Stored Procedure Language (PSQL)

PSQL enhancements

The following enhancements have been made to the PSQL language extensions for stored procedures and triggers:

Context variable ROW_COUNT enhanced

D. Yemanov

ROW_COUNT has been enhanced so that it can now return the number of rows returned by a SELECT statement.

For example, it can be used to check whether a singleton SELECT INTO statement has performed an assignment:

```
..
BEGIN
   SELECT COL FROM TAB INTO :VAR;

   IF (ROW_COUNT = 0) THEN
      EXCEPTION NO_DATA_FOUND;
END
..
```

See also its usage in the examples below for explicit PSQL cursors.

back to top of page

Explicit cursors

D. Yemanov

It is now possible to declare and use multiple cursors in PSQL. Explicit cursors are available in a DSQL EXECUTE BLOCK structure as well as in stored procedures and triggers.

Syntax pattern

```
DECLARE [VARIABLE] <cursor_name> CURSOR FOR ( <select_statement> );
OPEN <cursor_name>;
FETCH <cursor_name> INTO <var_name> [, <var_name> ...];
CLOSE <cursor_name>;
```

Examples

1.

```
DECLARE RNAME CHAR(31);
DECLARE C CURSOR FOR ( SELECT RDB$RELATION_NAME FROM RDB$RELATIONS );

BEGIN
   OPEN C;
   WHILE (1 = 1) DO
   BEGIN
      FETCH C INTO :RNAME;
      IF (ROW_COUNT = 0) THEN
         LEAVE;
      SUSPEND;
   END
   CLOSE C;
END
```

2.

```
DECLARE RNAME CHAR(31);
DECLARE FNAME CHAR(31);
DECLARE C CURSOR FOR ( SELECT RDB$FIELD_NAME
                       FROM RDB$RELATION_FIELDS
                       WHERE RDB$RELATION_NAME = :RNAME
                       ORDER BY RDB$FIELD_POSITION );
BEGIN
   FOR
      SELECT RDB$RELATION_NAME
      FROM RDB$RELATIONS
      INTO :RNAME
   DO
   BEGIN
      OPEN C;
      FETCH C INTO :FNAME;
      CLOSE C;
   SUSPEND;
   END
END
```

Note:

- Cursor declaration is allowed only in the declaration section of a PSQL block/procedure/trigger, as with any regular local variable declaration.
- Cursor names are required to be unique in the given context. They must not conflict with the name of another cursor that is “announced”, via the AS CURSOR clause, by a FOR SELECT cursor. However, a cursor can share its name with any other type of variable within the same context, since the operations available to each are different.
- Positioned updates and deletes with cursors using the WHERE CURRENT OF clause are allowed.
- Attempts to fetch from or close a FOR SELECT cursor are prohibited.
- Attempts to open a cursor that is already open, or to fetch from or close a cursor that is already

closed, will fail.
- All cursors which were not explicitly closed will be closed automatically on exit from the current PSQL block/procedure/trigger.
- The ROW_COUNT system variable can be used after each FETCH statement to check whether any row was returned.

back to top of page

Defaults for stored procedure arguments

V. Horsun

Defaults can now be declared for stored procedure arguments.

The syntax is the same as a default value definition for a column or domain, except that you can use '=' in place of the 'DEFAULT' keyword.

Arguments with default values must be last in the argument list; that is, you cannot declare an argument that has no default value after any arguments that have been declared with default values. The caller must supply the values for all of the arguments preceding any that are to use their defaults.

For example, it is illegal to do something like this: supply arg1, arg2, miss arg3, set arg4…

Substitution of default values occurs at run-time. If you define a procedure with defaults (say P1), call it from another procedure (say P2) and skip some final, defaulted arguments, then the default values for P1 will be substituted by the engine at time execution P1 starts. This means that, if you change the default values for P1, it is not necessary to recompile P2.

However, it is still necessary to disconnect all client connections, as discussed in the Borland InterBase® 6 beta Data Definition Guide (DataDef.pdf), in the section *Altering and dropping procedures in use*.

Examples

```
CONNECT ... ;
SET TERM ^;
CREATE PROCEDURE P1 (X INTEGER = 123)
RETURNS (Y INTEGER)
AS
BEGIN
   Y = X;
   SUSPEND;
END ^
COMMIT ^
SET TERM ;^

SELECT * FROM P1;

           Y
============
           123

EXECUTE PROCEDURE P1;

           Y
============
         123

SET TERM ^;
CREATE PROCEDURE P2
RETURNS (Y INTEGER)
AS
BEGIN
   FOR SELECT Y FROM P1 INTO :Y
   DO SUSPEND;
END ^
COMMIT ^
SET TERM ;^

SELECT * FROM P2;

           Y
============
         123

SET TERM ^;
ALTER PROCEDURE P1 (X INTEGER = CURRENT_TRANSACTION)
RETURNS (Y INTEGER)
AS
BEGIN
   Y = X;
   SUSPEND;
END; ^
COMMIT ^
SET TERM ;^

SELECT * FROM P1;

           Y
============
        5875

SELECT * FROM P2;

           Y
============
         123

COMMIT;
```

```
CONNECT ... ;

SELECT * FROM P2;

           Y
============
        5880
```

Note: The source and BLR for the argument defaults are stored in RDB$FIELDS.

back to top of page

LEAVE <label> syntax support

D. Yemanov

New LEAVE <label> syntax now allows PSQL loops to be marked with labels and terminated in Java style. 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.

