41.12. Porting from Oracle PL/SQL

This section explains differences between PostgreSQL's PL/pgSQL language and Oracle's PL/SQL language, to help developers who port applications from Oracle® to PostgreSQL.

PL/pgSQL is similar to PL/SQL in many aspects. It is a block-structured, imperative language, and all variables have to be declared. Assignments, loops, and conditionals are similar. The main differences you should keep in mind when porting from PL/SQL to PL/pgSQL are:

- If a name used in a SQL command could be either a column name of a table or a reference to a variable of the function, PL/SQL treats it as a column name. This corresponds to PL/pgSQL's `plpgsql.variable_conflict` = `use_column` behavior, which is not the default, as explained in Section 41.10.1. It's often best to avoid such ambiguities in the first place, but if you have to port a large amount of code that depends on this behavior, setting `variable_conflict` may be the best solution.
- In PostgreSQL the function body must be written as a string literal. Therefore you need to use dollar quoting or escape single quotes in the function body. (See Section 41.11.1.)
- Data type names often need translation. For example, in Oracle string values are commonly declared as being of type `varchar2`, which is a non-SQL-standard type. In PostgreSQL, use type `varchar` or `text` instead. Similarly, replace type `number` with `numeric`, or use some other numeric data type if there's a more appropriate one.
- Instead of packages, use schemas to organize your functions into groups.
- Since there are no packages, there are no package-level variables either. This is somewhat annoying. You can keep per-session state in temporary tables instead.
- Integer `FOR` loops with `REVERSE` work differently: PL/SQL counts down from the second number to the first, while PL/pgSQL counts down from the first number to the second, requiring the loop bounds to be swapped when porting. This incompatibility is unfortunate but is unlikely to be changed. (See Section 41.6.3.5.)
- `FOR` loops over queries (other than cursors) also work differently: the target variable(s) must have been declared, whereas PL/SQL always declares them implicitly. An advantage of this is that the variable values are still accessible after the loop exits.
- There are various notational differences for the use of cursor variables.

41.12.1. Porting Examples

Example 41-8 shows how to port a simple function from PL/SQL to PL/pgSQL.

Example 41-8. Porting a Simple Function from PL/SQL to PL/pgSQL

Here is an Oracle PL/SQL function:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar2,
                                                  v_version varchar2)
RETURN varchar2 IS
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
/
show errors;
```

Let's go through this function and see the differences compared to PL/pgSQL:

- The type name `varchar2` has to be changed to `varchar` or `text`. In the examples in this section, we'll use `varchar`, but `text` is often a better choice if you do not need specific string length limits.
- The `RETURN` key word in the function prototype (not the function body) becomes `RETURNS` in PostgreSQL. Also, `IS` becomes `AS`, and you need to add a `LANGUAGE` clause because PL/pgSQL is not the only possible function language.
- In PostgreSQL, the function body is considered to be a string literal, so you need to use quote marks or dollar quotes around it. This substitutes for the terminating `/` in the Oracle approach.
- The `show errors` command does not exist in PostgreSQL, and is not needed since errors are reported automatically.

This is how this function would look when ported to PostgreSQL:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar,
                                                  v_version varchar)
RETURNS varchar AS $$
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
$$ LANGUAGE plpgsql;
```

Example 41-9 shows how to port a function that creates another function and how to handle the ensuing quoting problems.

Example 41-9. Porting a Function that Creates Another Function from PL/SQL to PL/pgSQL

The following procedure grabs rows from a `SELECT` statement and builds a large function with the results in `IF` statements, for the sake of efficiency.

This is the Oracle version:

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
    CURSOR referrer_keys IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_cmd VARCHAR(4000);
BEGIN
    func_cmd := 'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host IN VARCHAR2,
                 v_domain IN VARCHAR2, v_url IN VARCHAR2) RETURN VARCHAR2 IS BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_cmd := func_cmd ||
          ' IF v_' || referrer_key.kind
          || ' LIKE ''' || referrer_key.key_string
          || ''' THEN RETURN ''' || referrer_key.referrer_type
          || '''; END IF;';
    END LOOP;

    func_cmd := func_cmd || ' RETURN NULL; END;';

    EXECUTE IMMEDIATE func_cmd;
END;
/
show errors;
```

Here is how this function would end up in PostgreSQL:

```
CREATE OR REPLACE FUNCTION cs_update_referrer_type_proc() RETURNS void AS $func$
DECLARE
    referrer_keys CURSOR IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_body text;
    func_cmd text;
BEGIN
    func_body := 'BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_body := func_body ||
          ' IF v_' || referrer_key.kind
          || ' LIKE ' || quote_literal(referrer_key.key_string)
          || ' THEN RETURN ' || quote_literal(referrer_key.referrer_type)
          || '; END IF;' ;
    END LOOP;

    func_body := func_body || ' RETURN NULL; END;';

    func_cmd :=
      'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host varchar,
                                                        v_domain varchar,
                                                        v_url varchar)
        RETURNS varchar AS '
      || quote_literal(func_body)
      || ' LANGUAGE plpgsql;' ;

    EXECUTE func_cmd;
END;
$func$ LANGUAGE plpgsql;
```

Notice how the body of the function is built separately and passed through `quote_literal` to double any quote marks in it. This technique is needed because we cannot safely use dollar quoting for defining the new function: we do not know for sure what strings will be interpolated from the `referrer_key.key_string` field. (We are assuming here that `referrer_key.kind` can be trusted to always be `host`, `domain`, or `url`, but `referrer_key.key_string` might be anything, in particular it might contain dollar signs.) This function is actually an improvement on the Oracle original, because it will not generate broken code when `referrer_key.key_string` or `referrer_key.referrer_type` contain quote marks.

Example 41-10 shows how to port a function with `OUT` parameters and string manipulation. PostgreSQL does not have a built-in `instr` function, but you can create one using a combination of other functions. In Section 41.12.3 there is a PL/pgSQL implementation of `instr` that you can use to make your porting easier.

Example 41-10. Porting a Procedure With String Manipulation and `OUT` Parameters from PL/SQL to PL/pgSQL

The following Oracle PL/SQL procedure is used to parse a URL and return several elements (host, path, and query).

This is the Oracle version:

```
CREATE OR REPLACE PROCEDURE cs_parse_url(
    v_url IN VARCHAR2,
    v_host OUT VARCHAR2,  -- This will be passed back
    v_path OUT VARCHAR2,  -- This one too
    v_query OUT VARCHAR2) -- And this one
IS
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
/
show errors;
```

Here is a possible translation into PL/pgSQL:

```
CREATE OR REPLACE FUNCTION cs_parse_url(
    v_url IN VARCHAR,
    v_host OUT VARCHAR,  -- This will be passed back
    v_path OUT VARCHAR,  -- This one too
    v_query OUT VARCHAR) -- And this one
AS $$
DECLARE
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
$$ LANGUAGE plpgsql;
```

This function could be used like this:

```
SELECT * FROM cs_parse_url('http://foobar.com/query.cgi?baz');
```

Example 41-11 shows how to port a procedure that uses numerous features that are specific to Oracle.

Example 41-11. Porting a Procedure from PL/SQL to PL/pgSQL

The Oracle version:

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
    a_running_job_count INTEGER;
    PRAGMA AUTONOMOUS_TRANSACTION;(1)
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;(2)

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- free lock(3)
        raise_application_error(-20000,
                 'Unable to create a new job: a job is currently running.');
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
    EXCEPTION
        WHEN dup_val_on_index THEN NULL; -- don't worry if it already exists
    END;
    COMMIT;
END;
/
show errors
```

Procedures like this can easily be converted into PostgreSQL functions returning `void`. This procedure in particular is interesting because it can teach us some things:

(1)
There is no `PRAGMA` statement in PostgreSQL.

(2)
If you do a `LOCK TABLE` in PL/pgSQL, the lock will not be released until the calling transaction is finished.

(3)
You cannot issue `COMMIT` in a PL/pgSQL function. The function is running within some outer transaction and so `COMMIT` would imply terminating the function's execution. However, in this particular case it is not necessary anyway, because the lock obtained by the `LOCK TABLE` will be released when we raise an error.

This is how we could port this procedure to PL/pgSQL:

```
CREATE OR REPLACE FUNCTION cs_create_job(v_job_id integer) RETURNS void AS $$
DECLARE
    a_running_job_count integer;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        RAISE EXCEPTION 'Unable to create a new job: a job is currently running';(1)
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
    EXCEPTION
        WHEN unique_violation THEN (2)
            -- don't worry if it already exists
    END;
END;
$$ LANGUAGE plpgsql;
```

(1)
The syntax of `RAISE` is considerably different from Oracle's statement, although the basic case `RAISE` ***`exception_name`*** works similarly.

