PL/pgSQL plans for v7.2

Stored procedures

Result sets

Parameterized views

Jan Wieck

May 2001

Introduction

Since the creation of **PL/pgSQL**, a procedural SQL dialect for implementing Functions and Trigger procedures in **PostgreSQL**, the capabilities have been improved only slightly. For version v7.2 one of the every so often asked database features will be addressed. This is multiple rows resulting from a procedure call.

Step by step

The implementation will be based on portals, the internal name of the functionality that implements SQL cursors. Some new functions in the Server Programming Interface (SPI) enable access to portals. This new SPI capability will be used by the **PL/pgSQL** handler to limit the memory usage when dealing with huge amounts of data and for the implementation of cursor and reference cursor variables.

- Portal access through SPI

  The SPI support for portal access is already available as a patch and could be applied to the current v7.2 development tree.

  - Portal SPI_cursor_open(char *name, void *plan, Datum *values, char *nulls)

    name is a string identifying the portal. If not given (NULL), the function will make up a generic name "<unnamed cursor *n*>", where *n* is a counter inside of the backend.

    plan is a prepared or saved SPI execution plan that must consist of one single SELECT query, as resulting from the existing SPI_prepare() and SPI_saveplan() functions.

    values and **nulls** specify the actual call parameters for the execution of the plan identical with the arguments for the existing SPI_execp() function.

    The return value is a Portal, used in subsequent calls to other functions. It is a complete portal, holding an already started executor for the given plan ready to return tuples.

  - Portal SPI_cursor_find(char *name)

    name is a string identifying an existing portal. With this function a procedure can gain access to a portal created by another procedure or even the main application using the regular DECLARE name CURSOR utility command.

- void SPI_cursor_fetch(Portal portal, bool forward, int count)

  portal is the open portal obtained from a previous call to SPI_cursor_open() or SPI_cursor_find().

  forward is the direction in which to fetch tuples. If true, the direction is FORWARD.

  count is the maximum number of tuples to fetch during this call. The value "0" means ALL.

  The resulting tuples will be placed into the usual SPI_tuptable structure in the procedures memory context and SPI_processed set accordingly. So the entire set of SPI_ support functions for access to the attributes is available.

- void SPI_cursor_move(Portal portal, bool forward, int count)

  Same as SPI_cursor_fetch(), except that the tuples aren't stuffed into the SPI_tuptable. SPI_processed is set to the real number of tuples skipped by the operation.

- void SPI_cursor_close(Portal portal)

  Closes the portal specified by **portal**.

- void SPI_freetuptable(SPITypleTable *tuptable)

  Frees all memory allocated for the SPI result identified by **tuptable**. SPI tuple tables now have their own memory context, which is a child of the procedure memory context. This should be called when a procedure is done with the results of a previous SPI_exec(), SPI_execp() or SPI_cursor_fetch() operation.

- int SPI_freeplan(void *plan)

  Likewise prepared SPI execution plans now have their own memory contexts, which is either a child of the procedures context (plan from SPI_prepare()) or the TopMemoryContext (plan from SPI_saveplan()).

  This function isn't part of the cursor implementation, but for better support of dynamic SQL.

- Implicit cursors inside **PL/pgSQL**

  The following functionality is already available as a patch for the current v7.2 development tree.

  The language constructs "FOR ... IN {SELECT | EXECUTE} ... LOOP" now use a portal internally. After creating the portal, the first 10 rows are fetched at maximum. When the procedure continues with the loop after these have been processed, the remaining rows are fetched in chunks of 50 at a time. When leaving the loop (either with an explicit EXIT statement or because of running out of result tuples), the portal is closed.

  The **PL/pgSQL** handler makes use of the new SPI_freeplan() function so usage of EXECUTE does not sum up memory consumption until procedure return any more.

- Reference cursors

  A new datatype *refcursor*, holding a string like *text* is available. This datatype is handled specially by **PL/pgSQL** as follows.

