DDL - Data Definition Language

DDL is the abbreviation for Data Definition Language.

The task of DDL is database definition, i.e. the predefinition and manipulation of the metadata. Using different DDL commands, the database metadata can be created, altered and deleted. For example table structure, use of indices, the activation of exceptions and construction of procedures can all be defined by DDL commands. DDL commands are a subarea of SQL; the range of the SQL language is composed of DDL and DML together.

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

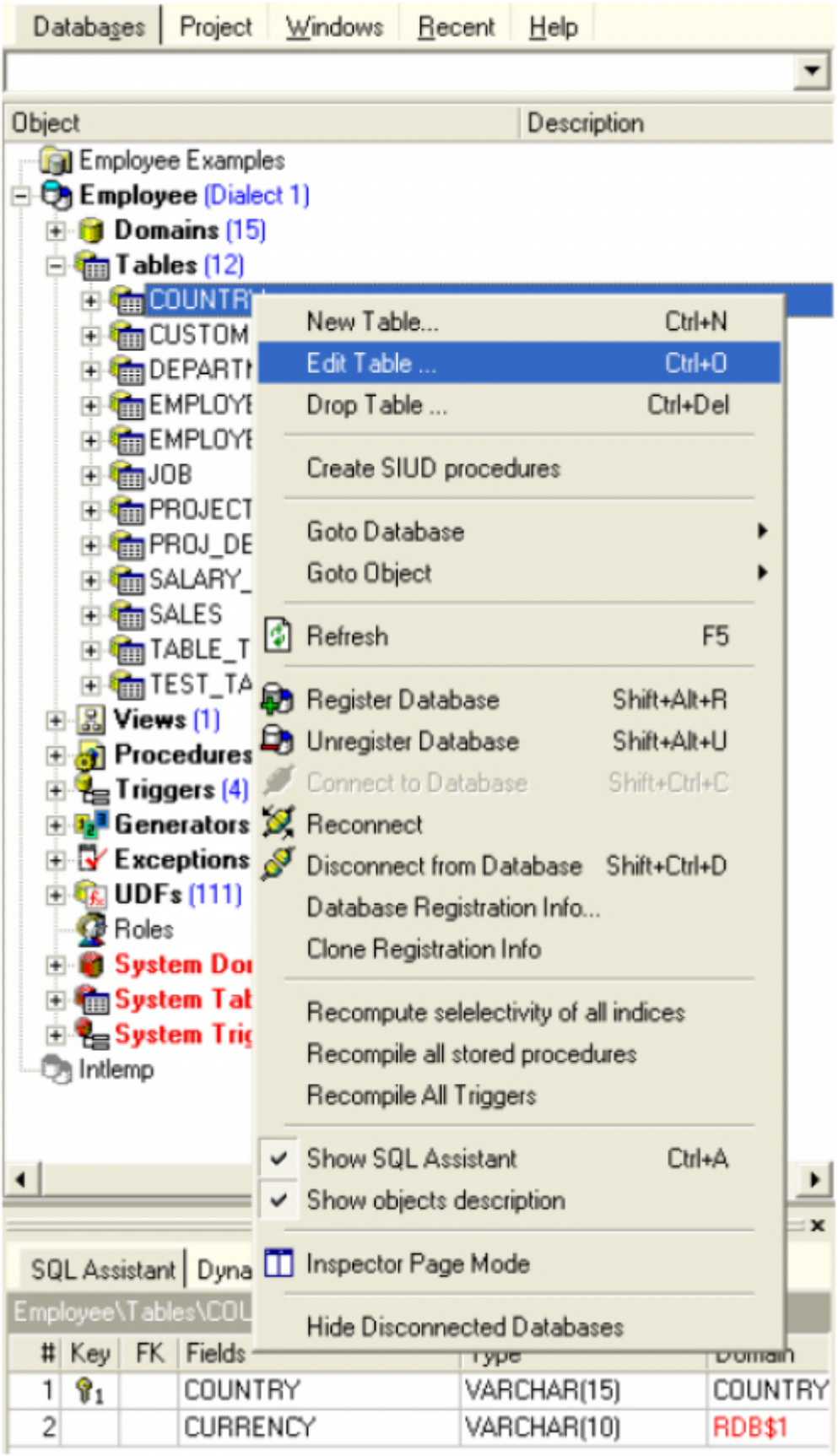
The source of all definitions included in this section is the Borland *InterBase® Language Reference*.

