EPAM Systems, RD Dep., RD Dep.

POSTGRESQL DB FOR DWH AND ETL BUILDING

Advanced PL/pgSQL

CONTENTS

1. CURSORS IN PL/PGSQL

If you execute SQL, the database will calculate the result and send it to your application. Once the entire result set has been sent to the client, the application can continue doing its job. The problem is this: what happens if the result set is so large that it doesn't fit into the memory anymore? What if the database returns 10 billion rows? The client application usually cannot handle so much data at once, and actually, it shouldn't. The solution to this problem is a cursor. The idea behind a cursor is that data is generated only when it is needed (when FETCH is called). Therefore, the application can already start to consume data while it is actually being generated by the database. On top of that, much less memory is required to perform this operation.

When it comes to PL/pgSQL, cursors also play a major role. Whenever you loop over a result set, PostgreSQL will automatically use a cursor internally. The advantage is that the memory consumption of your applications will be reduced dramatically, and there is hardly a chance of ever running out of memory, due to the large amounts of data that are processed. There are various ways to use cursors.

1.1 DECLARING CURSOR VARIABLES

All access to cursors in PL/pgSQL goes through cursor variables, which are always of the special data type refcursor. One way to create a cursor variable is just to declare it as a variable of type refcursor. Another way is to use the cursor declaration syntax:

```
name [ [ NO ] SCROLL ] CURSOR [ ( arguments ) ] FOR query;
```

If SCROLL is specified, the cursor will be capable of scrolling backward; if NO SCROLL is specified, backward fetches will be rejected; if neither specification appears, it is query-dependent whether backward fetches will be allowed. arguments, if specified, is a comma-separated list of pairs name datatype that define names to be replaced by parameter values in the given query. The actual values to substitute for these names will be specified later, when the cursor is opened.

1.2 OPENING CURSORS

Before a cursor can be used to retrieve rows, it must be opened. (This is the equivalent action to the SQL command DECLARE CURSOR.) PL/pgSQL has three forms of the OPEN statement, two of which use unbound cursor variables while the third uses a bound cursor variable:

1.2.1 OPEN FOR query

The cursor variable is opened and given the specified query to execute. The cursor cannot be open already, and it must have been declared as an unbound cursor variable (that is, as a simple refcursor variable). The query must be a SELECT, or something else that returns rows (such as EXPLAIN). The query is treated in the same way as other SQL commands in PL/pgSQL: PL/pgSQL variable names are substituted, and the query plan is cached for possible reuse. When a PL/pgSQL variable is substituted into the cursor query, the value that is substituted is the one it has at the time of the OPEN; subsequent changes to the variable will not affect the cursor's behavior.

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR query;
```

1.2.2 OPEN FOR EXECUTE

The cursor variable is opened and given the specified query to execute. The cursor cannot be open already, and it must have been declared as an unbound cursor variable (that is, as a simple refcursor variable). The query is specified as a string expression, in the same way as in the EXECUTE command. As usual, this gives flexibility so the query plan can vary from one run to the next, and it also means that variable substitution is not done on the command string. As with EXECUTE, parameter values can be inserted into the dynamic command via format() and USING.

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR EXECUTE query_string
                                     [ USING expression [, ... ] ];
```

1.2.3 Opening a Bound Cursor

This form of OPEN is used to open a cursor variable whose query was bound to it when it was declared. The cursor cannot be open already. A list of actual argument value expressions must appear if and only if the cursor was declared to take arguments. These values will be substituted in the query. The query plan for a bound cursor is always considered cacheable; there is no equivalent of EXECUTE in this case. Argument values can be passed using either positional or named notation.

```
OPEN bound_cursorvar [ ( [ argument_name := ] argument_value [, ...] ) ];
```

1.3 USING CURSORS

Once a cursor has been opened, it can be manipulated with the following statements:

1.3.1 FETCH

FETCH retrieves the next row from the cursor into a target, which might be a row variable, a record variable, or a comma-separated list of simple variables, just like SELECT INTO. If there is no next row, the target is set to NULL(s). As with SELECT INTO, the special variable FOUND can be checked to see whether a row was obtained or not.

```
FETCH [ direction { FROM | IN } ] cursor INTO target;
```

The direction clause can be any of the variants allowed in the SQL FETCH command except the ones that can fetch more than one row; namely, it can be NEXT, PRIOR, FIRST, LAST, ABSOLUTE count, RELATIVE count, FORWARD, or BACKWARD. Omitting direction is the same as specifying NEXT. In the forms using a count, the count can be any integer-valued expression.

