

# Procedural SQL (PSQL)

Several significant changes appear in Firebird's procedural language (PSQL), the language set for triggers, stored procedures and dynamic executable blocks, especially with regard to new extensions to the capabilities of `EXECUTE STATEMENT`. This release also heralds the arrival of the [autonomous transaction](#).

## Quick links

- [Autonomous transactions](#)
- [Borrow database column type for a PSQL variable](#)
- [New extensions to EXECUTE STATEMENT](#)
  - [Context issues](#)
    - [Authentication](#)
    - [Transaction behaviour](#)
    - [Inherited access privileges](#)
  - [External queries from PSQL](#)
  - [EXECUTE STATEMENT with dynamic parameters](#)
  - [Examples using EXECUTE STATEMENT](#)
- [Other PSQL improvements](#)
  - [Subqueries as PSQL expressions](#)
  - [SQLSTATE as a context variable](#)

## Autonomous transactions

Adriano dos Santos Fernandes

Tracker reference [CORE-1409](#).

This new implementation allows a piece of code to run in an autonomous [transaction](#) within a PSQL module. It can be handy for a situation where you need to raise an [exception](#) but do not want the database changes to be [rolled back](#).

The new transaction is initiated with the same [isolation level](#) as the one from which it is launched. Any exception raised in a block within the autonomous transaction will cause changes to be rolled back. If the block runs through until its end, the transaction is committed.

*Warning:* Because the autonomous transaction is independent from the one from which is launched, you need to use this feature with caution to avoid deadlocks.

## Syntax pattern

```
IN AUTONOMOUS TRANSACTION
DO
<simple statement | compound statement>
```

## Example of use

```
create table log (  
    logdate timestamp,  
    msg varchar(60)  
);  
  
create exception e_conn 'Connection rejected';  
  
set term !;  
  
create trigger t_conn on connect  
as  
begin  
    if (current_user = 'BAD_USER') then  
        begin  
            in autonomous transaction  
            do  
                begin  
                    insert into log (logdate, msg) values (current_timestamp, 'Connection  
rejected');  
                end  
  
                exception e_conn;  
            end  
        end!  
  
set term ;!
```

[back to top of page](#)

## Borrow database column type for a PSQL variable

Adriano dos Santos Fernandes

Tracker reference [CORE-1356](#).

This feature extends the implementation in v.2 whereby [domains](#) became available as [data types](#) for declaring variables in [PSQL](#). Now it is possible to borrow the data type of a column definition from a [table](#) or [view](#) for this purpose.

### Syntax pattern

```
data_type ::=  
    <builtin_data_type>  
    | <domain_name>  
    | TYPE OF <domain_name>  
    | TYPE OF COLUMN <table or view>.<column>
```

*Note:* `TYPE OF COLUMN` gets only the type of the column. Any [constraints](#) or default values defined for the column are ignored.

## Examples

```
CREATE TABLE PERSON (  
    ID INTEGER,  
    NAME VARCHAR(40)  
);  
  
CREATE PROCEDURE SP_INS_PERSON (  
    ID TYPE OF COLUMN PERSON.ID,  
    NAME TYPE OF COLUMN PERSON.NAME  
)  
AS  
DECLARE VARIABLE NEW_ID TYPE OF COLUMN PERSON.ID;  
BEGIN  
    INSERT INTO PERSON (ID, NAME)  
        VALUES (:ID, :NAME)  
        RETURNING ID INTO :NEW_ID;  
END
```

## Hidden trap!

In v.2.5 and beyond, it is possible to alter the data type of a column, even if the column is referenced in a stored procedure or trigger, without an exception being thrown. Because compiled PSQL is stored statically as a binary representation (“BLR”) in a BLOB, the original BLR survives even a backup and restore. Being static, the BLR is not updated by the data type change, either.

This means that, for variables declared using the `TYPE OF` syntax, as well as the affected columns from the tables, together with any view columns derived from them, the compiled BLR is broken by the change of data type. At best, the BLR will be flagged as “needing attention” but tests show that the flag is not set under all conditions.