ALTER

ALTER is the SQL command used to modify database objects, i.e. databases, domains, tables, fields, views, triggers, procedures, generators/sequences, UDFs etc. can all be changed using the ALTER command.

The different versions of the ALTER command serve to extend or change an already defined structure, the type of alteration defined as an additional attribute of the command. This allows, for example, the metadata in already defined tables, stored procedures or triggers to be manipulated.

A database object can be altered in IBExpert using the DB Explorer right mouse button menu (Edit ...) or simply by double-clicking on the object to be altered.



Alterations can of course also be made directly in the SQL Editor.

See also:

- ALTER CHARACTER SET

- ALTER DATABASE
- ALTER DOMAIN
- ALTER EXTERNAL FUNCTION
- ALTER PROCEDURE
- ALTER SEQUENCE
- ALTER TABLE
- ALTER TRIGGER
- ALTER USER
- ALTER VIEW
- CREATE OR ALTER EXCEPTION
- CREATE OR ALTER PROCEDURE
- CREATE OR ALTER TRIGGER
- CREATE OR ALTER VIEW

back to top of page

CONNECT

A connection can be made to one or more existing databases using the CONNECT command.

The connection parameters can be specified in IBExpert using the menu item Database / Register Database. Here a specified connection may also be tested. the IBExpert menu item Services / Communication Diagnostics may be used to analyze connection problems. It delivers a detailed protocol of the test connect to a registered Firebird/InterBase® server and the results. IBExpert also offers toolbar icons for connecting, reconnecting and disconnecting to a registered database.

The CONNECT statement initializes the database data structures and determines if the database is on the originating node (local database) or on another node (remote database). An error message occurs if Firebird/InterBase® cannot locate the database. The CONNECT statement 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 recognized by the installed InterBase® version on the server.

It is possible to specify a cache buffer for the process attaching to a database. In SQL programs, a database must first be declared with the SET DATABASE command, before it can be opened with the CONNECT statement. When attaching to a database, CONNECT uses the default character set (NONE), or one specified in a previous SET NAMES statement.

A subset of CONNECT features is available in isql (see syntax below). isql can only be connected to one database at a time. Each time the CONNECT statement is used to connect to a database, previous attachments are disconnected. isql does not use SET DATABASE.

Syntax isql form

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

SQL form:
```

```
CONNECT [TO] {ALL | DEFAULT} config_opts
   | db_specs config_opts [, db_specs config_opts...];
<db_specs> = dbhandle
   | {'filespec' | :variable} AS dbhandle
<config_opts> = [USER {'username' | :variable}]
   [PASSWORD {'password' | :variable}]
   [ROLE {'rolename' | :variable}]
   [CACHE int [BUFFERS]]
```

Argument	Description
{ALL | DEFAULT}	Connects to all databases specified with SET DATABASE; options specified with CONNECT TO ALL affect all databases.
'filespec'	Database file name - can include path specification and node. The filespec must be in quotes if it includes spaces.
dbhandle	Database handle declared in a previous SET DATABASE statement; available in embedded SQL but not in isql.
:variable	Host-language variable specifying a database, user name, or password; available in embedded SQL but not in isql.
AS dbhandle	Attaches to a database and assigns a previously declared handle to it; available in embedded SQL but not in isql.
USER {'username' - :variable}	String or host-language variable that optionally specifies a user name for use when attaching to the database. The server checks the user name against the . User names are case insensitive on the server. PC clients must always send a valid user name and password.
PASSWORD {'password' - :variable}	String or host-language variable, up to 8 characters in size, that specifies password for a user listed in the security database, if used, for use when attaching to the database. The server checks the user name and password against the security database. Case sensitivity is retained for the comparison. PC clients must always send a valid user name and password.
ROLE {'rolename' - :variable}	String or host-language variable, up to 67 characters in size, which optionally specifies the role that the user adopts on connection to the database. The user must have previously been granted membership in the role to gain the privileges of that role. Regardless of role memberships granted, the user has the privileges of a role at connect time only if a ROLE clause is specified in the connection. The user can adopt at most one role per connection, and cannot switch roles except by reconnecting.
CACHE int [BUFFERS]	Sets the number of cache buffers for a database (default is 75), which determines the number of database pages a program can use at the same time. Values for int: a) Default: 256, b) Maximum value: system-dependent. This can be used to set a new default size for all databases listed in the CONNECT statement that do not already have a specific cache size, or specify a cache for a program that uses a single database. The size of the cache persists as long as the attachment is active. A decrease in cache size does not affect databases that are already attached through a server. Do not use the filespec form of database name with cache assignments.

Example

