# Writing Stored Procedures and Triggers

The stored procedure and trigger language is a language created to run in a database. For this reason its range is limited to database operations and necessary functions; PSQL is in itself however a full and powerful language, and offers more functionalities than you can use if you were just sat on the client. The full range of keywords and functions available for use in procedures and triggers can be found in the Structured Query Language chapter, Stored Procedure and Trigger Language. New features can be found in the Firebird 2 Release Notes.

Firebird/InterBase® provides the same SQL extensions for use in both stored procedures and triggers. These include the following statements:

- DECLARE VARIABLE
- BEGIN ... END
- SELECT ... INTO : variable_list
- Variable = Expression
- /* comments */
- EXECUTE PROCEDURE
- FOR select DO ...
- IF condition THEN ... ELSE ...
- WHILE condition DO ...

and the following Firebird 2 features:

- DECLARE <cursor_name> CURSOR FOR ...
- OPEN <cursor_name>
- FETCH <cursor_name> INTO ...
- CLOSE <cursor_name>
- LEAVE <label>
- NEXT VALUE FOR <generator>

Both stored procedure and trigger statements includes SQL statements that are conceptually nested inside the main statement. In order for Firebird/InterBase® to correctly parse and interpret a procedure or trigger, the database software needs a way to terminate the CREATE PROCEDURE or CREATE TRIGGER that is different from the way the statements inside the CREATE PROCEDURE/TRIGGER are terminated. This can be done using the SET TERM statement.

back to top of page

### Stored procedure

Firebird/InterBase® uses stored procedures as the programming environment for integrating active processes in the database. Please refer to the IBExpert documentation chapter, Stored Procedure for the definition, description and variables of a stored procedure along with comprehensive instructions of how to use IBExpert's Stored Procedure Editor.

There are two types of stored procedure: executable and selectable. An executable procedure returns no more than one set of variables. A select procedure can, using the SUSPEND keyword, push back variables, one data set at a time. If an EXECUTE PROCEDURE statement contains a SUSPEND, then

SUSPEND has the same effect as EXIT. This usage is legal, but not recommended, and it is unfortunately an error that even experienced programmers often make.

The syntax for declaring both types of stored procedure is the same, but there are two ways of invoking or calling one: either a stored procedure can act like a functional procedure in another language, in so far as you execute it and it either gives you one answer or no answers:

```
execute procedure <procedure_name>
```

It just goes away and does something. The other is to make a stored procedure a little more like a table, in so far as you can

```
select * from <procedure_name>
```

and get data rows back as an answer.

Further reading:

- Stored procedure
- EXECUTE PROCEDURE
- Stored procedure and trigger language
- Stored procedure language

back to top of page

### Simple procedures

An example of a very simple procedure that behaves like a table, using SUSPEND to provide the returns:

```
CREATE PROCEDURE DUMMY
RETURNS (TXT VARCHAR(10))
AS
BEGIN
  TXT='DOG';
  SUSPEND;
  TXT='CAT';
  SUSPEND;
  TXT='MOUSE';
  SUSPEND;
END
```

In this example, the return variable is TXT. The text DOG is entered, and by specifying SUSPEND the server pushes the result, DOG into the buffer onto a result set stack. When the next data set is written, it is pushed onto the result pile. Using SUSPEND in a procedure, allows data definition that is not possible in this form in an SQL. It is an extremely powerful aid, particularly for reporting.

### FOR SELECT ... DO ...SUSPEND

```
CREATE PROCEDURE SEARCH_ACTOR(
    NAME VARCHAR(50))
RETURNS (
    TITLE VARCHAR(50),
    ACTOR VARCHAR(50),
    PRICE NUMERIC(18,2))
AS
BEGIN
  FOR
    select TITLE,ACTOR,PRICE from product
    where actor containing :name
    INTO :TITLE,:ACTOR,:PRICE
  DO
  BEGIN
    SUSPEND;
  END
END
```

This procedure is first given a name, SEARCH_ACTOR, then an input parameter is specified, so that the user can specify which name he wishes to search for. The columns to be returned are TITLE, ACTOR and PRICE. The procedure then searches in a FOR ...SELECT loop for the relevant information in the table and returns any data sets meeting the condition in the input parameter.

