41.5. Basic Statements

In this section and the following ones, we describe all the statement types that are explicitly understood by PL/pgSQL. Anything not recognized as one of these statement types is presumed to be an SQL command and is sent to the main database engine to execute, as described in Section 41.5.2 and Section 41.5.3.

41.5.1. Assignment

An assignment of a value to a PL/pgSQL variable is written as:

```
variable { := | = } expression;
```

As explained previously, the expression in such a statement is evaluated by means of an SQL SELECT command sent to the main database engine. The expression must yield a single value (possibly a row value, if the variable is a row or record variable). The target variable can be a simple variable (optionally qualified with a block name), a field of a row or record variable, or an element of an array that is a simple variable or field. Equal (=) can be used instead of PL/SQL-compliant :=.

If the expression's result data type doesn't match the variable's data type, the value will be coerced as though by an assignment cast (see Section 10.4). If no assignment cast is known for the pair of data types involved, the PL/pgSQL interpreter will attempt to convert the result value textually, that is by applying the result type's output function followed by the variable type's input function. Note that this could result in run-time errors generated by the input function, if the string form of the result value is not acceptable to the input function.

Examples:

```
tax := subtotal * 0.06;
my_record.user_id := 20;
```

41.5.2. Executing a Command With No Result

For any SQL command that does not return rows, for example INSERT without a RETURNING clause, you can execute the command within a PL/pgSQL function just by writing the command.

Any PL/pgSQL variable name appearing in the command text is treated as a parameter, and then the current value of the variable is provided as the parameter value at run time. This is exactly like the processing described earlier for expressions; for details see Section 41.10.1.

When executing a SQL command in this way, PL/pgSQL may cache and re-use the execution plan for the command, as discussed in Section 41.10.2.

Sometimes it is useful to evaluate an expression or SELECT query but discard the result, for example when calling a function that has side-effects but no useful result value. To do this in PL/pgSQL, use the PERFORM statement:

```
PERFORM query;
```

The execution query and discards the result. Write the **query** the same way you would write an SQL SELECT command, but replace the initial keyword SELECT with PERFORM. For WITH queries, use PERFORM and then place the query in parentheses. (In this case, the query can only return one row.) PL/pgSQL variables will be substituted into the query just as for commands that return no result, and the plan is cached in the same way. Also, the special variable FOUND is set to true if the query produced at least one row, or false if it produced no rows (see Section 41.5.5).

Note: One might expect that writing SELECT directly would accomplish this result, but at present the only accepted way to do it is PERFORM. A query that can return rows, such as SELECT, will be rejected as an error unless it has an INTO clause as discussed in the next section.

An example:

```
PERFORM create_mv('cs_session_page_requests_mv', my_query);
```

41.5.3. Executing a Query with a Single-row Result

The result of a SQL command yielding a single row (possibly of multiple columns) can be assigned to a record variable, row-type variable, or list of scalar variables. This is done by writing the base SELECT command and adding an INTO clause. For example,

```
SELECT select_expressions INTO [STRICT] target FROM ...;
INSERT ... RETURNING expressions INTO [STRICT] target;
UPDATE ... RETURNING expressions INTO [STRICT] target;
DELETE ... RETURNING expressions INTO [STRICT] target;
```

where **target** can be a record variable, a row variable, or a comma-separated list of simple variables and record/row fields. PL/pgSQL variables will be substituted into the rest of the query, and the plan is cached, just as described above for commands that do not return rows. This works for SELECT, INSERT/UPDATE/DELETE with RETURNING, and utility commands that return row-set results (such as EXPLAIN). Except for the INTO clause, the SQL command is the same as it would be written outside PL/pgSQL.

Tip: Note that this interpretation of SELECT with INTO is quite different from PostgreSQL's regular SELECT INTO command, wherein the INTO target is a newly created table. If you want to create a table from a SELECT result inside a PL/pgSQL function, use the syntax CREATE TABLE ... AS SELECT.

If a row or a variable list is used as target, the query's result columns must exactly match the structure of the target as to number and data types, or else a run-time error occurs. When a record variable is the target, it automatically configures itself to the row type of the query result columns.