```
CONNECT C:\DB01\DB01.GDB USER SYSDBA PASSWORD masterkey
```

In the above example a connection is made to the InterBase® database DB01.GDB in the C:\DB01 directory on a Windows NT Server.

When making a connection to a UNIX server the path definitions need to be adapted accordingly:

```
CONNECT /usr/db01/db01.gdb USER SYSDBA PASSWORD masterkey
```

If the user details are not specified when performing the CONNECT command, the relevant system variables for establishing the connection to the specified database are used. This can have the consequence, that if these variables have undefined values, a database connection is not made, and instead an appropriate error message appears.

back to top of page

CREATE

CREATE is the SQL command used to create database objects, i.e. databases, domain, tables, views, triggers, procedures, generators, UDFs etc. can all be defined using the CREATE command.

A database object can be created in IBExpert using the DB Explorer right mouse button menu (New ...), the Database menu, or the respective *New Database Object* icon.

It can of course also be created, by those who are competent in SQL, directly in the SQL Editor. CREATE command syntax can be found under the respective subjects (e.g. Create Database, Create Domain, Create Table, etc.).

See also:

- CREATE DATABASE
- CREATE DOMAIN
- CREATE GENERATOR
- CREATE INDEX
- CREATE PROCEDURE
- CREATE SEQUENCE
- CREATE TABLE
- CREATE TRIGGER
- CREATE USER
- CREATE VIEW
- CREATE OR ALTER EXCEPTION
- CREATE OR ALTER PROCEDURE
- CREATE OR ALTER TRIGGER
- CREATE COLLATION (Firebird 2.1)
- CREATE EXCEPTION
- CREATE GLOBAL TEMPORARY TABLE
- CREATE ROLE
- CREATE SHADOW

back to top of page

DECLARE EXTERNAL FUNCTION (incorporating a new UDF library)

In order to use an already defined or programmed UDF (User-Defined Function) within an Firebird/InterBase® database, this has to be explicitly declared using the DECLARE EXTERNAL FUNCTION command.

The DECLARE EXTERNAL FUNCTION command syntax is as follows:

```
DECLARE EXTERNAL FUNCTION name [datatype | CSTRING (int)
[, datatype | CSTRING (int) …]]
   RETURNS {datatype [BY VALUE] | CSTRING (int) | PARAMETER n}  [FREE_IT]
   ENTRY_POINT <External_Function_Name>
   MODULE_NAME <Library_Name>;
```

By declaring the UDF, the database is informed of the following for an existing UDF (<External_Function_Name>):

Argument	**Description**
name	Name of the UDF to use in SQL statements. It can be different to the name of the function specified after the ENTRY_POINT keyword.
datatype	Datatype of an input or return parameter. All input parameters are passed to a UDF by reference. Return parameters can be passed by value. It cannot be an array element.
CSTRING (int)	Specifies a UDF that returns a null-terminated string int bytes in length.
RETURNS	Specifies the return value of a function.
BY VALUE	Specifies that a return value should be passed by value rather than by reference.
PARAMETER n	Specifies that the nth input parameter is to be returned. Used when the return datatype is a blob.
FREE_IT	Frees memory of the return value after the UDF finishes running.
<External_Function_Name>	Quoted string that contains the function name as it is stored in the library that is referenced by the UDF. The 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.
<Library_Name>	Quoted specification identifying the library that contains the UDF. The library must reside on the same machine as the Firebird/InterBase® server. On any platform, the module can be referenced with no path name if it is in. <InterBase/Firebird_home>/UDF or <InterBase/Firebird_home>/intl. If the library is in a directory other than «color #c3c3c3>InterBase/Firebird_home>/UDF</color> or InterBase/Firebird_home>/intl, you must specify its location in Firebird/InterBase®'s configuration file (ibconfig) using the EXTERNAL_FUNCTION_DIRECTORY parameter. It is not necessary to supply the extension to the module name.

The UDF name in the database does not have to correspond to the original function name. The input parameters are basically transferred BY REFERENCE. In the case of the return parameters it is also

possible to specify the form BY VALUE, using the optional BY VALUE parameter.

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

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

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

For UNIX, the statement does not include a drive letter:

```
EXTERNAL_FUNCTION_DIRECTORY \Mylibraries\InterBase
```

The Firebird/InterBase® configuration file is called ibconfig or firebird.conf on all platforms.

Examples

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

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

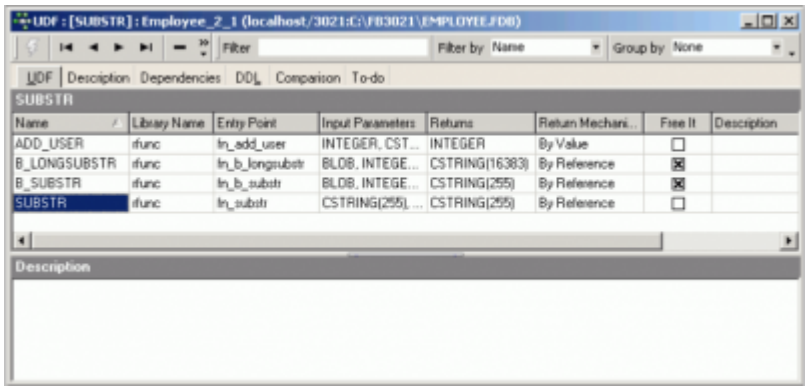
This example does not need the FREE_IT keyword because only cstrings, CHAR and VARCHAR return types require memory allocation.

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

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

In the example below (taken from the RFunc library) a function SUBSTR is declared, which calculates the substring of strings, from character i1 and length maximum i2:

```
DECLARE EXTERNAL FUNCTION SUBSTR
    CSTRING(256),
    INTEGER,
    INTEGER
RETURNS CSTRING(256)
ENTRY_POINT 'fn_substr' MODULE_NAME 'rfunc';
```



ENTRY_POINT

ENTRY_POINT is a term used in the declaration of an external function.

Syntax

```
ENTRY_POINT <External_Function_Name>
```

The entry point is a text which specifies when the function should jump into a starting address from a DLL.

MODULE_NAME

The DLL name of a UDF is entered as the last parameter when declaring an external function.

Syntax

```
MODULE_NAME <Library_Name>
```

It specifies in which UDF library the UDF can be found (<Library_Name>). Whether the file suffix needs to be entered or not, and how, is dependent upon the operating system. For example, Linux requires the suffix .SO (Shared Object Library); in Windows .DLL (Dynamic Link Library).

RETURNS

RETURNS is a term used in the declaration of an external function. Here the output parameters are specified (i.e. datatype and in which form).

Syntax

```
RETURNS <Return_Type>
```

RETURN parameters can also be specified in the form BY VALUE, using the optional BY VALUE parameter.

See also:

- External functions (UDFs)
- User-defined function (UDF)
- UDFs callable as void functions
- DECLARE EXTERNAL FUNCTION
- ALTER EXTERNAL FUNCTION
- Threaded Server and UDFs
- Passing NULL to UDFs in Firebird 2
- Using descriptors with UDFs

back to top of page

DISCONNECT

The DISCONNECT command detaches an application from one or more databases, defined by its/their database handle, and frees the relevant sources. Available in gpre.

In IBExpert there is a toolbar icon to execute this command (or alternatively use the IBExpert menu item Database / Disconnect from Database).

Syntax

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

- **ALL|DEFAULT**: Either keyword detaches all open databases.
- **dbhandle**: Previously declared database handle specifying a database to detach.

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.

Examples

The following embedded SQL statements close all databases:

```
EXEC SQL
DISCONNECT DEFAULT;

EXEC SQL
DISCONNECT ALL;
```

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