It is also possible to add conditions; below all films costing more that $30.00 are to be rounded down to $30.00:

```
CREATE PROCEDURE SEARCH_ACTOR(
    NAME VARCHAR(50))
RETURNS (
    TITLE VARCHAR(50),
    ACTOR VARCHAR(50),
    PRICE NUMERIC(18,2))
AS
BEGIN
  FOR
    SELECT TITLE,ACTOR,PRICE FROM PRODUCT
    WHERE ACTOR CONTAINING :NAME
    INTO :TITLE,:ACTOR,:PRICE
  DO
  BEGIN
    IF (PRICE<30)THEN PRICE=30
    SUSPEND;
  END
END
```

A good way of analyzing such procedures is to view them in the IBExpert Stored Procedure and Trigger Debugger.

To proceed further, the number of returns can be limited, for example, FIRST 10:

```
CREATE PROCEDURE SEARCH_ACTOR(
    NAME VARCHAR(50))
RETURNS (
    TITLE VARCHAR(50),
    ACTOR VARCHAR(50),
    PRICE NUMERIC(18,2))
AS
BEGIN
  FOR
    SELECT FIRST 10 TITLE,ACTOR,PRICE FROM PRODUCT
    WHERE ACTOR CONTAINING :NAME
    INTO :TITLE,:ACTOR,:PRICE
  DO
  BEGIN
    IF (PRICE<30)THEN PRICE=30
    SUSPEND;
  END
END
```

If you declare a variable for the FIRST statement, it needs to be put into brackets when referred to lower down in the procedure:

```
CREATE PROCEDURE SEARCH_ACTOR(
    NAME VARCHAR(50))
RETURNS (
    TITLE VARCHAR(50),
    ACTOR VARCHAR(50),
    PRICE NUMERIC(18,2))
AS
DECLARE VARIABLE i INTEGER;
BEGIN
  FOR
    SELECT FIRST (:i) TITLE,ACTOR,PRICE FROM PRODUCT
    WHERE ACTOR CONTAINING :NAME
    INTO :TITLE,:ACTOR,:PRICE
  DO
  BEGIN
    IF (PRICE<30)THEN PRICE=30
    SUSPEND;
  END
END
```

back to top of page

**FOR EXECUTE ... DO ...**

EXECUTE STATEMENT allows statements to be used in procedures, allowing dynamic SQLs to be

executed contained in a string expression. Here, the above example has been adapted accordingly:

```
CREATE PROCEDURE SEARCH_ACTOR(
  NAME VARCHAR(50))
RETURNS (
  TITLE VARCHAR(50),
  ACTOR VARCHAR(50),
  PRICE NUMERIC(18,2))
AS
Declare variable i integer;
BEGIN
  i=10;
  FOR
    execute statement
    'select first '|| :I ||' TITLE,ACTOR,PRICE from product
    where actor containing '''||name||''''
    INTO :TITLE,:ACTOR,:PRICE
  DO
  BEGIN
    if (price>30) then price=30;
    SUSPEND;
  END
END
```

It is also possible to define the SQL as a variable:

```
CREATE PROCEDURE SEARCH_ACTOR(
  NAME VARCHAR(50))
RETURNS (
  TITLE VARCHAR(50),
  ACTOR VARCHAR(50),
  PRICE NUMERIC(18,2))
AS
Declare variable i integer;
Declare variable SQL varchar(1000);
BEGIN
  i=10;
  Sql =  'select first '|| :i ||' TITLE,ACTOR,PRICE from product
          where actor containing '''||name||''''
  FOR
    execute statement :sql
    INTO :TITLE,:ACTOR,:PRICE
  DO
  BEGIN
    if (price>30) then price=30;
    SUSPEND;
  END
END
```

Theoretically it is possible to store complete SQL statements in the database itself, and they can be called at any time. It allows an enormous flexibility and a high level of user customization. Using such

dynamic procedures allows you to define your SQL at runtime, making on the fly alterations as the situation may demand.

Note that not all SQL statements are allowed. Statements that alter the state of the current transaction (such as COMMIT and ROLLBACK) are not allowed and will cause a runtime error.

The INTO clause is only meaningful if the SQL statement returns values, such as SELECT, INSERT ... RETURNING or UPDATE ... RETURNING. If the SQL statement is a SELECT statement, it must be a 'singleton' SELECT, i.e. it must return exactly one row. To work with SELECT statements that return multiple rows, use the FOR EXECUTE INTO statement.

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

back to top of page

### WHILE ... DO

The WHILE ... DO statement also provides a looping capability. It repeats a statement as long as a condition holds true. The condition is tested at the start of each loop.