In short, the engine now no longer stops you from changing the type of a field that has any dependencies in compiled PSQL. It will be a matter for your own change control to identify the affected procedures and triggers and recompile them to accommodate the changes.

[back to top of page](#)

## New extensions to EXECUTE STATEMENT

Unusually for our release notes, we begin this chapter with the full, newly extended syntax for the `EXECUTE STATEMENT` statement in PSQL and move on afterwards to explain the various new features and their usage.

```
[FOR] EXECUTE STATEMENT <query_text> [( <input_parameters> )]  
    [ON EXTERNAL [DATA SOURCE] <connection_string>]  
    [WITH {AUTONOMOUS | COMMON} TRANSACTION]  
    [AS USER <user_name>]  
    [PASSWORD <password>]  
    [ROLE <role_name>]  
    [WITH CALLER PRIVILEGES]
```

[INTO <variables>]

*Note:* The order of the optional clauses is not fixed so, for example, a statement based on the following model would be just as valid:

```
[ON EXTERNAL [DATA SOURCE] <connection_string>]
[WITH {AUTONOMOUS | COMMON} TRANSACTION]
[AS USER <user_name>]
[PASSWORD <password>]
[ROLE <role_name>]
[WITH CALLER PRIVILEGES]
```

Clauses cannot be duplicated.

## Context issues

If there is no `ON EXTERNAL DATA SOURCE` clause present, `EXECUTE STATEMENT` is normally executed within the `CURRENT_CONNECTION` context. This will be the case if the `AS USER` clause is omitted, or it is present with its `<user_name>` argument equal to `CURRENT_USER`.

However, if `<user_name>` is not equal to `CURRENT_USER`, then the statement is executed in a separate connection, established without Y-Valve and remote layers, inside the same engine instance.

*Note:* In the absence of an `AS USER <user_name>` clause, `CURRENT_USER` is the default.

## Authentication

Where server authentication is needed for a connection that is different to `CURRENT_CONNECTION`, e.g., for executing an `EXECUTE STATEMENT` command on an external datasource, the `AS USER` and `PASSWORD` clauses are required. However, under some conditions, the `PASSWORD` may be omitted and the effects will be as follows:

1. On Windows, for the `CURRENT_CONNECTION` (i.e., no external data source), trusted authentication will be performed if it is active and the `AS USER` parameter is missing, null or equal to `CURRENT_USER`.
2. If the external data source parameter is present and its `<connection_string>` refers to the same database as the `CURRENT_CONNECTION`, the effective user account will be that of the `CURRENT_USER`.
3. If the external data source parameter is present and its `<connection_string>` refers to a different database than the one `CURRENT_CONNECTION` is attached to, the effective user account will be the operating system account under which the Firebird process is currently running.

In any other case where the `PASSWORD` clause is missing, only `isc_dpb_user_name` will be presented in the DPB (attachment parameters) and native authentication will be attempted.

## Transaction behaviour

The new syntax has an optional clause for setting the appropriate transaction behaviour: `WITH AUTONOMOUS TRANSACTION` and `WITH COMMON TRANSACTION`. `WITH COMMON TRANSACTION` is the default and does not need to be specified. Transaction lifetimes are bound to the lifetime of `CURRENT_TRANSACTION` and are committed or rolled back in accordance with the `CURRENT_TRANSACTION`.

The behaviour for `WITH COMMON TRANSACTION` is as follows:

- a. Causes any transaction in an external data source to be started with the same parameters as `CURRENT_TRANSACTION`; otherwise
- b. Executes the statement inside the `CURRENT_TRANSACTION`; or
- c. May use another transaction that is started internally in `CURRENT_CONNECTION`.

The `WITH AUTONOMOUS TRANSACTION` setting starts a new transaction with the same parameters as `CURRENT_TRANSACTION`. That transaction will be committed if the statement is executed without exceptions or rolled back if the statement encounters an error.

## Inherited access privileges

Vladyslav Khorsun

Tracker reference [CORE-1928](#).