```
EXEC SQL
```

```
DISCONNECT DB1;

EXEC SQL
DISCONNECT DB1, DB2;
```

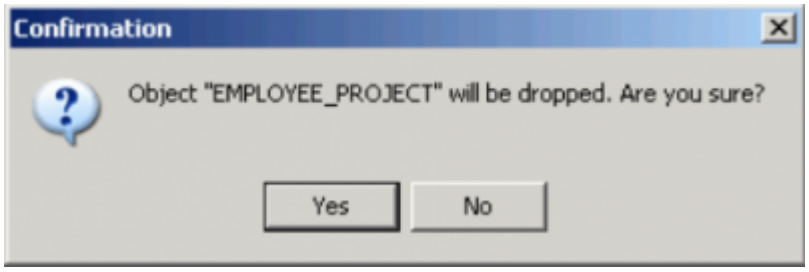
back to top of page

DROP

DROP is the SQL command used to delete database objects, i.e. databases, domains, tables, views, triggers, procedures, generators, UDFs etc. can all be deleted using the DROP command.

A database object can be dropped in IBExpert using the DB Explorer right mouse button menu (*Drop …*).

IBExpert requires confirmation of this command, as it is irreversible.



The DROP command can of course also be used directly in the SQL Editor. More information can be found under the respective subjects (e.g. Drop Database, Drop Domain, Drop Table, etc.).

Syntax

```
DROP <database_object_type> <object_name>;
```

Example

```
DROP TABLE Customer;
```

See also:

- DROP DATABASE
- Drop Domain
- DROP TABLE
- DROP VIEW
- DROP GENERATOR
- DROP PROCEDURE
- DROP SEQUENCE
- DROP TRIGGER
- DROP USER
- DROP EXCEPTION

- DROP EXTERNAL FUNCTION
- DROP FILTER
- DROP INDEX
- DROP ROLE
- DROP SHADOW

back to top of page

END DECLARE SECTION

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

Syntax

```
END DECLARE SECTION;
```

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

Example:

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

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

EXEC SQL
   END DECLARE SECTION;
```

back to top of page

EVENT

EVENT INIT

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. The command EVENT INIT is only required by embedded SQL programmers, and not required when programming the BDE.

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.

The EVENT INIT command is constructed as follows:

Syntax

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

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

Example:

The following embedded SQL statement registers interest in an event:

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

See also:

- Create Procedure
- Create Trigger
- SET DATABASE

EVENT WAIT

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

Syntax

```
EVENT WAIT request_name;
```

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

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

Examples

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

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

EXEC SQL
   EVENT WAIT ORDER_WAIT;
```

back to top of page

EXECUTE

The EXECUTE command performs a specified SQL 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.

SQL commands can be executed using the [F9] key or following icon:



enabling the SQL code to be executed and tested before finally committing.

Should a part of the text have been highlighted, only the marked portion is executed, which often causes an error message. If the execution has been successful, the SQL can be committed using the respective Icons|icon or [Ctrl + Alt + C].

Syntax

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

Argument	Description
request_name	Application event handle declared in a previous EVENT INIT statement.
TRANSACTION transaction	Specifies the transaction under which execution occurs: This clause can be used in SQL applications running multiple, simultaneous transactions to specify which transaction controls the EXECUTE operation.
USING SQL DESCRIPTOR	Specifies those values corresponding to the prepared statement's parameters should be taken from the specified XSQLDA. It need only be used for statements that have dynamic parameters.
INTO SQL DESCRIPTOR	Specifies that return values from the executed statement should be stored in the specified XSQLDA. It need only be used for DSQL statements that return values.
xsqlda	XSQLDA host-language variable.

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

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

- **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.

Example

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

```
EXEC SQL
   EXECUTE DOUBLE_SMALL_BUDGET;
```

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

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

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

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

EXECUTE PROCEDURE

Calls a specified stored procedure. Available in gpre, DSQL, and isql.

In IBExpert a procedure can be executed in the Stored Procedure Editor or SQL Editor using the [F9] key or following icon:



Syntax SQL form

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

DSQL form

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

isql form

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

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

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

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

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

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

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

Examples

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

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

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

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

back to top of page

SET

SET DATABASE

The SET DATABASE command creates a so-called database handle when creating embedded SQL applications for a specified database. It is available in gpre.

As it is possible to access several databases with embedded SQL applications, the desired database can be explicitly specified with the aid of the handle. The SET DATABASE command is only required by embedded SQL programmers and is not necessary for programming the BDE.

Syntax