  - Declaration

```
DECLARE
    curvar REFCURSOR [:= <string_expr>];
```

  Declares a local variable that can hold a portal. *refcursor* is basically a string datatype, so it really holds the portal name only. All subsequent access to the portal is done through SPI_cursor_open() or SPI_cursor_find() and following access to the real portal. Since the portal lookup is internally through a hashtable, there is no big performance penalty for that.

- Usage in the body

  A refcursor variable can be used inside the body for OPEN, FETCH and CLOSE commands.

```
BEGIN
  curvar := <string_expr>;

    OPEN curvar FOR <select_stmt>;

    OPEN curvar FOR EXECUTE <string_expr>;

    FETCH curvar INTO {<record> | <row> | <var> [, ...]};

    CLOSE curvar;
```

By default, a *refcursor* variable holds a NULL value. If not assigned an explicit string via either a default value or explicit assignment, this causes the OPEN statement to call SPI_cursor_open() with a NULL name. The generic name used for the portal will then be placed into the variable, so the procedure knows the name and could pass around the variable as described below. If a value is assigned to the *refcursor* variable, that will be used in the call to SPI_cursor_open().

The FETCH operation causes the next row in forward direction placed into the target, or NULL values if there is no next row. The global variable FOUND flags wether another row was found or not, so a typical loop will look like

```
BEGIN
    OPEN curvar FOR SELECT * FROM mytable;
    LOOP
        FETCH curvar INTO mytable_row;
        EXIT WHEN NOT FOUND;
        -- Processing of one row
    END LOOP;
    CLOSE curvar;
```

Portal operations in **PostgreSQL** must be done inside of transaction blocks. Since all statements inside a function are automatically covered by at least the implicit transaction of the statement that invoked the function, there is no need for an explicit transaction block in the above example.

Because *refcursor* is a regular **PostgreSQL** datatype, it can be used in function declarations for arguments and return values as well as the following example demonstrates (a little overcomplicated version of counting the number of rows in the sample_table).

```
CREATE FUNCTION sample_main() RETURNS integer
AS '
DECLARE
    curvar refcursor;
    retval integer;
BEGIN
    curvar := sample_open();
    retval := sample_process(curvar);
    CLOSE curvar();
    RETURN retval;
END;'
LANGUAGE 'plpgsql';

CREATE FUNCTION sample_open() RETURNS refcursor
AS '
DECLARE
    curvar refcursor;
```

```
BEGIN
    OPEN curvar FOR SELECT * FROM sample_table;
    RETURN curvar;
END;'
LANGUAGE 'plpgsql';

CREATE FUNCTION sample_process(refcursor) RETURNS integer
AS '
DECLARE
    retval integer := 0;
    s_row  sample_table%ROWTYPE;
BEGIN
    LOOP
        FETCH $1 INTO s_row;
        EXIT WHEN NOT FOUND;

        retval := retval + 1;
    END LOOP;
    RETURN retval;
END;'
LANGUAGE 'plpgsql';
```

Again, there is no need for an explicit transaction block here, because all the access to the portal happens inside of one transaction either way.

- Passing refcursor between application and **PL/pgSQL**

The next advantage using a simple string datatype for passing around portal names is, that interaction between functions and the main application is possible. A function receiving a reference cursor can in fact receive a cursor provided by the application through this. Or a function opening a cursor with a generic name could pass it back to the application. Of course, since all portals are automatically closed at transaction end, this requires explicit transaction blocks to work. The above function sample_process() could be used by the application like:

```
BEGIN TRANSACTION;
DECLARE c1 CURSOR FOR SELECT * FROM sample_table;
SELECT sample_process('c1');
CLOSE c1;
COMMIT;
```

A simple string doesn't mean, that a portal with that name has to exist at all when passing the value around. So the application can call a function that opens a cursor for it and tell the wanted cursor name as in

```
CREATE FUNCTION sample_open(refcursor, integer) RETURNS refcursor
AS '
DECLARE
```

```
        curvar ALIAS FOR $1;
        maxid  ALIAS FOR $2;
    BEGIN
        OPEN curvar FOR SELECT * FROM sample_table
                        WHERE id <= maxid;
        RETURN curvar;
    END;'
    LANGUAGE 'plpgsql';

    BEGIN TRANSACTION;
    SELECT sample_open('c1', 55);
    FETCH ALL IN c1;
    CLOSE c1;
    COMMIT;
```

Here, passing in a NULL value for the refcursor argument would again cause the OPEN statement to make up a generic name for the cursor. So the application could use the returned "<unnamed cursor *n*>" portal name for the FETCH and CLOSE statements on the SQL level.