By design, the original implementation of `EXECUTE STATEMENT` isolated the executable code from the access privileges of the calling [stored procedure](#) or [trigger](#), falling back to the privileges available to the `CURRENT_USER`. In general, the strategy is wise, since it reduces the vulnerability inherent in providing for the execution of arbitrary statements. However, in hardened environments, or where privacy is not an issue, it could present a limitation.

The introduction of the optional clause `WITH CALLER PRIVILEGES` now makes it possible to have the executable statement inherit the access privileges of the calling stored procedure or trigger. The statement is prepared using any additional privileges that apply to the calling stored procedure or trigger. The effect is the same as if the statement were executed by the stored procedure or trigger directly.

*Important:* The `WITH CALLER PRIVILEGES` option is not compatible with the `ON EXTERNAL DATA SOURCE` option.

[back to top of page](#)

## External queries from PSQL

Vladyslav Khorsun

Tracker reference [CORE-1853](#).

`EXECUTE STATEMENT` now supports queries against external databases by inclusion of the `ON EXTERNAL DATA SOURCE` clause with its `<connection_string>` argument.

## The `<connection_string>` argument

The format of `<connection_string>` is the usual one that is passed through the [API](#) function `isc_attach_database()`, viz.

```
[<host_name><protocol_delimiter>]database_path
```

## Character set

The connection to the external data source uses the same [character set](#) as is being used by the `CURRENT_CONNECTION` context.

## Access privileges

If the external data source is on another server then the clauses `AS USER <user_name>` and `PASSWORD <password>` will be needed.

The clause `WITH CALLER PRIVILEGES` is a no-op if the external data source is on another server.

## MORE INFORMATION REQUIRED. ROLES?

*Note:* Use of a two-phase transaction for the external connection is not available in v.2.5.

## EXECUTE STATEMENT with dynamic parameters

Vladyslav Khorsun Alex Peshkov

Tracker reference [CORE-1221](#).

The new extensions provide the ability to prepare a statement with dynamic input parameters (placeholders) in a manner similar to a parameterised [DSQL](#) statement. The actual text of the query itself can also be passed as a parameter.

## Syntax conventions

The mechanism employs some conventions to facilitate the run-time parsing and to allow the option of “naming” parameters in a style comparable with the way some popular client wrapper layers, such as Delphi, handle DSQL parameters. The API's own convention, of passing unnamed parameters in a predefined order, is also supported.

**However, named and unnamed parameters cannot be mixed.**

## The new binding operator

At this point in the implementation of the dynamic parameter feature, to avoid clashes with equivalence tests, it was necessary to introduce a new assignment operator for binding run-time values to named parameters. The new operator mimics the Pascal assignment operator: `:=`.

### Syntax for defining parameters

```
<input_parameters> ::=
    <named_parameter> | <input_parameters>, <named_parameter>
```

```
<named_parameter> ::=
```

```
<parameter name> := <expression>
```

### Example for named input parameters

For example, the following block of PSQL defines both `<query_text>` and named `<input_parameters>` (`<named_parameter>`):

```
EXECUTE BLOCK AS
  DECLARE S VARCHAR(255);
  DECLARE N INT = 100000;
  BEGIN
    /* Normal PSQL string assignment of <query_text> */
    S = 'INSERT INTO TTT VALUES (:a, :b, :a)';

    WHILE (N > 0) DO
      BEGIN
        /* Each loop execution applies both the string value
           and the values to be bound to the input parameters */

        EXECUTE STATEMENT (:S) (a := CURRENT_TRANSACTION, b :=
CURRENT_CONNECTION)
        WITH COMMON TRANSACTION;
        N = N - 1;
      END
    END
  END
```

### Example for unnamed input parameters

A similar block using a set of unnamed input parameters instead and passing constant arguments directly:

```
EXECUTE BLOCK AS
  DECLARE S VARCHAR(255);
  DECLARE N INT = 100000;
  BEGIN
    S = 'INSERT INTO TTT VALUES (?, ?, ?)';
```

```
WHILE (N > 0) DO
BEGIN
    EXECUTE STATEMENT (:S) (CURRENT_TRANSACTION, CURRENT_CONNECTION,
CURRENT_TRANSACTION);
    N = N - 1;
END
END
```

Note: Observe that, if you use both `<query_text>` and `<input_parameters>` then the `<query_text>` must be enclosed in parentheses, viz.

```
EXECUTE STATEMENT (:sql) (p1 := 'abc', p2 := :second_param) ...
```

[back to top of page](#)

## Exception handling

The handling of [exceptions](#) depends on whether the ON EXTERNAL DATA SOURCE is present.