Direction values that require moving backward are likely to fail unless the cursor was declared or opened with the SCROLL option.

Cursor must be the name of a refcursor variable that references an open cursor portal.

1.3.2 MOVE

MOVE repositions a cursor without retrieving any data. MOVE works exactly like the FETCH command, except it only repositions the cursor and does not return the row moved to. As with SELECT INTO, the special variable FOUND can be checked to see whether there was a next row to move to.

```
MOVE [ direction { FROM | IN } ] cursor;
```

1.3.3 CURRENT OF

When a cursor is positioned on a table row, that row can be updated or deleted using the cursor to identify the row. There are restrictions on what the cursor's query can be and it's best to use FOR UPDATE in the cursor.

```
UPDATE table SET ... WHERE CURRENT OF cursor;
DELETE FROM table WHERE CURRENT OF cursor;
```

1.3.4 CLOSE

CLOSE closes the portal underlying an open cursor. This can be used to release resources earlier than end of transaction, or to free up the cursor variable to be opened again.

```
CLOSE cursor;
```

1.4 RETURNING CURSORS

PL/pgSQL functions can return cursors to the caller. This is useful to return multiple rows or columns, especially with very large result sets. To do this, the function opens the cursor and returns the cursor name to the caller or simply opens the cursor using a portal name specified by or otherwise known to the caller.

The caller can then fetch rows from the cursor. The cursor can be closed by the caller, or it will be closed automatically when the transaction closes.

The portal name used for a cursor can be specified by the programmer or automatically generated. To specify a portal name, simply assign a string to the refcursor variable before opening it. The string value of the refcursor variable will be used by OPEN as the name of the underlying portal. However, if the refcursor variable is null, OPEN automatically generates a name that does not conflict with any existing portal, and assigns it to the refcursor variable.

```
CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
    OPEN $1 FOR SELECT col FROM test;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc('funccursor');
FETCH ALL IN funccursor;
COMMIT;
```

1.4.1 LOOPING THROUGH A CURSOR'S RESULT

There is a variant of the FOR statement that allows iterating through the rows returned by a cursor.

```
[ <<label>> ]
FOR recordvar IN bound_cursorvar [ ( [ argument_name := ] argument_value [, ...] ) ] LOOP
    statements
END LOOP [ label ];
```

The cursor variable must have been bound to some query when it was declared, and it cannot be open already. The FOR statement automatically opens the cursor, and it closes the cursor again when the loop exits. A list of actual argument value expressions must appear if and only if the cursor was declared to take arguments. These values will be substituted in the query, in just the same way as during an OPEN.

The variable recordvar is automatically defined as type record and exists only inside the loop (any existing definition of the variable name is ignored within the loop). Each row returned by the cursor is successively assigned to this record variable and the loop body is executed.

2. COMPOSITE TYPES

In most other database systems, stored procedures are only used with primitive data types, such as integers, numeric, varchar, and so on. However, PostgreSQL is very different. We can use all the data types that are available to us. This includes primitive, composite, and custom types of data. There are simply no restrictions as far as data types are concerned. To unleash the full power of PostgreSQL, composite types are highly important and are often used by extensions that are found on the internet.

A composite type represents the structure of a row or record; it is essentially just a list of field names and their data types. PostgreSQL allows composite types to be used in many of the same ways that simple types can be used. For example, a column of a table can be declared to be of a composite type.

```
CREATE TYPE my_type AS (s text, t text);

CREATE FUNCTION f(my_type)
RETURNS my_type AS
$$
DECLARE
         v_row my_type;
BEGIN

         SELECT schemaname, tablename
         INTO v_row
         FROM pg_tables
         WHERE tablename = trim(my_type.t)
         AND schemaname = trim(my_type.s)
         LIMIT 1;

         RETURN v_row;

END;
$$ LANGUAGE 'plpgsql';
```

To access a field of a composite column, one writes a dot and the field name, much like selecting a field from a table name.

