Chapter 39. PL/pgSQL - SQL Procedural Language

39.1. Overview

PL/pgSQL is a loadable procedural language for the PostgreSQL database system. The design goals of PL/pgSQL were to create a loadable procedural language that

- can be used to create functions and trigger procedures,
- adds control structures to the SQL language,
- can perform complex computations,
- inherits all user-defined types, functions, and operators,
- can be defined to be trusted by the server,
- is easy to use.

Functions created with PL/pgSQL can be used anywhere that built-in functions could be used. For example, it is possible to create complex conditional computation functions and later use them to define operators or use them in index expressions.

In PostgreSQL 9.0 and later, PL/pgSQL is installed by default. However it is still a loadable module, so especially security-conscious administrators could choose to remove it.

39.1.1. Advantages of Using PL/pgSQL

SQL is the language PostgreSQL and most other relational databases use as query language. It's portable and easy to learn. But every SQL statement must be executed individually by the database server.

That means that your client application must send each query to the database server, wait for it to be processed, receive and process the results, do some computation, then send further queries to the server. All this incurs interprocess communication and will also incur network overhead if your client is on a different machine than the database server.

With PL/pgSQL you can group a block of computation and a series of queries *inside* the database server, thus having the power of a procedural language and the ease of use of SQL, but with considerable savings of client/server communication overhead.

- Extra round trips between client and server are eliminated
- Intermediate results that the client does not need do not have to be marshaled or transferred between server and client
- Multiple rounds of query parsing can be avoided

This can result in a considerable performance increase as compared to an application that does not use stored functions.

Also, with PL/pgSQL you can use all the data types, operators and functions of SQL.

39.1.2. Supported Argument and Result Data Types

Functions written in PL/pgSQL can accept as arguments any scalar or array data type supported by the server, and they can return a result of any of these types. They can also accept or return any composite type (row type) specified by name. It is also possible to declare a PL/pgSQL function as returning `record`, which means that the result is a row type whose columns are determined by specification in the calling query, as discussed in Section 7.2.1.4.

PL/pgSQL functions can be declared to accept a variable number of arguments by using the `VARIADIC` marker. This works exactly the same way as for SQL functions, as discussed in Section 35.4.5.

PL/pgSQL functions can also be declared to accept and return the polymorphic types `anyelement`, `anyarray`, `anynonarray`, and `anyenum`. The actual data types handled by a polymorphic function can vary from call to call, as discussed in Section 35.2.5. An example is shown in Section 39.3.1.

PL/pgSQL functions can also be declared to return a "set" (or table) of any data type that can be returned as a single instance. Such a function generates its output by executing `RETURN NEXT` for each desired element of the result set, or by using `RETURN QUERY` to output the result of evaluating a query.

Finally, a PL/pgSQL function can be declared to return `void` if it has no useful return value.

PL/pgSQL functions can also be declared with output parameters in place of an explicit specification of the return type. This does not add any fundamental capability to the language, but it is often convenient, especially for returning multiple values. The `RETURNS TABLE` notation can also be used in place of `RETURNS SETOF`.

Specific examples appear in Section 39.3.1 and Section 39.6.1.

39.2. Structure of PL/pgSQL

PL/pgSQL is a block-structured language. The complete text of a function definition must be a *block*. A block is defined as:

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END [ label ];
```

Each declaration and each statement within a block is terminated by a semicolon. A block that appears within another block must have a semicolon after `END`, as shown above; however the final `END` that concludes a function body does not require a semicolon.

Tip: A common mistake is to write a semicolon immediately after `BEGIN`. This is incorrect and will result in a syntax error.

A *`label`* is only needed if you want to identify the block for use in an `EXIT` statement, or to qualify the names of the variables declared in the block. If a label is given after `END`, it must match the label at the block's beginning.

All key words are case-insensitive. Identifiers are implicitly converted to lower case unless double-quoted, just as they are in ordinary SQL commands.

Comments work the same way in PL/pgSQL code as in ordinary SQL. A double dash (`--`) starts a comment that extends to the end of the line. A `/*` starts a block comment that extends to the matching occurrence of `*/`. Block comments nest.

Any statement in the statement section of a block can be a *subblock*. Subblocks can be used for logical grouping or to localize variables to a small group of statements. Variables declared in a subblock mask any similarly-named variables of outer blocks for the duration of the subblock; but you can access the outer variables anyway if you qualify their names with their block's label. For example:

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 30
    quantity := 50;
    --
    -- Create a subblock
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 80
        RAISE NOTICE 'Outer quantity here is %', outerblock.quantity;  -- Prints 50
    END;

    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 50

    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

Note: There is actually a hidden "outer block" surrounding the body of any PL/pgSQL function. This block provides the declarations of the function's parameters (if any), as well as some special variables such as `FOUND` (see Section 39.5.5). The outer block is labeled with the function's name, meaning that parameters and special variables can be qualified with the function's name.

It is important not to confuse the use of `BEGIN`/`END` for grouping statements in PL/pgSQL with the similarly-named SQL commands for transaction control. PL/pgSQL's `BEGIN`/`END` are only for grouping; they do not start or end a transaction. Functions and trigger procedures are always executed within a transaction established by an outer query — they cannot start or commit that transaction, since there would be no context for them to execute in. However, a block containing an `EXCEPTION` clause effectively forms a subtransaction that can be rolled back without affecting the outer transaction. For more about that see Section 39.6.6.

39.3. Declarations

All variables used in a block must be declared in the declarations section of the block. (The only exceptions are that the loop variable of a FOR loop iterating over a range of integer values is automatically declared as an integer variable, and likewise the loop variable of a FOR loop iterating over a cursor's result is automatically declared as a record variable.)

PL/pgSQL variables can have any SQL data type, such as integer, varchar, and char.

Here are some examples of variable declarations:

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

The general syntax of a variable declaration is:

```
name [ CONSTANT ] type [ COLLATE collation_name ] [ NOT NULL ] [ { DEFAULT | := } expressio
```

The DEFAULT clause, if given, specifies the initial value assigned to the variable when the block is entered. If the DEFAULT clause is not given then the variable is initialized to the SQL null value. The CONSTANT option prevents the variable from being assigned to after initialization, so that its value will remain constant for the duration of the block. The COLLATE option specifies a collation to use for the variable (see Section 39.3.6). If NOT NULL is specified, an assignment of a null value results in a run-time error. All variables declared as NOT NULL must have a nonnull default value specified.

A variable's default value is evaluated and assigned to the variable each time the block is entered (not just once per function call). So, for example, assigning now() to a variable of type timestamp causes the variable to have the time of the current function call, not the time when the function was precompiled.

Examples:

```
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

39.3.1. Declaring Function Parameters

Parameters passed to functions are named with the identifiers $1, $2, etc. Optionally, aliases can be declared for $*n* parameter names for increased readability. Either the alias or the numeric identifier can then be used to refer to the parameter value.

There are two ways to create an alias. The preferred way is to give a name to the parameter in the CREATE FUNCTION command, for example:

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

The other way, which was the only way available before PostgreSQL 8.0, is to explicitly declare an alias, using the declaration syntax

```
name ALIAS FOR $n;
```

The same example in this style looks like:

```
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

> **Note:** These two examples are not perfectly equivalent. In the first case, `subtotal` could be referenced as `sales_tax.subtotal`, but in the second case it could not. (Had we attached a label to the inner block, `subtotal` could be qualified with that label, instead.)

Some more examples:

```
CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN
    -- some computations using v_string and index here
END;
$$ LANGUAGE plpgsql;


CREATE FUNCTION concat_selected_fields(in_t sometablename) RETURNS text AS $$
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;
```

When a PL/pgSQL function is declared with output parameters, the output parameters are given `$n` names and optional aliases in just the same way as the normal input parameters. An output parameter is effectively a variable that starts out NULL; it should be assigned to during the execution of the function. The final value of the parameter is what is returned. For instance, the sales-tax example could also be done this way:

```
CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
    tax := subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Notice that we omitted `RETURNS real` — we could have included it, but it would be redundant.