The INTO clause can appear almost anywhere in the SQL command. Customarily it is written either just before or just after the list of **select_expressions** in a SELECT command, or at the end of the command for other command types. It is recommended to follow this convention in case the PL/pgSQL parser becomes stricter in future versions.

If STRICT is not specified in the INTO clause, then **target** will be set to the first row returned by the query, or to nulls if the query returned no rows. (Think first row" is not well-defined unless you've used ORDER BY.) Any result rows after the first row are discarded. You can check the special FOUND variable (see Section 41.5.5) to determine whether a row was returned:

```
SELECT * INTO myrec FROM emp WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION 'employee % not found', myname;
END IF;
```

If the STRICT option is specified, the query must return exactly one row or a run-time error will be reported, either NO_DATA_FOUND (no rows) or TOO_MANY_ROWS (more than one row). You can use an exception block if you wish to catch the error, for example:

```
BEGIN
    SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RAISE EXCEPTION 'employee % not found', myname;
        WHEN TOO_MANY_ROWS THEN
            RAISE EXCEPTION 'employee % not unique', myname;
END;
```

Successful execution of a command with STRICT always sets FOUND to true.

For INSERT/UPDATE/DELETE with RETURNING, PL/pgSQL reports an error for more than one returned row, even when STRICT is not specified. This is because there is no option such as ORDER BY with which to determine which affected row should be returned.

If print_strict_params is enabled for the function, then when an error is thrown because the requirements of STRICT are not met, the DETAIL part of the error message will include information about the parameters passed to the query. You can change the print_strict_params setting for all functions by setting plpgsql.print_strict_params, though only subsequent function compilations will be affected. You can also enable it on a per-function basis by using a compiler option, for example:

```
CREATE FUNCTION get_user(id username_text) RETURNING int
AS $$
#print_strict_params on
DECLARE
    rec int;
BEGIN
    SELECT users.userid INTO STRICT userid
        FROM users WHERE users.username = get_userid.username;
    RETURN userid;
END;
$$ LANGUAGE plpgsql;
```

On failure, this function might produce an error message such as

```
ERROR:  query returned no rows
DETAIL:  parameters: $1 = 'nosuchuser'
CONTEXT:  PL/pgSQL function get_user(text) line 6 at SQL statement
```

Note: The STRICT option matches the behavior of Oracle PL/SQL's SELECT INTO and related statements.

To handle cases where you need to process multiple result rows from a SQL query, see Section 41.6.4.

41.5.4. 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. PL/pgSQL's normal attempts to cache plans for commands (as discussed in Section 41.10.2) will not work in such scenarios. To handle this sort of problem, the EXECUTE statement is provided:

```
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 in 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. An example is:

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

Note that parameter symbols can only be used for data values — if you want to use dynamically determined table or column names, you must insert them into the command string textually. For example, if the preceding query needed to be done against a dynamically selected table, you could do this:

```
EXECUTE 'SELECT count(*) FROM '
    || quote_ident(tablename)
    || ' 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', tablename)
    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, as in the first example above, 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; 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 prevent a generic plan from being selected.

SELECT INTO is not currently supported within EXECUTE; instead, execute a plain SELECT command and specify INTO as part of EXECUTE's result clause.

Note: The PL/pgSQL EXECUTE statement is not related to the EXECUTE SQL statement supported by the PostgreSQL server. The server's EXECUTE statement cannot be used directly within PL/pgSQL functions (and is not needed).

Example 41-1. 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. (If you have legacy code that does not use dollar quoting, please refer to the overview in Section 41.12.1, which can save you some effort when translating said code to a more readable form.)

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:

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

This example demonstrates the use of the quote_ident and quote_literal functions (see Section 9.4). 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 on a null argument it returns the string NULL. For example,

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

If you are dealing with values that might be null, you should usually use quote_nullable in place of quote_literal.