```
SET DATABASE DB_Handle =
 [GLOBAL | STATIC | EXTERN]
 [COMPILETIME] [FILENAME] "<db_Name>"
 [USER "UserName" PASSWORD "PassString"]
  [RUNTIME] [FILENAME] {"<DB_Name>"|:VarDB}
 [USER {"Name"| :VarName}
  PASSWORD {"Password"| :VarPassWord=};
```

DB_Handle: This is the name of the database handle, defined by the application. It is an alias (usually an abbreviation) for a specified database. It must be unique within the program, follow the file syntax conventions for the server where the database resides, and be used in subsequent SQL statements that support database handles. For example, they can be used in subsequent CONNECT, COMMIT and ROLLBACK statements, or can also be used within transactions to differentiate table names when two or more attached databases contain tables with the same names. The optional parameters GLOBAL, STATIC and EXTERN can be used to specify the validity range of the database declaration. Following rules apply for the validity range:

Global	The database declaration is visible for all modules (default).
Static	Limits the database declaration to the current module (i.e. limit the database handle availability to the code module where the handle is declared).
Extern	References a global database handle in another module, rather than actually declaring a new handle.
Compiletime	Identifies the database used to look up column references during preprocessing. If only one database is specified in SET DATABASE, it is used both at runtime and compiletime.

Runtime	Specifies a database to use at runtime if different thatn the one specified for use during preprocessing. And if necessary, different standard users can be specified for both situations. Firebird/InterBase® sets the same database for runtime and development time as standard, if the optional parameters COMPILETIME and RUNTIME are not used.
<DB_Name>	Represents a file specification for the database to associate with db_handle. It is platform-specific.
:VarDB	This is the host-language variable containing a database specification, user name, or password.
USER and PASSWORD	Valid user name and password on the server where the database resided. Required for PC client attachments, optional for all others.

Example

```
EXEC SQL
 SET DATABASE EMPDB = 'employee.gdb'
 COMPILETIME "Test.gdb"
 RUNTIME :db_runtime;
```

SET GENERATOR

The SET GENERATOR command sets a new start value for an existing generator.

The SET GENERATOR command syntax is composed as follows:

```
SET GENERATOR Gen_Name TO int_value;
```

As soon as the function GEN_ID() enters or alters a value in a table column, this value is calculated from the int_value plus the increment defined by the GEN_ID() step parameter.

Example

```
SET GENERATOR CUST_ID_GEN TO 1030;
```

Assuming that the step parameter in the function GEN_ID() is given the value 1, the next customer would receive the customer number 1031.

This statement can also be easily and quickly performed using IBExpert's Generator Editor (please refer to Alter Generator for further information):



See also:

Firebird 2.0 Language Reference Update: SET GENERATOR

SET NAMES

The SET NAMES statement specifies an active character set to use for subsequent database attachments. Available in gpre, and isql.

Syntax

```
SET NAMES [charset | :var];
```

charset	Name of a character set that identifies the active character set for a given process; default: NONE.
:var	Host variable containing string identifying a known character set name. Must be declared as a character set name. SQL only.

SET NAMES specifies the character set to use for subsequent database attachments in an application. It enables the server to translate between the default character set for a database on the server and the character set used by an application on the client.

SET NAMES must appear before the SET DATABASE and CONNECT statements it is to affect.

Tip: Use a host-language variable with SET NAMES in an embedded application to specify a character set interactively.

Choice of character sets limits possible collation orders to a subset of all available collation orders. Given a specific character set, a specific collation order can be specified when data is selected, inserted, or updated in a column. If a default character set is not specified, the character set defaults to NONE.

Using character set NONE means that there is no character set assumption for columns; data is stored and retrieved just as it is originally entered. You can load any character set into a column defined with

NONE, but you cannot load that same data into another column that has been defined with a different character set. No transliteration is performed between the source and destination character sets, so in most cases, errors occur during assignment.

Example

The following statements demonstrate the use of SET NAMES in an embedded SQL application:

```
EXEC SQL
   SET NAMES ISO8859_1;

EXEC SQL
   SET DATABASE DB1 = 'employee.gdb';

EXEC SQL
   CONNECT;
```

The next statements demonstrate the use of SET NAMES in isql:

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

SET SQL DIALECT

SET SQL DIALECT declares the SQL dialect for database access.

n is the SQL dialect type, either 1, 2, or 3. If no dialect is specified, the default dialect is set to that of the specified compile-time database. If the default dialect is different than the one specified by the user, a warning is generated and the default dialect is set to the user-specified value. Available in gpre and isql.

Syntax

```
SET SQL DIALECT n;
```

where n is the SQL dialect type, either 1, 2, or 3.

SQL Dialect	Used for
1	InterBase® 5 and earlier compatibility.
2	Transitional dialect used to flag changes when migrating from dialect 1 to dialect 3.
3	Current Firebird/InterBase®; allows you to use delimited identifiers, exact NUMERICs, and DATE, TIME, and TIMESTAMP datatypes.

SET STATISTICS

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, which is made by the Firebird/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. Available in gpre, DSQL, and isql.

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.

Syntax:

```
SET STATISTICS INDEX name;
```

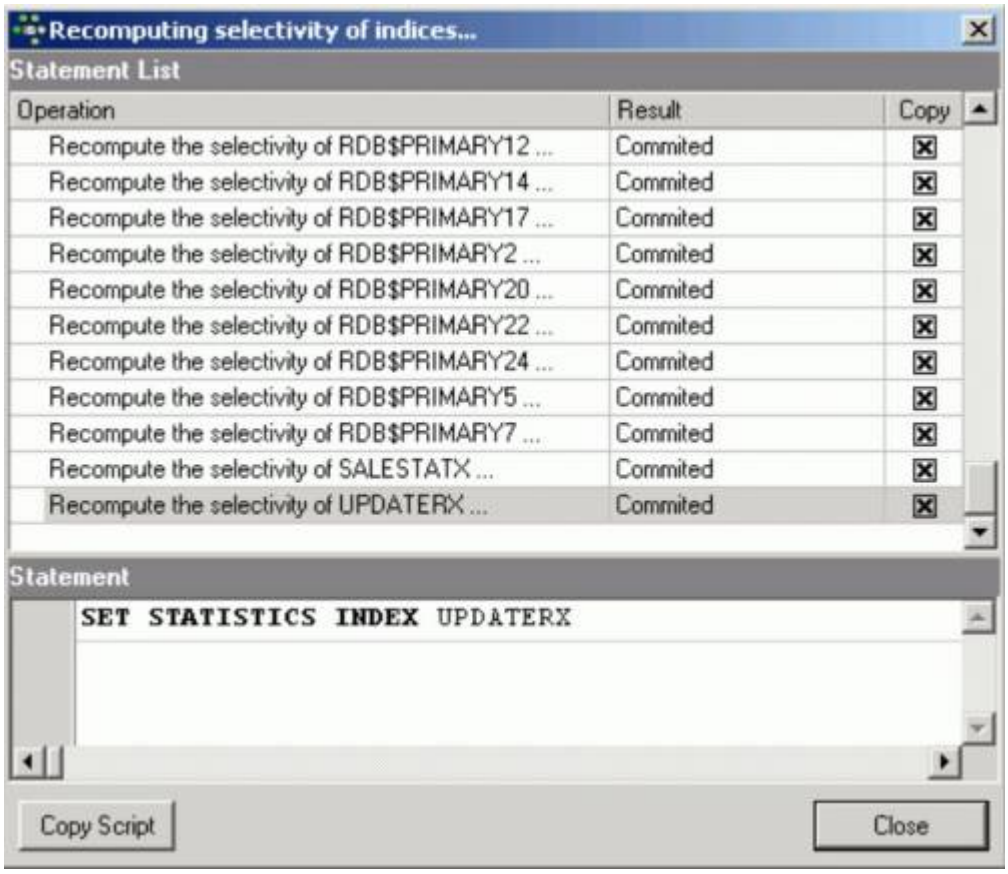
name	Name of an existing index for which to recompute selectivity.

Example:

The following embedded SQL statement recomputes the selectivity for an index:

```
EXEC SQL
   SET STATISTICS INDEX MINSALX;
```

It is possible to recompute the selectivity for all indices using the IBExpert Database menu item Recompute selectivity of all indices.



SET TRANSACTION

SET TRANSACTION starts a transaction, and optionally specifies its database access, lock conflict behavior, and level of interaction with other concurrent transactions accessing the same data. It can also reserve locks for tables. As an alternative to reserving tables, multiple database SQL applications can restrict a transaction's access to a subset of connected databases. Available in gpre, DSQL, and isql.

Important: applications preprocessed with the gpre -manual switch must explicitly start each transaction with a SET TRANSACTION statement.

Syntax

```
SET TRANSACTION [NAME transaction]
   [READ WRITE | READ ONLY]
   [WAIT | NO WAIT]
    [[ISOLATION LEVEL] {SNAPSHOT [TABLE STABILITY]
      | READ COMMITTED [[NO] RECORD_VERSION]}]
   [RESERVING reserving_clause
      | USING dbhandle [, dbhandle …]];
<reserving_clause> = table [, table …]
   [FOR [SHARED | PROTECTED] {READ | WRITE}] [, reserving_clause]
```

NAME transaction	Specifies the name for this transaction. Transaction is a previously declared and initialized host-language variable. SQL only.
READ WRITE [Default]	Specifies that the transaction can read and write to tables.
READ ONLY	Specifies that the transaction can only read tables.

WAIT [Default] | Specifies that a transaction wait for access if it encounters a lock conflict with another transaction. |

NO WAIT	Specifies that a transaction immediately return an error if it encounters a lock conflict.
ISOLATION LEVEL	Specifies the isolation level for this transaction when attempting to access the same tables as other simultaneous transactions; default: SNAPSHOT.
RESERVING reserving_clause	Reserves lock for tables at transaction start.
USING dbhandle [, dbhandle …]	Limits database access to a subset of available databases; SQL only.

Examples

The following embedded SQL statement sets up the default transaction with an isolation level of READ COMMITTED. If the transaction encounters an update conflict, it waits to get control until the first (locking) transaction is committed or rolled back.

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

The next embedded SQL statement starts a named transaction:

```
EXEC SQL
   SET TRANSACTION NAME T1 READ COMMITTED;
```

The following embedded SQL statement reserves three tables:

```
EXEC SQL
   SET TRANSACTION NAME TR1
   ISOLATION LEVEL READ COMMITTED
   NO RECORD_VERSION WAIT
   RESERVING TABLE1, TABLE2 FOR SHARED WRITE,
      TABLE3 FOR PROTECTED WRITE;
```

See also:

- Firebird 2.0 Language Reference Update: SET TRANSACTION
- Firebird 2.5 Release Notes: OldSetClauseSemantics
- SET NAMES
- COMMIT
- ROLLBACK
- Transaction options explained

back to top of page

WHENEVER

WHENEVER traps for SQLCODE errors and warnings. Every executable SQL statement returns a 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. Available in gpre.

If the appropriate condition is trapped, 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.

Syntax

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

NOT FOUND	Traps SQLCODE = 100, no qualifying rows found for the executed statement.
SQLERROR	Traps SQLCODE < 0, failed statement.

SQLWARNING	Traps SQLCODE > 0 AND < 100, system warning or informational message.
GOTO label	Jumps to program location specified by label when a warning or error occurs.
CONTINUE	Ignores the warning or error and attempts to continue processing.

Example

In the following code from an embedded SQL application, three WHENEVER statements determine which label to branch to for error and warning handling:

```
EXEC SQL
   WHENEVER SQLERROR GO TO Error; /* Trap all errors. */

EXEC SQL
   WHENEVER NOT FOUND GO TO AllDone; /* Trap SQLCODE = 100 */

EXEC SQL
   WHENEVER SQLWARNING CONTINUE; /* Ignore all warnings.
```

From:
https://www.ibexpert.com/docu/ - **IBExpert**

Permanent link:
https://www.ibexpert.com/docu/doku.php?id=01-documentation:01-09-sql-language-references:language-reference:ddl

Last update: **2023/07/18 03:29**