Output parameters are most useful when returning multiple values. A trivial example is:

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int) AS $$
BEGIN
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

As discussed in Section 35.4.4, this effectively creates an anonymous record type for the function's results. If a `RETURNS` clause is given, it must say `RETURNS record`.

Another way to declare a PL/pgSQL function is with `RETURNS TABLE`, for example:

```
CREATE FUNCTION extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
    RETURN QUERY SELECT quantity, quantity * price FROM sales
                 WHERE itemno = p_itemno;
END;
$$ LANGUAGE plpgsql;
```

This is exactly equivalent to declaring one or more `OUT` parameters and specifying `RETURNS SETOF` *`sometype`*.

When the return type of a PL/pgSQL function is declared as a polymorphic type (`anyelement`, `anyarray`, `anynonarray`, or `anyenum`), a special parameter `$0` is created. Its data type is the actual return type of the function, as deduced from the actual input types (see Section 35.2.5). This allows the function to access its actual return type as shown in Section 39.3.3. `$0` is initialized to null and can be modified by the function, so it can be used to hold the return value if desired, though that is not required. `$0` can also be given an alias. For example, this function works on any data type that has a + operator:

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS $$
DECLARE
    result ALIAS FOR $0;
BEGIN
    result := v1 + v2 + v3;
    RETURN result;
END;
$$ LANGUAGE plpgsql;
```

The same effect can be had by declaring one or more output parameters as polymorphic types. In this case the special `$0` parameter is not used; the output parameters themselves serve the same purpose. For example:

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement,
                                 OUT sum anyelement)
AS $$
BEGIN
    sum := v1 + v2 + v3;
END;
$$ LANGUAGE plpgsql;
```

39.3.2. ALIAS

```
newname ALIAS FOR oldname;
```

The `ALIAS` syntax is more general than is suggested in the previous section: you can declare an alias for any variable, not just function parameters. The main practical use for this is to assign a different name for variables with predetermined names, such as `NEW` or `OLD` within a trigger procedure.

Examples:

```
DECLARE
  prior ALIAS FOR old;
  updated ALIAS FOR new;
```

Since `ALIAS` creates two different ways to name the same object, unrestricted use can be confusing. It's best to use it only for the purpose of overriding predetermined names.

39.3.3. Copying Types

```
variable%TYPE
```

`%TYPE` provides the data type of a variable or table column. You can use this to declare variables that will hold database values. For example, let's say you have a column named `user_id` in your `users` table. To declare a variable with the same data type as `users.user_id` you write:

```
user_id users.user_id%TYPE;
```

By using `%TYPE` you don't need to know the data type of the structure you are referencing, and most importantly, if the data type of the referenced item changes in the future (for instance: you change the type of `user_id` from `integer` to `real`), you might not need to change your function definition.

`%TYPE` is particularly valuable in polymorphic functions, since the data types needed for internal variables can change from one call to the next. Appropriate variables can be created by applying `%TYPE` to the function's arguments or result placeholders.

39.3.4. Row Types

```
name table_name%ROWTYPE;
name composite_type_name;
```

A variable of a composite type is called a *row* variable (or *row-type* variable). Such a variable can hold a whole row of a `SELECT` or `FOR` query result, so long as that query's column set matches the declared type of the variable. The individual fields of the row value are accessed using the usual dot notation, for example `rowvar.field`.

A row variable can be declared to have the same type as the rows of an existing table or view, by using the *`table_name`*`%ROWTYPE` notation; or it can be declared by giving a composite type's name. (Since every table has an associated composite type of the same name, it actually does not matter in PostgreSQL whether you write `%ROWTYPE` or not. But the form with `%ROWTYPE` is more portable.)