Syntax pattern

```
<label_name>: <loop_statement>
...
LEAVE [<label_name>]
```

where <loop_statement> is one of: WHILE, FOR SELECT, FOR EXECUTE STATEMENT.

Examples

1.

```
FOR
   SELECT COALESCE(RDB$SYSTEM_FLAG, 0), RDB$RELATION_NAME
      FROM RDB$RELATIONS
      ORDER BY 1
   INTO :RTYPE, :RNAME
   DO
   BEGIN
      IF (RTYPE = 0) THEN
         SUSPEND;
      ELSE
         LEAVE; -- exits current loop
   END
```

2.

```
CNT = 100;
L1:
WHILE (CNT >= 0) DO
```

```
BEGIN
   IF (CNT < 50) THEN
       LEAVE L1; -- exists WHILE loop
   CNT = CNT - l;
END
```

3.

STMT1 = 'SELECT RDB$RELATION_NAME FROM RDB$RELATIONS';

```
L1:
FOR
   EXECUTE STATEMENT :STMT1 INTO :RNAME
DO
BEGIN
    STMT2 = 'SELECT RDB$FIELD_NAME FROM RDB$RELATION_FIELDS
      WHERE RDB$RELATION_NAME = ';
   L2:
   FOR
      EXECUTE STATEMENT :STMT2 || :RNAME INTO :FNAME
   DO
   BEGIN
      IF (RNAME = 'RDB$DATABASE') THEN
         LEAVE L1; -- exits the outer loop
      ELSE IF (RNAME = 'RDB$RELATIONS') THEN
         LEAVE L2; -- exits the inner loop
      ELSE
        SUSPEND;
   END
END
```

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

back to top of page

OLD context variables now read-only

D. Yemanov

The set of OLD context variables available in trigger modules is now read-only. An attempt to assign a value to OLD.something will be rejected.

Note: NEW context variables are now read-only in AFTER-triggers as well.

back to top of page

PSQL stack trace

V. Horsun

The API client can now extract a simple stack trace Error Status Vector when an exception occurs during PSQL execution (stored procedures or triggers). A stack trace is represented by one string (2048 bytes max.) and consists of all the stored procedure and trigger names, starting from the point where the exception occurred, out to the outermost caller. If the actual trace is longer than 2Kb, it is truncated.

Additional items are appended to the status vector as follows:

```
isc_stack_trace, isc_arg_string, <string length>, <string>
```

isc_stack_trace is a new error code with value of 335544842L.

Examples

Metadata creation

```
CREATE TABLE ERR (
   ID INT NOT NULL PRIMARY KEY,
   NAME VARCHAR(16));

CREATE EXCEPTION EX '!';
SET TERM ^;

CREATE OR ALTER PROCEDURE ERR_1 AS
BEGIN
   EXCEPTION EX 'ID = 3';
END ^

CREATE OR ALTER TRIGGER ERR_BI FOR ERR
   BEFORE INSERT AS
BEGIN
   IF (NEW.ID = 2)
   THEN EXCEPTION EX 'ID = 2';

   IF (NEW.ID = 3)
   THEN EXECUTE PROCEDURE ERR_1;

   IF (NEW.ID = 4)
   THEN NEW.ID = 1 / 0;
END ^

CREATE OR ALTER PROCEDURE ERR_2 AS
BEGIN
   INSERT INTO ERR VALUES (3, '333');
END ^
```

1. User exception from a trigger:

```
SQL" INSERT INTO ERR VALUES (2, '2');
```

```
Statement failed, SQLCODE = -836
exception 3
-ID = 2
-At trigger 'ERR_BI'
```

2. User exception from a procedure called by a trigger:

```
SQL" INSERT INTO ERR VALUES (3, '3');
Statement failed, SQLCODE = -836
exception 3
-ID = 3
-At procedure 'ERR_1'
At trigger 'ERR_BI'
```

3. Run-time exception occurring in trigger (division by zero):

```
SQL" INSERT INTO ERR VALUES (4, '4');
Statement failed, SQLCODE = -802
arithmetic exception, numeric overflow, or string truncation
-At trigger 'ERR_BI'
```

4. User exception from procedure:

```
SQL" EXECUTE PROCEDURE ERR_1;
Statement failed, SQLCODE = -836
exception 3
-ID = 3
-At procedure 'ERR_1'
```

5. User exception from a procedure with a deeper call stack:

```
SQL" EXECUTE PROCEDURE ERR_2;
Statement failed, SQLCODE = -836
exception 3
-ID = 3
-At procedure 'ERR_1'
At trigger 'ERR_BI'
At procedure 'ERR_2'
```

back to top of page

Call a UDF as a void function (procedure)

N. Samofatov

In PSQL, supported UDFs, e.g. RDB$SET_CONTEXT, can be called as though they were void functions (a.k.a “procedures” in Object Pascal). For example:

```
BEGIN
```

```
...
RDB$SET_CONTEXT('USER_TRANSACTION', 'MY_VAR', '123');
...
END
```