Returning the composite type is not difficult, either.

You just have to assemble the composite type variable on the fly and return it, just like any other data type. You can even easily use arrays, or even more complex structures.

```
SELECT (f).s, (f).t
FROM f('("public", "t_test")'::my_type);
```

3. DYNAMIC SQL

Dynamic SQL is any SQL statement that is first built as a text string and then executed using the EXECUTE command. Possibilities that are opened by using dynamic SQL are underused in most of RDBMSs, but even more so in PostgreSQL.

Consider the following points:

- First, in PostgreSQL, execution plans are not cached even for prepared queries (i.e., queries that are prepared, analyzed, and rewritten using the PREPARE command). That means that optimization always happens immediately before execution.
- Second, the optimization step in PostgreSQL happens later than in other systems. For example, in Oracle, the execution plan for a parameterized query is always prepared for a generic query, even if the specific values are there. Moreover, a plan with binding variables is cached for future usage if the same query with different values is executed. The optimizer takes the table and index statistics into account but does not take into account the specific values of parameters. PostgreSQL does the opposite. The execution plan is generated for specific values.

Often, if you suggest to a team of developers to use dynamic SQL for better performance, the response would be alarmed looks: what about SQL injection? Indeed, everyone has heard stories about stolen passwords and deleted data, because somebody was smart enough to inject a dangerous command instead of date of birth in a registration form.

True, there are multiple ways for hackers to get access to data they should not get access to. However, when we are considering dynamic SQL, there are some simple rules that help minimize possible risks.

In cases when parameter values for a function call are obtained from the database directly (i.e., referencing IDs), they can’t contain any SQL injection. Values obtained from user input must be protected with PostgreSQL functions (quote_literal, quote_indent, etc., or format).

3.1 EXECUTING DYNAMIC COMMANDS

Oftentimes you will want to generate dynamic commands inside your PL/pgSQL functions, that is, commands that will involve different tables or different data types each time they are executed.

```
EXECUTE command-string [ INTO [STRICT] target ] [ USING expression [, ... ] ];
```

where ***command-string*** is an expression yielding a string (of type text) containing the command to be executed. The optional target is a record variable, a row variable, or a comma-separated list of simple variables and record/row fields, into which the results of the command will be stored. The optional ***USING*** expressions supply values to be inserted into the command.

No substitution of PL/pgSQL variables is done on the computed command string. Any required variable values must be inserted in the command string as it is constructed; or you can use parameters as described below.

Also, there is no plan caching for commands executed via ***EXECUTE***. Instead, the command is always planned each time the statement is run. Thus, the command string can be dynamically created within the function to perform actions on different tables and columns.

The ***INTO*** clause specifies where the results of a SQL command returning rows should be assigned. If a row or variable list is provided, it must exactly match the structure of the query's results (when a record variable is used, it will configure itself to match the result structure automatically). If multiple rows are returned, only the first will be assigned to the ***INTO*** variable. If no rows are returned, NULL is assigned to the ***INTO*** variable(s). If no ***INTO*** clause is specified, the query results are discarded.

If the ***STRICT*** option is given, an error is reported unless the query produces exactly one row.

The command string can use parameter values, which are referenced in the command as $1, $2, etc. These symbols refer to values supplied in the ***USING*** clause. This method is often preferable to inserting data values into the command string as text: it avoids run-time overhead of converting the values to text and back, and it is much less prone to SQL-injection attacks since there is no need for quoting or escaping.

If you want to use dynamically determined table or column names, you must insert them into the command string textually.

```
EXECUTE 'SELECT count(*) FROM '
    || quote_ident(tabname)
    || ' WHERE inserted_by = $1 AND inserted <= $2'
   INTO c
   USING checked_user, checked_date;
```

A cleaner approach is to use format()'s %I specification for table or column names (strings separated by a newline are concatenated):

```
EXECUTE format('SELECT count(*) FROM %I '
   'WHERE inserted_by = $1 AND inserted <= $2', tabname)
   INTO c
   USING checked_user, checked_date;
```

Another restriction on parameter symbols is that they only work in SELECT, INSERT, UPDATE, and DELETE commands. In other statement types (generically called utility statements), you must insert values textually even if they are just data values.