### LEAVE and BREAK

LEAVE and BREAK are used to exit a loop. You may want to exit a loop because you've found the information you were looking for, or you only require, for example, the first 50 results.

By issuing a BREAK, if a specified condition isn't met, the procedure will break out of this loop and carry on executing past it, i.e. you go out of the layer you're in and proceed to the next one.

LEAVE is new to Firebird 2.0. The LEAVE statement also terminates the flow in a loop, and moves to the statement following the END statement that completes that loop. It is only available inside of WHILE, FOR SELECT and FOR SELECT ... DO ...SUSPEND#FOR EXECUTE ... DO ...|FOR EXECUTE]] statements, otherwise a syntax error is thrown.

The LEAVE «color #c3c3c3>label</color» syntax allows PSQL loops to be marked with labels and terminated in Java style. They can be nested and exited back to a certain level using the «color #c3c3c3>label</color» function. Using the BREAK statement this is possible using flags.

```
CNT = 100;
L1:
WHILE (CNT >= 0) DO
   BEGIN
   IF (CNT < 50) THEN
      LEAVE L1; -- exists WHILE loop
   CNT = CNT – l;
   END
```

The purpose is to stop execution of the current block and unwind back to the specified label. After

that execution resumes at the statement following the terminated loop. Don't forget to specify the condition carefully, otherwise you could end up with an infinite loop! As soon as you insert your WHILE loop, specify whatever should cause the loop to finish.

Note that LEAVE without an explicit label means interrupting the current (most inner) loop:

```
FOR SELECT  ... INTO ......
DO
  BEGIN
  IF () THEN
     SUSPEND;
  ELSE
     LEAVE; -- exits current loop
  END
```

The Firebird 2.0 keyword LEAVE deprecates the existing BREAK, so in new code the use of LEAVE is preferred.

back to top of page

#### EXECUTE statement

To create a simple table statistic, we can create a new procedure, TBLSTATS:

```
CREATE PROCEDURE TBLSTATS
RETURNS (
  table_name VARCHAR(100),
  no_recordsInteger)
BEGIN
  FOR SELECT r.rdb$relation_name FROM rdb$relations r
     WHERE r.rdb$relation_name NOT CONTAINING '$'
  INTO :table_name
  DO
  BEGIN
     EXECUTE STATEMENT 'select count (*) from '||:table_name into
:no_records;
  END
  SUSPEND;
END
```

This TBLSTATS fetches a table and a count, and goes through all tables, pushes the table names in and counts all data sets in the database, allowing you to see how large your tables are.

back to top of page

#### Recursions and modularity

If a procedure calls itself, it is recursive. Recursive procedures are useful for tasks that involve

repetitive steps. Each invocation of a procedure is referred to as an instance, since each procedure call is a separate entity that performs as if called from an application, reserving memory and stack space as required to perform its tasks.

Stored procedures can be nested up to 1,000 levels deep. This limitation helps to prevent infinite loops that can occur when a recursive procedure provides no absolute terminating condition. Nested procedure calls may be restricted to fewer than 1,000 levels by memory and stack limitations of the server.

Recursive procedures are often built for tree structure. For example:

```
Create procedure spx
(inp integer)
returns
(outp integer)
as
declare variable vx integer;
declare variable vy integer;
begin
  ...
  execute procedure spx(:vx) returning values :vy;
  ...
end
```

The input integer is defined and the variables computed in some way. Then the procedure calls itself and the returning values are returned to another variable.

A good example of this is a typical employee table in a large hierarchical company, where the table has a column containing a pointer to the employees' boss. Every employee has a boss, and the bosses have bosses, who may also have bosses. If you wished to see a list of all bosses for one individual or the upstream management, then you could create a procedure selecting into and finish this with a suspend. Then it would go and call the same procedure again, this time with the resulting boss's ID. The procedure would carry on in this way until it reached the top level management, who answer to no one (the CEO).