As always, care must be taken to ensure that null values in a query do not deliver unintended results. For example the WHERE clause

```
'WHERE key = ' || quote_nullable(keyvalue)
```

will never succeed if keyvalue is null, because the result of using the equality operator with a null operand is always null. If you wish null to work like an ordinary key value, you would need to rewrite the above as

```
'WHERE key IS NOT DISTINCT FROM ' || quote_nullable(keyvalue)
```

(At present, IS NOT DISTINCT FROM is handled much less efficiently than =, so don't do this unless you must. See Section 9.2 for more information on nulls and IS DISTINCT.)

Note that dollar quoting is only useful for quoting fixed text. It would be a very bad idea to try to write this example as:

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

because it would break if the contents of newvalue happened to contain $$. The same objection would apply to any other dollar-quoting delimiter you might pick. So, to safely quote text that is not known in advance, you must use quote_literal, quote_nullable, or quote_ident, as appropriate.

Dynamic SQL statements can also be safely constructed using the format function (see Section 9.4). For example:

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

%I is equivalent to quote_ident, and %L is equivalent to quote_nullable. The format function can be used in conjunction with the USING clause:

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

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.

A much larger example of a dynamic command and EXECUTE can be seen in Example 41-9, which builds and executes a CREATE FUNCTION command to define a new function.

41.5.5. Obtaining the Result Status

There are several ways to determine the effect of a command. The first method is to use the GET DIAGNOSTICS command, which has the form:

```
GET [ CURRENT ] DIAGNOSTICS variable { = | := } item [ , ... ];
```

This command retrieves system status indicators. CURRENT is a noise word (but one also GET STACKED DIAGNOSTICS in Section 41.6.6.5). Each item is a key word identifying a status value to be assigned to the specified **variable** (which should be of the right data type to receive it). The currently available status items are shown in Table 41.1. Column names may be used instead of the SQL-standard ROW_COUNT.

```
GET DIAGNOSTICS integer_var = ROW_COUNT;
```

Table 41.1. Available Diagnostics Items

Name	Type	Description
ROW_COUNT	bigint	the number of rows processed by the most recent SQL command
RESULT_OID	oid	the OID of the last row inserted by the most recent SQL command (only useful after an INSERT command into a table having OIDs)
PG_CONTEXT	text	text(s) of text describing the current call stack (see Section 41.6.7)

The second method to determine the effects of a command is to check the special variable named FOUND, which is of type boolean. FOUND starts out false within each PL/pgSQL function call. It is set by each of the following types of statements:

- A SELECT INTO statement sets FOUND true if a row is assigned, false if no row is returned.
- A PERFORM statement sets FOUND true if it produces (and discards) one or more rows, false if no row is produced.
- UPDATE, INSERT, and DELETE statements set FOUND true if at least one row is affected, false if no row is affected.
- A FETCH statement sets FOUND true if it returns a row, false if no row is returned.
- A MOVE statement sets FOUND true if it successfully repositions the cursor, false otherwise.
- A FOR or FOREACH statement sets FOUND true if it iterates one or more times, else false. FOUND is set this way when the loop exits; inside the execution of the loop, FOUND is not modified by the loop statement, although it might be changed by the execution of other statements within the loop body.
- RETURN QUERY and RETURN QUERY EXECUTE statements set FOUND true if the query returns at least one row, false if no row is returned.

Other PL/pgSQL statements do not change the state of FOUND. Note in particular that EXECUTE changes the output of GET DIAGNOSTICS, but does not change FOUND.

FOUND is a local variable within each PL/pgSQL function; any changes to it affect only the current function.

41.5.6. Doing Nothing At All

Sometimes a placeholder statement that does nothing is useful. For example, it can indicate that one arm of an if/then/else chain is deliberately empty. For this purpose, use the NULL statement:

```
NULL;
```

For example, the following two fragments of code are equivalent:

```
BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        NULL;  -- ignore the error
END;
```

```
BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN -- ignore the error
END;
```

Which is preferable is a matter of taste.

Note: In Oracle's PL/SQL, empty statement lists are not allowed, and so NULL statements are required for situations such as this. PL/pgSQL allows you to just write nothing, instead.