(2)
The exception names supported by PL/pgSQL are different from Oracle's. The set of built-in exception names is much larger (see Appendix A). There is not currently a way to declare user-defined exception names, although you can throw user-chosen SQLSTATE values instead.

The main functional difference between this procedure and the Oracle equivalent is that the exclusive lock on the `cs_jobs` table will be held until the calling transaction completes. Also, if the caller later aborts (for example due to an error), the effects of this procedure will be rolled back.

41.12.2. Other Things to Watch For

This section explains a few other things to watch for when porting Oracle PL/SQL functions to PostgreSQL.

41.12.2.1. Implicit Rollback after Exceptions

In PL/pgSQL, when an exception is caught by an `EXCEPTION` clause, all database changes since the block's `BEGIN` are automatically rolled back. That is, the behavior is equivalent to what you'd get in Oracle with:

```
BEGIN
    SAVEPOINT s1;
    ... code here ...
EXCEPTION
    WHEN ... THEN
        ROLLBACK TO s1;
        ... code here ...
    WHEN ... THEN
        ROLLBACK TO s1;
        ... code here ...
END;
```

If you are translating an Oracle procedure that uses `SAVEPOINT` and `ROLLBACK TO` in this style, your task is easy: just omit the `SAVEPOINT` and `ROLLBACK TO`. If you have a procedure that uses `SAVEPOINT` and `ROLLBACK TO` in a different way then some actual thought will be required.

41.12.2.2. EXECUTE

The PL/pgSQL version of `EXECUTE` works similarly to the PL/SQL version, but you have to remember to use `quote_literal` and `quote_ident` as described in Section 41.5.4. Constructs of the type `EXECUTE 'SELECT * FROM $1';` will not work reliably unless you use these functions.

41.12.2.3. Optimizing PL/pgSQL Functions

PostgreSQL gives you two function creation modifiers to optimize execution: "volatility" (whether the function always returns the same result when given the same arguments) and "strictness" (whether the function returns null if any argument is null). Consult the CREATE FUNCTION reference page for details.

When making use of these optimization attributes, your `CREATE FUNCTION` statement might look something like this:

```
CREATE FUNCTION foo(...) RETURNS integer AS $$
...
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

41.12.3. Appendix

This section contains the code for a set of Oracle-compatible `instr` functions that you can use to simplify your porting efforts.

```
--
-- instr functions that mimic Oracle's counterpart
-- Syntax: instr(string1, string2 [, n [, m]])
-- where [] denotes optional parameters.
--
-- Search string1, beginning at the nth character, for the mth occurrence
-- of string2.  If n is negative, search backwards, starting at the abs(n)'th
-- character from the end of string1.
-- If n is not passed, assume 1 (search starts at first character).
-- If m is not passed, assume 1 (find first occurrence).
-- Returns starting index of string2 in string1, or 0 if string2 is not found.
--

CREATE FUNCTION instr(varchar, varchar) RETURNS integer AS $$
BEGIN
    RETURN instr($1, $2, 1);
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;


CREATE FUNCTION instr(string varchar, string_to_search_for varchar,
                      beg_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        temp_str := substring(string FROM beg_index);
        pos := position(string_to_search_for IN temp_str);

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN pos + beg_index - 1;
        END IF;
    ELSIF beg_index < 0 THEN
        ss_length := char_length(string_to_search_for);
        length := char_length(string);
        beg := length + 1 + beg_index;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            IF string_to_search_for = temp_str THEN
                RETURN beg;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    ELSE
        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;


CREATE FUNCTION instr(string varchar, string_to_search_for varchar,
                      beg_index integer, occur_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    occur_number integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    i integer;
    length integer;
    ss_length integer;
BEGIN
    IF occur_index <= 0 THEN
        RAISE 'argument ''%'' is out of range', occur_index
          USING ERRCODE = '22003';
    END IF;

    IF beg_index > 0 THEN
        beg := beg_index - 1;
        FOR i IN 1..occur_index LOOP
            temp_str := substring(string FROM beg + 1);
            pos := position(string_to_search_for IN temp_str);
            IF pos = 0 THEN
                RETURN 0;
            END IF;
            beg := beg + pos;
        END LOOP;

        RETURN beg;
    ELSIF beg_index < 0 THEN
        ss_length := char_length(string_to_search_for);
        length := char_length(string);
        beg := length + 1 + beg_index;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            IF string_to_search_for = temp_str THEN
                occur_number := occur_number + 1;
                IF occur_number = occur_index THEN
                    RETURN beg;
                END IF;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    ELSE
        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```