back to top of page

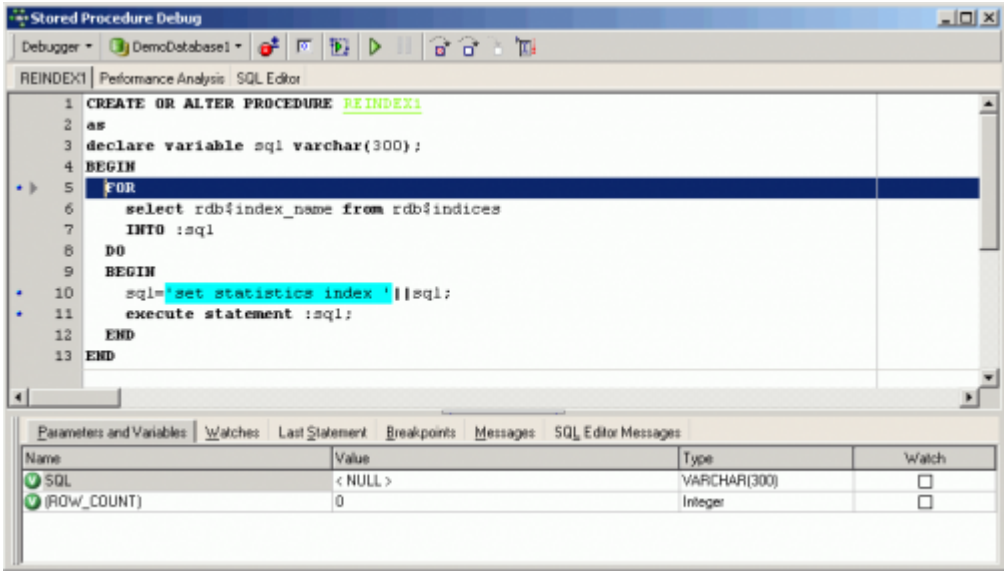
# Debugging

Up to Firebird version 2.1, Firebird offered no integrated debugging API at all. The only solution was to create log tables or external tables to record what the procedure was doing, and try to debug that way. However, as your triggers and procedures become more complex, an intelligent and sound debugging tool is vital.

### Stored procedure and trigger debugger

IBExpert has an integrated Stored Procedure and Trigger Debugger which simulates running a

procedure or trigger on the database server by interpreting the procedure and running the commands one at a time.




It offers a number of useful functionalities, such as *breakpoints, step into, trace or run to cursor*, you can watch certain parameters, analyze the performance and indices used, and you can even change values on the fly. If you have Delphi experience you will easily find your way around the Debugger as key strokes etc. are the same.

Please refer to the IBExpert documentation chapter, Debug procedure or trigger (IBExpert Debugger) for details.

back to top of page

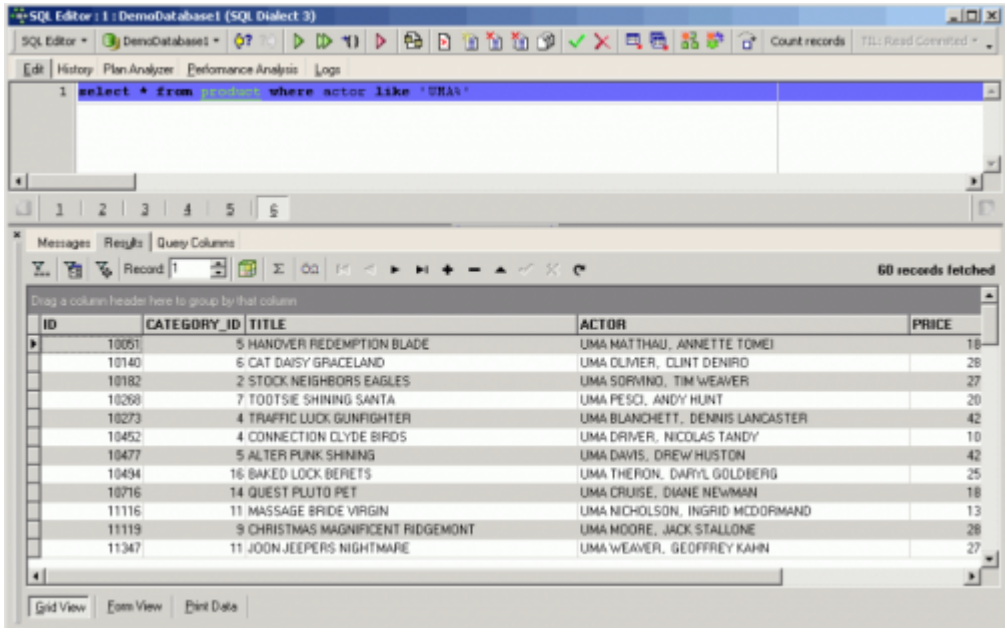
### Optimizing procedures

Procedure operations are planned on Prepare, which means that the index plan is created upon the first prepare. When working with huge amounts of data, it is critical that you write it, rewrite it, look at each of the SQLs in it and break it down to ensure that it is optimally set up. A major contributing factor to the performance and efficiency of procedures are indices. The subject of indices is an extensive subject, which has been covered in detail in other areas of this documentation site:

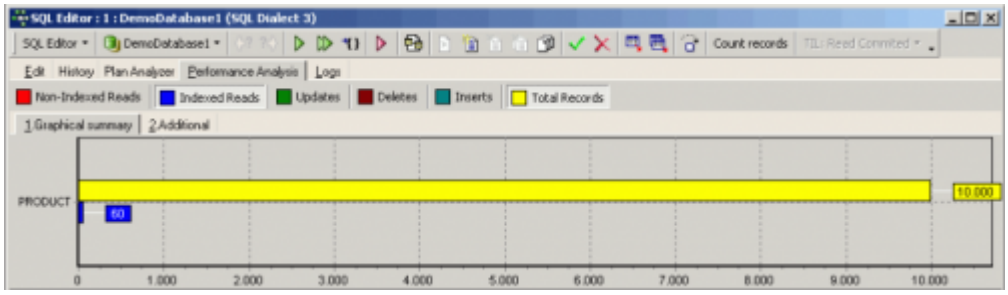
- Index
- SQL Editor / Plan Analyzer
- SQL Editor / Performance Analysis
- Using the PLAN operator
- IBExpert Table Editor / Indices
- Recompute selectivity of all indices
- Firebird Administration using IBExpert: The Firebird Optimizer and index statistics
- Firebird Administration using IBExpert: Automating the recalculation of index statistics
- Firebird for the database expert: Episode 1 - Indexes
- Enhancements to indexing in Firebird 2.0

Also take into consideration the use of operators such as LIKE and CONTAINING, as well as the use of strings such as %STRING%, as none of these can use indices. For example, in the DemoDB, db1, compare:

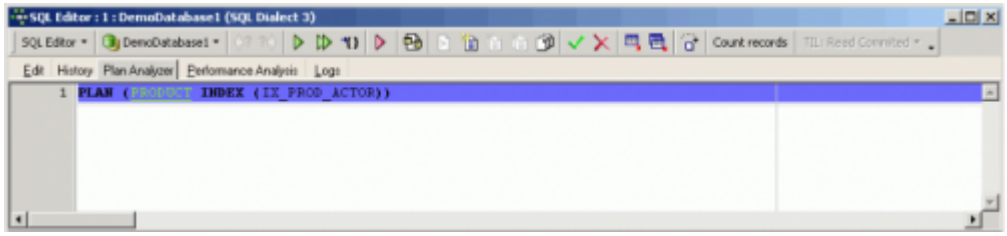
```
select * from product where actor like 'UMA%'
```




The server returns all data sets beginning with the name UMA. If you examine the Performance Analysis:

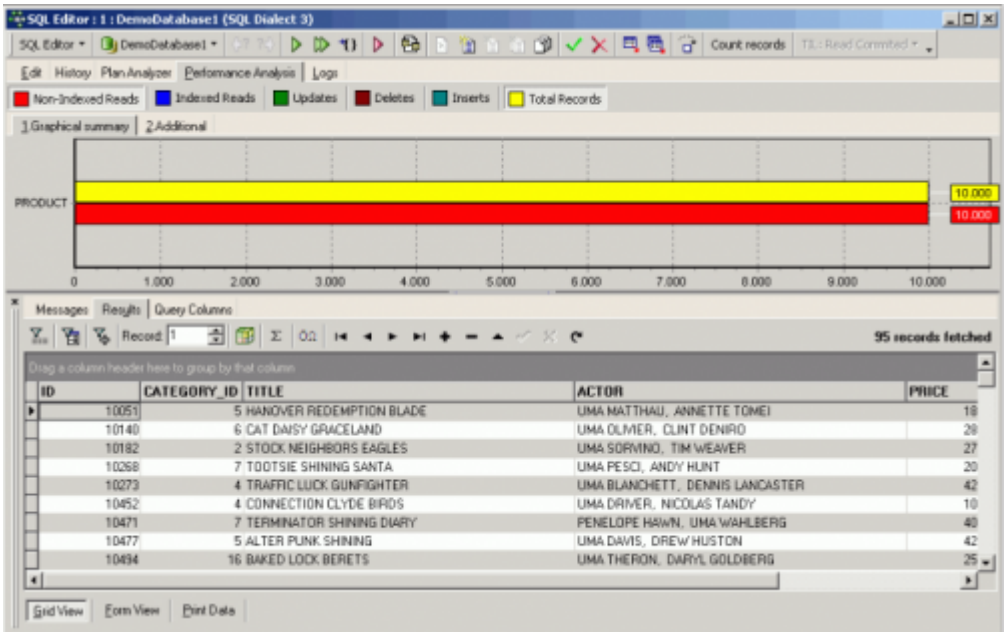



you will see that 60 indexed read operations were performed, and the Plan Analysis shows that the IX_PROD_ACTOR index was used:




If however you need to view all records where the name UMA appears somewhere in the ACTOR field:

```
select * from product where actor like ''
```




Now the server has had to perform 10,000 non-indexed reads to fetch 95 records, rather more than the 60 reads for the 60 resulting records in the last example!

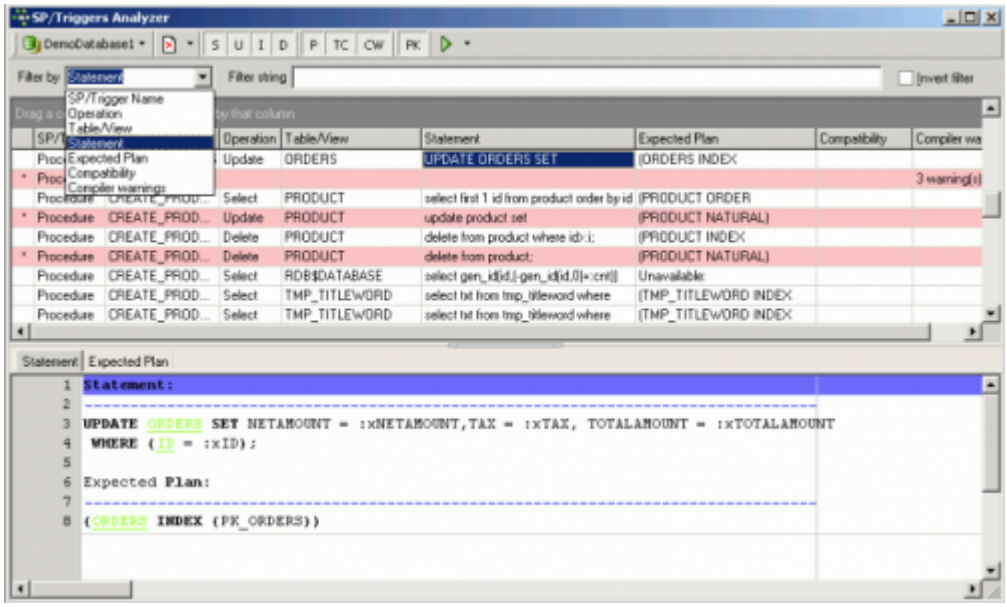
So if you can, use STARTING WITH instead of LIKE or CONTAINING. Check each procedure operation individually and remove bottlenecks, use the debugger and the SP/Triggers/Views Analyzer, check the index plans, not forgetting to recompute the selectivity of your indices regularly. Check for indices on columns used in WHERE and JOIN clauses. Use the Plan Analyzer and Performance Analysis to help you compare and improve your more complex procedures.

Another consideration with extremely complex procedures is to postpone the SUSPEND. If you have a SUSPEND on every data row on a report that may be returning thousands of rows of calculated results, it will slow your system. If you wish to have an element of control over it, then put your SUSPEND every 100 or 1,000 rows. This way the database server fills a buffer and sends the results back in the specified quantity. It makes it more manageable, and you can stop it at any time should it congest your system too much.

back to top of page

### Using the SP/Triggers/Views Analyzer

A quick and easy method to review all your procedures (and triggers and views) is to use the IBExpert Tools menu item, SP/Triggers/Views Analyzer.