- Explicit cursors in **PL/pgSQL**

Explicit cursors have their SELECT statement defined in the declaration section of a procedure, rather then specified at OPEN time. The syntax is

```
    DECLARE
        curvar CURSOR [(argname <type> [, ...])]
               IS <select_stmt>;

    BEGIN
        OPEN curvar [(<expr> [, ...])];
```

Internally they are implemented as *refcursor* variables too, but with additional knowledge of the **PL/pgSQL** handler inside of the procedure that defines them. In contrast to *refcursor* variables, they are automatically initialized to a value identical to their name. The above example would in fact open a portal with the name "curvar", which must not exist already or a 'cursor already in use' error is raised.

Only the procedure where the explicit cursor is declared can use the above OPEN syntax, because outside of the function the specified query is unknown.

Since the internal datatype again is *refcursor*, explicit cursors can as well be passed around as arguments to other functions or returned as function result to the caller or even the main application. Except for the way they are openend, there is no difference.

If the optional argument list is given in the cursor declaration, the open must have a matching expression list returning the defined data types. In addition to these arguments, every so far declared variable or call parameters to the function can be used in the explicit cursors SELECT statements. These variables will get evaluated at OPEN time, so modifications will be handled appropriately if the procedure opens and closes the explicit cursor multiple times during one invocation.

Like for reference cursor variables, a different "value" can be assigned to an explicit cursor, changing the name of the portal created at OPEN time. Surprise, surprise, assigning NULL will - you guess what.

Next steps

The above functionality is the base for a couple of possible future enhancements. Since the required work includes major changes in the database backend, this could only be done after in depth discussion with the entire developers team on the HACKERS list.

- Functions returning tuple sets

  In the current implementation, a function returning a SET is expected to return one HeapTuple per call and to detect by looking into the fcinfo structure if it's called the first or a subsequent time. This expected behavior requires a real state machine for functions internally, which is nearly impossible for most procedural languages (closing out PL/Perl, PL/python, PL/Tcl, and probably others). In addition, there is up to now no way to really use this feature from SQL, because the returned HeapTuple address is just output as an integer value.

  My idea is to change the expected return value from a function that is declared as returning a set to be a Portal. Then we extent the possible tuple sources inside of the portal to be either the existing, started executor, a sort tape or a function call returning a tuple at a time. So it's up to the creator of the portal where and how the set returned is exactly produced.

  Based on that functionality, a function returning a tuple set could appear in the rangetable (FROM clause) of a query. The new plan node holding the function is a functional scan, calling the function during plan initialization (ExecutorStart) to get the portal, using new portal interface functions for fetching tuples (not calling printtup but returning the HeapTuple instead) during ExecutorRun, and the portal is closed at ExecutorEnd.

- Implementation of RETURN ... AND RESUME

  The construct RETURN ... AND RESUME would normally require the language to be implemented as a real state machine. Only few languages are, most are call stack based and resuming at exactly the point where the last RETURN happened, including all restored local variables and states is usually imp\b\b\b a huge challenge :-).

Stuffing every result set into a temporary file before returning is insufficient for small sets. But collecting the result set in memory on the other hand is insufficient for dealing with huge result sets.

Extending the portal structure so it can hold a sort tape will help out here. The sort tapes in **PostgreSQL** are smart enough to hold a reasonable amount of data in memory and automatically rolling out to disc with temporary file(s) when the number of sort buffers is exceeded. All we need here is utility functions that open a portal in collect mode, stuff the tuples in there and switch it to return mode (internally rewinding the tape) so subsequent calls to portal fetch will return these tuples.