# Stored procedure and trigger language

The Firebird/InterBase® procedure and trigger language (which is also used for and dynamic executable blocks) includes all the constructs of a basic structured programming language, as well as statements unique to working with table data. The SQL SELECT, INSERT, UPDATE and DELETE statements can be used in stored procedures exactly as they are used in a query, with only minor syntax changes. Local variables or input parameters can be used for all of these statements in any place that a literal value is allowed. Certain constructs, including all DDL (Data Definition Language) statements, are omitted.

Firebird 2.0 introduced high performance cursor processing, for cursors originating from a SELECT query and for cursors originating from a Select procedures|selectable stored procedure. And since Firebird 2.1 domains can be used in PSQL. Please refer to Using domains in procedures for details and examples. Collations can also now be applied to PSQL variables and arguments.

Firebird 2.5 introduced several significant changes to Firebird's procedural language (PSQL), especially with regard to new extensions to the capabilities of EXECUTE STATEMENT. See below for details.

Because PSQL programs run on the server, data transfer between the relational core and the PSQL engine is very fast, much faster than transfer to a client application.

Other statements that are specific to stored procedures include, among others, error handling and raising exceptions. Please refer to the relevant sections for further information.

Note that the string concatenation operator in Firebird/InterBase® procedure and trigger language is || (a double vertical bar, or pipe), and not the + that is used in many programming languages. Please refer to concatenation of strings for further information.

Within a trigger or stored procedure, statements are separated by semicolons.

For further reading, particularly for those new to PSQL, please refer to Writing stored procedures and triggers.

# Summary of PSQL commands

| **Command** | **Description** |
|---|---|
| BEGIN <statements> END | Compound statement like in PASCAL. |
| variable = expression | Assignment. variable can be a local variable, an in or an out parameter. |
| compound_statement | A single command or a BEGIN/END block. |
| select_statement | Normal SELECT statement. The INTO clause must be present at the end of the statement. Variable names can be used with a colon preceding them. Example: SELECT PRICE FROM ARTICLES WHERE ARTNO = :ArticleNo INTO :EPrice |
| /* Comment */ | Comment, like in C. |
| – Comment | Single line SQL comment. |

| Command | Description |
|---|---|
| DECLARE VARIABLE name datatype [= startval] | Variable declaration. After AS, before the first BEGIN. |
| EXCEPTION | Re-fire the current exception. Only makes sense in a WHEN clause. |
| EXCEPTION name [message] | Fire the specified exception. Can be handled with WHEN. |
| EXECUTE PROCEDURE name arg, arg RETURNING_VALUES arg, arg | Calling a procedure. arg's must be local variables. Nesting and recursion allowed. |
| EXIT | Leaves the procedure (like in PASCAL). |
| FOR select_statement DO compound_statement | Executes compound_statement for every line that is returned by the SELECT statement. |
| IF (condition) THEN compound_statement [ELSE compound_statement] | IF statement, like in PASCAL. |
| POST_EVENT name | Posts the specified event. |
| SUSPEND | Only for SELECT procedures which return tables: Waits for the client to request the next line. Returns the next line to the client. |
| WHILE (condition) DO compound_statement | WHILE statement. Like in PASCAL. |
| WHEN {EXCEPTION a \| SQLCODE x \| ANY} DO compound_statement | Exception handling. WHEN statements must be at the end of the procedure, directly before the final END. |
| EXECUTE STATEMENT stringvalue | Executes the DML statement in stringvalue. |
| EXECUTE STATEMENT stringvalue INTO variable_list | Executes the statement and returns variables (singleton). |
| FOR EXECUTE STATEMENT stringvalue INTO variable_list DO compound_statement | Executes the statement and iterates through the resulting lines. |

(Source: Stored Procedures in Firebird by Stefan Heymann, 2004)

A complete Firebird 2.0 PSQL Language Reference including expressions, conditions and statements can be found at: https://www.janus-software.com/fbmanual/index.php?book=psql.

The most important items are listed in detail below.

back to top of page

# Using DML statements

The SQL Data Manipulation Language (DML), consists primarily of the SELECT, INSERT, UPDATE and DELETE statements.

Statements that are not recognized or permitted in the stored procedures and trigger language include DDL statements such as CREATE, ALTER, DROP, and SET as well as statements such as GRANT, REVOKE, COMMIT, and ROLLBACK.

Wherever a literal value is specified in an INSERT, UPDATE or DELETE statement, an input or local variable can be substituted in place of this literal. For example, variables can be used for the values to be inserted into a new row, or the new values in an UPDATE statement. They can also be used in a WHERE clause, to specify the rows that are to be updated or deleted.

