DISCONNECT

Syntax Examples See also

**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.

This statement may be used in SQL.

## DISCONNECT syntax

Example Definition

```
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. |

## DISCONNECT examples

Syntax Definition

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

## END DECLARE SECTION

Syntax Examples See also

### Description

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

This statement may be used in SQL.

## END DECLARE SECTION syntax

Example Definition

```
END DECLARE SECTION;
```

## END DECLARE SECTION example

Syntax Definition

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

Syntax Examples See also

**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.

This statement may be used in SQL.

## EVENT INIT syntax

Example Definition

```
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. |

## EVENT INIT example

Syntax Definition

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

## EVENT WAIT

Syntax Examples See also

### 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.

This statement may be used in SQL.

## EVENT WAIT syntax

Example Definition

```
EVENT WAIT request_name;
```

| Argument | Description |
|---|---|
| request_name | Application event handle declared in a previous EVENT INIT statement. |

## EVENT WAIT examples

Syntax Definition

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

## EXECUTE

Syntax Examples See also

### 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.

This statement may be used in SQL.

## EXECUTE syntax

Example Definition

```
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. |

## EXECUTE example

Syntax Definition

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

## EXECUTE IMMEDIATE

Syntax Examples See also

**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 a statement’s parameters from an XSQLDA structure previously loaded with appropriate values.

This statement may be used in SQL.

## EXECUTE IMMEDIATE syntax

Example Definition

```
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. |

## EXECUTE IMMEDIATE example

Syntax Definition

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

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

**See also**

DESCRIBE

EXECUTE

PREPARE

## FETCH

Syntax Examples See also

### 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, data type, 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.

This statement may be used in SQL and DSQL.

## FETCH syntax

Example Definition

**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.<br>▪ Optional if FETCH gets rows for DELETE or UPDATE.<br>▪ 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. |

## FETCH examples

Syntax Definition

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

## FETCH (BLOB)

Syntax Examples See also

**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.

This statement may be used in SQL.

## FETCH (BLOB) syntax

Example Definition

```
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. |

## FETCH (BLOB) example

Syntax Definition

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)

## INSERT CURSOR (BLOB)

Syntax Examples See also

### 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.

This statement may be used in SQL.

## INSERT CURSOR (BLOB) syntax

Example Definition

```
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. |

## INSERT CURSOR (BLOB) example

Syntax Definition

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)

## OPEN

Syntax Examples See also

**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 | 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. |

This statement may be used in SQL and DSQL.

## OPEN syntax

Example Definition

**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). |

## OPEN examples

Syntax Definition

The following embedded SQL statement opens a cursor:

```
EXEC SQL
  OPEN C;
```

**See also**

CLOSE

DECLARE CURSOR

FETCH

## OPEN (BLOB)

Syntax Examples See also

**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.

This statement may be used in SQL.

## OPEN (BLOB) syntax

Example Definition

```
OPEN [TRANSACTION name] cursor
{INTO | USING} :blob_id;
```

| Argument | Description |
|---|---|
| TRANSACTION name | Specifies the transaction under which the cursor is opened.<br>Default: The default transaction. |
| cursor | Name of the BLOB cursor. |
| INTO \| USING | Depending on BLOB cursor type, use one of these:<br>▪ INTO: For INSERT BLOB.<br>▪ USING: For READ BLOB. |
| blob_id | Identifier for the BLOB column. |

## OPEN (BLOB) examples

Syntax Definition

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

Syntax Examples See also

**Description**

PREPARE readies a DSQL statement for repeated execution by:

- Checking the statement for syntax errors.
- Determining data types 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 data type, 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.

This statement may be used in SQL.

## PREPARE syntax

Example Definition

```
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. |

## PREPARE examples

Syntax Definition

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

## SET DATABASE

Syntax Examples See also

**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.

This statement may be used in SQL.

## SET DATABASE syntax

Example Definition

```
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. The alias must be unique within the program. It is 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 for run time and compile time. |
| "<*dbname*>" | Location and path name of the database associated with dbhandle. For specific platform file specifications, see that platform's operating system manuals. |
| RUNTIME | Specifies a different database to use at run time than the one specified to use during preprocessing. |
| :<*var*> | Host-language variable containing a database specification, user name, or password. |
| USER "<*name*>" | Required for PC client attachments, optional for all others. A valid user name on the server where the database resides. Used with PASSWORD to gain database access on the server. |
| PASSWORD "<*string*>" | Required for PC client attachments, optional for all others. A valid password on the server where the database resides. Used with USER to gain database access on the server. |

## SET DATABASE examples

Syntax Definition

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

## WHENEVER

Syntax Examples See also

**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.

This statement may be used in SQL.

## WHENEVER syntax

Example Definition

```
WHENEVER {NOT FOUND | SQLERROR | SQLWARNING}
{GOTO label | CONTINUE};
```

| Argument | Description |
|---|---|
| NOT FOUND | Traps SQLCODE = 100, no qualifiying 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. |

## WHENEVER example

Syntax Definition

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. */
```

**See also**

Limitations of WHENEVER Statements

Handling Errors With WHENEVER Statements

Error Handling Recovery

Changing Error-handling Routines