An EXECUTE with a simple constant command string and some USING parameters is functionally equivalent to just writing the command directly in PL/pgSQL and allowing replacement of PL/pgSQL variables to happen automatically. The important difference is that EXECUTE will re-plan the command on each execution, generating a plan that is specific to the current parameter values; whereas PL/pgSQL may otherwise create a generic plan and cache it for re-use. In situations where the best plan depends strongly on the parameter values, it can be helpful to use EXECUTE to positively ensure that a generic plan is not selected.

3.2 QUOTING VALUES IN DYNAMIC QUERIES

When working with dynamic commands you will often have to handle escaping of single quotes. The recommended method for quoting fixed text in your function body is dollar quoting. Note that dollar quoting is only useful for quoting fixed text.

Dynamic values require careful handling since they might contain quote characters. An example using format() (this assumes that you are dollar quoting the function body so quote marks need not be doubled).

```
EXECUTE format('UPDATE tbl SET %I = $1 '
   'WHERE key = $2', colname) USING newvalue, keyvalue;
```

It is also possible to call the quoting functions directly by using the ***quote_ident*** and ***quote_literal*** functions. For safety, expressions containing column or table identifiers should be passed through ***quote_ident*** before insertion in a dynamic query. Expressions containing values that should be literal strings in the constructed command should be passed through ***quote_literal***. These functions take the appropriate steps to return the input text enclosed in double or single quotes respectively, with any embedded special characters properly escaped.

Because ***quote_literal*** is labeled STRICT, it will always return null when called with a null argument. In the above example, if newvalue or keyvalue were null, the entire dynamic query string would become null, leading to an error from EXECUTE. You can avoid this problem by using the ***quote_nullable*** function, which works the same as ***quote_literal*** except that when called with a null argument it returns the string NULL.

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = '
        || quote_nullable(newvalue)
        || ' WHERE key = '
        || quote_nullable(keyvalue);
```

Dynamic SQL statements can also be safely constructed using the format function where %I is equivalent to quote_ident, and %L is equivalent to quote_nullable.

```
EXECUTE format('UPDATE tbl SET %I = %L '
   'WHERE key = %L', colname, newvalue, keyvalue);
```

The format function can be used in conjunction with the USING clause. This form is better because the variables are handled in their native data type format, rather than unconditionally converting them to text and quoting them via %L. It is also more efficient.

```
EXECUTE format('UPDATE tbl SET %I = $1 WHERE key = $2', colname)
   USING newvalue, keyvalue;
```

3.3 USING IN FUNCTION

Often, it may be beneficial to build dynamic SQL inside a function and then to execute it rather than to pass parameter values as binding variables.

Now, let’s create a function with two parameters: ISO country code and the timestamp of the last update.