Since Firebird 2.0, the SQL language extension EXECUTE BLOCK makes “dynamic PSQL” available to SELECT specifications. It has the effect of allowing a self-contained block of PSQL code to be executed in dynamic SQL as if it were a stored procedure. For further information please refer to EXECUTE BLOCK statement.

## Using SELECT statements

Firebird/InterBase® supports an extension to the standard SELECT statement, to solve the problem of what to do with the results when using a SELECT statement inside a stored procedure. The INTO clause appoints variables that receive the results of the SELECT statement. The syntax is as follows:

```
 SELECT <result1, result2, ..., resultN>
 FROM ...
 WHERE ...
GROUP BY ...
INTO : <Variable1, : Variable2, ..., VariableN>;
```

The INTO clause must be the final clause in the SELECT statement. A variable must be given for each result generated by the statement. *Important*: this form of SELECT statement can generate only one row. Therefore the ORDER BY clause is unnecessary here.

To use a SELECT that generates more than one row within a stored procedure, use the FOR SELECT statement.

**New to Firebird 2.0**: support for derived tables in DSQL (subqueries in FROM clause) as defined by SQL200X. A derived table is a set, derived from a dynamic SELECT statement. Derived tables can be nested, if required, to build complex queries and they can be involved in joins as though they were normal tables or views.

**Syntax**

```
SELECT
   <select list>
FROM
   <table reference list>

   <table reference list> ::= <table reference> [{<comma> <table
reference>}...]

   <table reference> ::=
      <table primary>
     | <joined table>

   <table primary> ::=
      <table> [[AS] <correlation name>]
        | <derived table>
   <derived table> ::=
      <query expression> [[AS] <correlation name>]
        [<left paren> <derived column list> <right paren>]

   <derived column list> ::= <column name> [{<comma> <column name>}...]
```

Examples can be found in the Data Manipulation Language chapter.

Points to Note

- Every column in the derived table must have a name. Unnamed expressions like constants should be added with an alias or the column list should be used.
- The number of columns in the column list should be the same as the number of columns from the query expression.
- The optimizer can handle a derived table very efficiently. However, if the derived table is involved in an inner join and contains a subquery, then no join order can be made.

See also:

- Data Retrieval
- SQL basics

back to top of page

# SET TERM terminator or terminating character

Normally InterBase® processes a script step by step and separates two statements by a semicolon. Each statement between two semicolons is parsed, interpreted, converted into an internal format and executed. This is not possible in the case of stored procedures or triggers where there are often multiple commands which need to be successively executed, i.e. there are several semicolons in their source codes. So if CREATE PROCEDURE ... was called, Firebird/InterBase® assumes that the command has finished when it arrives at the first semi colon.

In order for Firebird/InterBase® to correctly interpret and transfer a stored procedure to the database, it is necessary to temporarily alter the terminating character using the SET TERM statement. The syntax for this is as follows (Although when using the IBExpert templates this is not necessary, as IBExpert automatically inserts the SET TERM command):

```
SET TERM NEW_TERMINATOR OLD_TERMINATOR
```

**Example**

```
SET TERM ^;
CREATE PROCEDURE NAME
   AS
     BEGIN
     <procedure body>;
  END^
SET TERM ;^
```

Before the first SET TERM statement appears, Firebird/InterBase® regards the semicolon as the statement terminating character and interprets and converts the script code up until each semicolon.

Following the first SET TERM statement, the terminator is switched and all following semicolons are no longer interpreted as terminators. The CREATE PROCEDURE statement is then treated as one statement up until the new terminating character, and parsed and interpreted. The final SET TERM statement is necessary to change the terminating character back to a semicolon, using the syntax:

```
SET TERM OLD_TERMINATOR NEW_TERMINATOR
```

(refer to above example: SET TERM ;^).

The statement must be concluded by the previously defined temporary termination character. This concluding statement is again interpreted as a statement between the two last termination characters. Finally the semicolon becomes the termination character for use in further script commands.

It is irrelevant which character is used to replace the semi colon; however it should be a seldom-used sign to prevent conflicts e.g. ^, and not * or + (used in mathematical formulae) or ! (this is used for “not equal”: A!=B).

back to top of page

## SUSPEND

SUSPEND is used in stored procedures; It is used to return a row of data from a procedure to its caller. It acts as if it was a data set, i.e. returns the named data set visually as a result.

It suspends procedure execution until the next FETCH is issued by the calling application and returns output values, if there are any, to the calling application. It prevents the stored procedure from terminating until the client has fetched all the results. This statement is not recommended for executable procedures.