### ON EXTERNAL DATA SOURCE clause is present

If ON EXTERNAL DATA SOURCE clause is present, Firebird cannot interpret error codes supplied by the unknown data source so it interprets the error information itself and wraps it as a string into its own error wrapper (`isc_eds_connection` or `isc_eds_statement`).

The text of the interpreted remote error contains both error codes and corresponding messages.

#### 1. Format of `isc_eds_connection` error

Template string

Execute statement error at @1 :\n@2Data source : @3

Status-vector tags

`isc_eds_connection`,  
`isc_arg_string`, <failed API function name>,  
`isc_arg_string`, <text of interpreted external error>,  
`isc_arg_string`, <data source name>

#### 2. Format of `isc_eds_statement` error

Template string

Execute statement error at @1 :\n@2Statement : @3\nData source : @4

Status-vector tags

`isc_eds_statement`,  
`isc_arg_string`, <failed API function name>,  
`isc_arg_string`, <text of interpreted external error>,  
`isc_arg_string`, <query>,



```
isc_arg_string, <data source name>
```

At PSQL level the symbols for these errors can be handled by treating them like any other [gdscode](#). For example

```
WHEN GDSCODE isc_edc_statement
```

*Note:* Currently, the originating error codes are not accessible in a `WHEN` statement. The situation could be improved in future.

### **ON EXTERNAL DATA SOURCE clause is not present**

If `ON EXTERNAL DATA SOURCE` clause is not present, the original status-vector with the error is passed as-is to the caller PSQL code.

For example, if a dynamic statement were to raise the `isc_lock_conflict` exception, the exception would be passed to the caller and could be handled using the usual handler:

```
WHEN GDSCODE isc_lock_conflict
```

[back to top of page](#)

### **Examples using EXECUTE STATEMENT**

The following examples offer a sampler of ways that the `EXECUTE STATEMENT` extensions might be applied in your applications.

### **Test connections and transactions**

A couple of tests you can try to compare variations in settings:

Test a) Execute this block few times in the same transaction - it will create three new connections to the current database and reuse it in every call. Transactions are also reused.

```
EXECUTE BLOCK
  RETURNS (CONN INT, TRAN INT, DB VARCHAR(255))
AS
  DECLARE I INT = 0;
  DECLARE N INT = 3;
  DECLARE S VARCHAR(255);
BEGIN
  SELECT A.MON$ATTACHMENT_NAME FROM MON$ATTACHMENTS A
    WHERE A.MON$ATTACHMENT_ID = CURRENT_CONNECTION
    INTO :S;
  WHILE (i < N) DO
  BEGIN
    DB = TRIM(CASE i - 3 * (I / 3)
```

```
WHEN 0 THEN '\\.\' WHEN 1 THEN 'localhost:' ELSE '' END) || :S;

FOR EXECUTE STATEMENT
  'SELECT CURRENT_CONNECTION, CURRENT_TRANSACTION
  FROM RDB$DATABASE'
  ON EXTERNAL :DB
  AS USER CURRENT_USER PASSWORD 'masterkey' -- just for example
  WITH COMMON TRANSACTION
  INTO :CONN, :TRAN
DO SUSPEND;

  i = i + 1;
END
END
```