```
CREATE OR REPLACE FUNCTION select_booking_leg_country_dynamic (
  p_country text,
  p_updated timestamptz
)
RETURNS setof booking_leg_part
AS
$body$
BEGIN
RETURN QUERY
EXECUTE $$
          SELECT departure_airport, booking_id, is_returning
          FROM booking_leg bl
          JOIN flight f USING (flight_id)
          WHERE departure_airport IN
                    (SELECT airport_code
                    FROM airport WHERE iso_country=$$|| quote_literal(
                    p_country) ||
                    $$ AND bl.booking_id IN
                    (SELECT booking_id FROM booking
                    WHERE update_ts>$$|| quote_literal(p_updated)||$$)$$;
END;
$body$ LANGUAGE plpgsql;
```

Observe that its execution time for each set of parameters is consistent, which is exactly what we want in production systems.

Why did this behavior change? Since the SQL is built immediately prior to execution, the optimizer does not use a cached plan. Instead, it evaluates the execution plan for each execution. It may seem that this would take extra time, but in reality the opposite happens.

Also note that this function uses the ***quote_literal*** function to protect from SQL injections.

This is the first but not the only reason why using dynamic SQL in functions is beneficial.

4. TRIGGER FUNCTIONS

PL/pgSQL can be used to define trigger functions on data changes or database events. A trigger function is created with the **CREATE FUNCTION** command, declaring it as a function with no arguments and a return type of ***trigger*** (for data change triggers) or ***event_trigger*** (for database event triggers). Special local variables named TG_something are automatically defined to describe the condition that triggered the call.

4.1 TRIGGERS ON DATA CHANGES

A ***data change trigger*** is declared as a function with no arguments and a return type of trigger. Note that the function must be declared with no arguments even if it expects to receive some arguments specified in CREATE TRIGGER — such arguments are passed via TG_ARGV.

When a PL/pgSQL function is called as a trigger, several special variables are created automatically in the top-level block. They are:

- NEW - Data type RECORD; variable holding the new database row for INSERT/UPDATE operations in row-level triggers. This variable is null in statement-level triggers and for DELETE operations.
- OLD - Data type RECORD; variable holding the old database row for UPDATE/DELETE operations in row-level triggers. This variable is null in statement-level triggers and for INSERT operations.
- TG_NAME - Data type name; variable that contains the name of the trigger actually fired.
- TG_WHEN - Data type text; a string of BEFORE, AFTER, or INSTEAD OF, depending on the trigger's definition.
- TG_LEVEL - Data type text; a string of either ROW or STATEMENT depending on the trigger's definition.
- TG_OP - Data type text; a string of INSERT, UPDATE, DELETE, or TRUNCATE telling for which operation the trigger was fired.
- TG_RELID - Data type oid; the object ID of the table that caused the trigger invocation.
- TG_RELNAME - Data type name; the name of the table that caused the trigger invocation. This is now deprecated, and could disappear in a future release. Use TG_TABLE_NAME instead.
- TG_TABLE_NAME - Data type name; the name of the table that caused the trigger invocation.
- TG_TABLE_SCHEMA - Data type name; the name of the schema of the table that caused the trigger invocation.
- TG_NARGS - Data type integer; the number of arguments given to the trigger function in the CREATE TRIGGER statement.

- TG_ARGV[] - Data type array of text; the arguments from the CREATE TRIGGER statement. The index counts from 0. Invalid indexes (less than 0 or greater than or equal to tg_nargs) result in a null value.

A trigger function must return either NULL or a record/row value having exactly the structure of the table the trigger was fired for.

As an example, create trigger ensures that any insert, update or delete of a row in the emp table is recorded (i.e., audited) in the emp_audit table. The current time and user name are stamped into the row, together with the type of operation performed on it.

```
CREATE TABLE emp (
    empname           text NOT NULL,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --
        -- Create a row in emp_audit to reflect the operation performed on emp,
        -- making use of the special variable TG_OP to work out the operation.
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
        END IF;
        RETURN NULL; -- result is ignored since this is an AFTER trigger
    END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE FUNCTION process_emp_audit();
```

4.2 TRIGGERS ON EVENTS

PL/pgSQL can be used to define event triggers. PostgreSQL requires that a function that is to be called as an event trigger must be declared as a function with no arguments and a return type of event_trigger.

When a PL/pgSQL function is called as an event trigger, several special variables are created automatically in the top-level block. They are:

- TG_EVENT - Data type text; a string representing the event the trigger is fired for.
- TG_TAG - Data type text; variable that contains the command tag for which the trigger is fired.

This example trigger simply raises a NOTICE message each time a supported command is executed.

```
CREATE OR REPLACE FUNCTION snitch() RETURNS event_trigger AS $$
BEGIN
    RAISE NOTICE 'snitch: % %', tg_event, tg_tag;
END;
$$ LANGUAGE plpgsql;

CREATE EVENT TRIGGER snitch ON ddl_command_start EXECUTE FUNCTION snitch();
```

5. SOURCE BOOKS AND ARTICLES

1. Hans-Jürgen Schönig, Mastering PostgreSQL 12 Third Edition, Packt Publishing, 2019
2. PostgreSQL 13.4 Documentation, The PostgreSQL Global Development Group, 1996–2021