### Syntax

```
<suspend_stmt> ::=
   SUSPEND ;
```

Suspends execution of a PSQL routine until the next value is requested by the calling application, and returns output values, if any, to the calling application. If the procedure is called from a SELECT statement, processing will continue following SUSPEND when the next row of data is needed. Use the EXIT statement or let the code path end at the final END of the body to signal that there are no more rows to return.

If the procedure is called from a EXECUTE PROCEDURE statement, then SUSPEND has the same effect as EXIT. This usage is legal, but not recommended.

## BEGIN and END statement

As well as defining the contents of the stored procedure, these keywords also delimit a block of statements which then executes as a single statement. This means that BEGIN and END can be used to enclose several statements and so form a simple compound statement. Unlike all other PSQL statements, a BEGIN … END block is not followed by a semicolon.

See also:

Firebird 2.0 Language Reference Update: BEGIN ... END blocks

back to top of page

## DECLARE VARIABLE

Please refer to local variables.

## FOR EXECUTE INTO

Use the FOR EXECUTE INTO statement to execute a (can also be dynamically created) SELECT statement contained in a string and process all its result rows.

The execute SQL statement allows the execution of dynamically constructed SELECT statements. The rows of the result set are sequentially assigned to the variables specified in the INTO clause, and for each row the statement in the DO clause is executed.

To work with SELECT statements that return only a single row, consider using the EXECUTE INTO statement.

It is not possible to use parameter markers (?) in the SELECT statement as there is no way to specify the input actuals. Rather than using parameter markers, dynamically construct the SELECT statement, using the input actuals as part of the construction process.

## FOR SELECT ... DO ...

The FOR SELECT DO statement allows the compact processing of a SELECT statement. The rows of the result set are sequentially assigned to the variables specified in the INTO clause, and for each row the statement in the DO clause is executed.

If the AS CURSOR clause is present, the select statement is assigned a cursor name. The current row being processed by the FOR SELECT DO statement can be referred to in DELETE and UPDATE statements in the body of the FOR SELECT DO by using the WHERE CURRENT OF clause of those statements.

Examples can be found in Writing stored procedures and triggers.

back to top of page

## IF THEN ELSE

A condition is evaluated and if it evaluates to TRUE the statement in the THEN clause is executed. If it is not TRUE, i.e. It evaluates to FALSE or to NULL, and an ELSE clause is present, then the statement in the ELSE clause is executed.

IF statements can be nested, i.e. The statements in the THEN or ELSE clauses can be IF statements also. If the THEN clause contains a IF THEN ELSE statement, then that ELSE clause is deemed to be part of the nested IF, just as in nearly all other programming languages. Enclose the nested IF in a compound statement if you want the ELSE clause to refer to the enclosing IF statement.

```
variable = expression;
```

The variable can be an input or output parameter, or a local variable defined in a DECLARE VARIABLE statement. The expression needs to be concluded with a semicolon. The syntax for the IF statement is as follows:

```
IF <conditional_test>
THEN
<statements>;
ELSE
<statements>;
```

Any of the standard comparison operators available in SQL an be used (please refer to comparison operators for a full list).

The value can be a constant or one of the input parameters, output parameters or local variables used in the procedure.

If a single statement is placed after the THEN or ELSE clauses, it should be terminated with a semicolon.

If multiple statements need to be placed after one of these clauses, use the BEGIN and END keywords as follows:

```
IF <conditional_test> THEN
BEGIN
<statement1>;
<statement2>;
...
```

```
<statementN>;
END
ELSE
etc.;
```

See also:

Firebird Null Guide: Conditional statements and loops

back to top of page

## WHILE and DO

The WHILE … DO statement provides a looping capability. The syntax for this statement is as follows:

```
WHILE
<conditional_test>
DO
<statements>;
```

Firebird/InterBase® evaluates the conditional test. If it is TRUE, the statements following the WHILE are executed. If it is FALSE, the statements are ignored. If only one statement is placed after the DO clause, it should be terminated with a semicolon. If multiple statements are used after one of these clauses, use the BEGIN and END keywords. Brackets need to be put around the conditional test.

## OPEN CURSOR

New to Firebird 2.0, the OPEN statement allows you to open a local cursor.

**Syntax**

```
<open_stmt> ::=
   OPEN <cursor_name>;

<cursor_name> ::=   <identifier>
```

where cursor_name is the name of a local cursor.

The OPEN statement opens a local cursor. Opening a cursor means that the associated query is executed and the that the result set is kept available for subsequent processing by the FETCH statement. The cursor must have been declared in the declarations section of the PSQL program.

Attempts to open a cursor that is already open, or attempts to open a named FOR SELECT cursor will fail and generate a runtime exception. All cursors which were not explicitly closed will be closed automatically on exit from the current PSQL program.

Please also refer to Explicit cursors in the Firebird 2.0.4 Release Notes.