Test b) : Execute this block few times in the same transaction - it will create three new connections to the current database on every call.

```
EXECUTE BLOCK
  RETURNS (CONN INT, TRAN INT, DB VARCHAR(255))
AS
  DECLARE I INT = 0;
  DECLARE N INT = 3;
  DECLARE S VARCHAR(255);
BEGIN
  SELECT A.MON$ATTACHMENT_NAME
  FROM MON$ATTACHMENTS A
  WHERE A.MON$ATTACHMENT_ID = CURRENT_CONNECTION
  INTO :S;

  WHILE (i < N) DO
  BEGIN
    DB = TRIM(CASE i - 3 * (I / 3)
      WHEN 0 THEN '\\.\'
      WHEN 1 THEN 'localhost:'
      ELSE '' END) || :S;

    FOR EXECUTE STATEMENT
      'SELECT CURRENT_CONNECTION, CURRENT_TRANSACTION FROM RDB$DATABASE'
      ON EXTERNAL :DB
      WITH AUTONOMOUS TRANSACTION -- note autonomous transaction
      INTO :CONN, :TRAN
    DO SUSPEND;

    i = i + 1;
  END
END
```

## Input evaluation demo

Demonstrating that input expressions evaluated only once:

```
EXECUTE BLOCK
  RETURNS (A INT, B INT, C INT)
AS
BEGIN
  EXECUTE STATEMENT (
    'SELECT CAST(:X AS INT),
           CAST(:X AS INT),
           CAST(:X AS INT)
    FROM RDB$DATABASE' )
    (x := GEN_ID(G, 1))
    INTO :A, :B, :C;
  SUSPEND;
END
```

## Insert speed test

Recycling our earlier examples for input parameter usage for comparison with the non-parameterised form of EXECUTE STATEMENT:

```
RECREATE TABLE TTT (
  TRAN INT,
  CONN INT,
  ID INT);

-- Direct inserts:

EXECUTE BLOCK AS
  DECLARE N INT = 100000;
BEGIN
  WHILE (N > 0) DO
  BEGIN
    INSERT INTO TTT VALUES (CURRENT_TRANSACTION, CURRENT_CONNECTION,
    CURRENT_TRANSACTION);
    N = N - 1;
  END
END

-- Inserts via prepared dynamic statement
-- using named input parameters:

EXECUTE BLOCK AS
  DECLARE S VARCHAR(255);
  DECLARE N INT = 100000;
BEGIN
  S = 'INSERT INTO TTT VALUES (:a, :b, :a)';
```

```
WHILE (N > 0) DO
BEGIN
    EXECUTE STATEMENT (:S)
        (a := CURRENT_TRANSACTION, b := CURRENT_CONNECTION)
    WITH COMMON TRANSACTION;
    N = N - 1;
END
END

-- Inserts via prepared dynamic statement
-- using unnamed input parameters:

EXECUTE BLOCK AS
DECLARE S VARCHAR(255);
DECLARE N INT = 100000;
BEGIN
    S = 'INSERT INTO TTT VALUES (?, ?, ?)';

    WHILE (N > 0) DO
    BEGIN
        EXECUTE STATEMENT (:S) (CURRENT_TRANSACTION, CURRENT_CONNECTION,
CURRENT_TRANSACTION);
        N = N - 1;
    END
END
```

[back to top of page](#)

## Other PSQL improvements

Improvements made to existing PSQL syntax include the following:

### Subqueries as PSQL expressions

A. dos Santos Fernandes

Tracker reference [CORE-2580](#).

Previously, a subquery used as a PSQL expression would return an [exception](#), even though it was logically valid in SQL terms. For example, the following constructions would all return errors:

```
var = (select ... from ...);
if ((select ... from ...) = 1) then
if (1 = any (select ... from ...)) then
if (1 in (select ... from ...)) then
```

Now, such potentially valid expressions are allowed, removing the need to jump through hoops to fetch the output of a scalar subquery into an intermediate variable using [SELECT...INTO](#).

## SQLSTATE as a context variable

D. Yemanov

Tracker reference [CORE-2890](#).

(v.2.5.1) `SQLSTATE` is made available as a PSQL context variable, to be used with `WHEN` in an exception block, like `GDSCODE` and `SQLCODE`.

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

Permanent link:  
<http://ibexpert.com/docu/doku.php?id=01-documentation:01-08-firebird-documentation:firebird-2.5.3-release-notes:procedural-sql>

Last update: **2023/06/25 23:36**