This allows you to analyze a selection of actions for all or a filtered selection of procedures, triggers and views in a database, providing information by statement, displaying plans and indices used, issuing compatibility warnings and compiler warnings for all objects analyzed. Please refer to the IBExpert chapter, SP/Triggers/Views Analyzer for details.

back to top of page

### Complex SELECTs or selectable stored procedures?

Selectable procedures can sometimes offer higher performance than complex selects. For example:

```
CREATE PROCEDURE SPPROD
RETURNS (TITLE VARCHAR(50),TXT VARCHAR(20))
AS
declare variable cid bigint;
BEGIN
  FOR                                  --outer select
    Select p.title,p.category_id
    from product p
    INTO :TITLE,:cid
  DO
  BEGIN
    select c.txt from category c
    where c.id=:cid into :txt;         --inner select
    SUSPEND;
  END
END
```

This simple example is mimicking a join. You have a procedure here which is going to return a title and some text. First it goes through all the products, selecting the relevant titles. This outer select is however only providing one of the output fields. So another select is nested within the procedure, providing the information for the second output field, cid.

Although some developers feel there's no reason to construct procedures this way, ever so often you will find that the optimizer really has a problem with a certain join, because it takes too long for it to work out how to approach the query. Breaking things down like this can actually often provide a more immediate response.

back to top of page

### Trigger

A trigger on the other hand is a special table- or database-bound procedure that is started automatically. After creating your database and constructing your table structure, you need to get your triggers sorted. Triggers are extremely powerful - the so-called police force of the database. They ensure database integrity because you just can't get round them. You, the developer, tell the system how to invoke them and whether they should react to an INSERT, UPDATE or DELETE. And once we're there in a table inserting, updating or deleting, it is impossible not to execute them. You can specify whether your trigger should fire on an INSERT or an UPDATE or a DELETE, or on all three actions (universal trigger).

Comprehensive details concerning triggers, how to create them, the different types and variables can be found in the IBExpert documentation chapter, Trigger.

Don't put all your logic into one trigger, build up layers of them, e.g. one for generating the primary key, one for logging or replication, one for passing on information of the data manipulation to another table etc. The order in which such a series of triggers is executed can be important. The before insert logging trigger needs to know the primary key, so the before insert primary key trigger needs to be fired first. The firing position is user-defined, beginning with 0. Please refer to Trigger position in the IBExpert documentation chapter, Trigger.

back to top of page

### Using procedures to create and drop triggers

```
CREATE EXCEPTION ERRORTXT 'ERROR';
CREATE PROCEDURE createautoinc
AS
declare variable sql varchar(500);
declare variable tbl varchaR(30);
BEGIN
  FOR
    select rdb$relation_name from rdb$relations r
    where r.rdb$relation_name not containing '$'
    INTO :TBL
  DO
  BEGIN
     sql='CREATE trigger '||:tbl||'_bi0 for '||:tbl||' '||
         'active before insert position 0 AS '||
         'BEGIN '||
         '  if (new.id is null) then '||
         '  new.id = gen_id(id, 1); '||
```

```
          'END';
       execute statement :sql;
  END
  when any do exception errortxt :tbl;
END
```

This is a simple procedure which uses all table names (all tables are stored in rdb$relations) and creates a BEFORE INSERT trigger which adds an autoincrement ID. The following procedure then drops the trigger:

```
CREATE PROCEDURE dropautoinc
AS
declare variable sql varchar(500);
declare variable tbl varchaR(30);
BEGIN
  FOR
    select rdb$relation_name from rdb$relations r
    where r.rdb$relation_name not containing '$'
    INTO :TBL
  DO
  BEGIN
     sql='DROP trigger '||:tbl||'_bi0;';
     execute statement :sql;
  END
  when any do exception errortxt :tbl;
END
```

back to top of page

### Using domains in stored procedures

Introduced in Firebird 2.1, this feature finally allows developers to declare local variables and input and output arguments for stored procedures using domains in lieu of canonical data types. In earlier Firebird versions it was necessary to write the data type of the domain instead of the domain name. This meant a time-consuming checking of domain data types, which then had to be written in the procedure definition. For example:

```
create procedure insert_orderline(
  article_name varchar(50),
  price decimal(15,2)
  active smallint
)
begin
  ...
end
```

In Firebird 2.1 you can either type the domain name if you also want any CHECK clauses and default values to be taken into consideration, or use the TYPE OF keyword if you just want the data type. The above example would then look something like this:

```
create procedure insert_orderline(
  article_name string,
  price currency,
  active bool
)
begin
  ...
end
```

From:
http://ibexpert.com/docu/ - **IBExpert**

Permanent link:
**http://ibexpert.com/docu/doku.php?id=01-documentation:01-06-white-papers:firebird-development-using-ibexpert:writing-stored-procedures-and-triggers**

Last update: **2023/06/21 12:00**