Parameters to a function can be composite types (complete table rows). In that case, the corresponding identifier `$n` will be a row variable, and fields can be selected from it, for example `$1.user_id`.

Only the user-defined columns of a table row are accessible in a row-type variable, not the OID or other system columns (because the row could be from a view). The fields of the row type inherit the table's field size or precision for data types such as `char(n)`.

Here is an example of using composite types. `table1` and `table2` are existing tables having at least the mentioned fields:

```
CREATE FUNCTION merge_fields(t_row table1) RETURNS text AS $$
DECLARE
    t2_row table2%ROWTYPE;
BEGIN
    SELECT * INTO t2_row FROM table2 WHERE ... ;
    RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;

SELECT merge_fields(t.*) FROM table1 t WHERE ... ;
```

39.3.5. Record Types

```
name RECORD;
```

Record variables are similar to row-type variables, but they have no predefined structure. They take on the actual row structure of the row they are assigned during a `SELECT` or `FOR` command. The substructure of a record variable can change each time it is assigned to. A consequence of this is that until a record variable is first assigned to, it has no substructure, and any attempt to access a field in it will draw a run-time error.

Note that `RECORD` is not a true data type, only a placeholder. One should also realize that when a PL/pgSQL function is declared to return type `record`, this is not quite the same concept as a record variable, even though such a function might use a record variable to hold its result. In both cases the actual row structure is unknown when the function is written, but for a function returning `record` the actual structure is determined when the calling query is parsed, whereas a record variable can change its row structure on-the-fly.

39.3.6. Collation of PL/pgSQL Variables

When a PL/pgSQL function has one or more parameters of collatable data types, a collation is identified for each function call depending on the collations assigned to the actual arguments, as described in Section 22.2. If a collation is successfully identified (i.e., there are no conflicts of implicit collations among the arguments) then all the collatable parameters are treated as having that collation implicitly. This will affect the behavior of collation-sensitive operations within the function. For example, consider

```
CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
BEGIN
    RETURN a < b;
END;
$$ LANGUAGE plpgsql;

SELECT less_than(text_field_1, text_field_2) FROM table1;
SELECT less_than(text_field_1, text_field_2 COLLATE "C") FROM table1;
```

The first use of `less_than` will use the common collation of `text_field_1` and `text_field_2` for the comparison, while the second use will use `C` collation.

Furthermore, the identified collation is also assumed as the collation of any local variables that are of collatable types. Thus this function would not work any differently if it were written as

```
CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
DECLARE
    local_a text := a;
    local_b text := b;
BEGIN
    RETURN local_a < local_b;
END;
$$ LANGUAGE plpgsql;
```

If there are no parameters of collatable data types, or no common collation can be identified for them, then parameters and local variables use the default collation of their data type (which is usually the database's default collation, but could be different for variables of domain types).

A local variable of a collatable data type can have a different collation associated with it by including the `COLLATE` option in its declaration, for example

```
DECLARE
    local_a text COLLATE "en_US";
```

This option overrides the collation that would otherwise be given to the variable according to the rules above.

Also, of course explicit `COLLATE` clauses can be written inside a function if it is desired to force a particular collation to be used in a particular operation. For example,

```
CREATE FUNCTION less_than_c(a text, b text) RETURNS boolean AS $$
BEGIN
    RETURN a < b COLLATE "C";
END;
$$ LANGUAGE plpgsql;
```

This overrides the collations associated with the table columns, parameters, or local variables used in the expression, just as would happen in a plain SQL command.

39.4. Expressions

All expressions used in PL/pgSQL statements are processed using the server's main SQL executor. For example, when you write a PL/pgSQL statement like

```
IF expression THEN ...
```

PL/pgSQL will evaluate the expression by feeding a query like

```
SELECT expression
```

to the main SQL engine. While forming the `SELECT` command, any occurrences of PL/pgSQL variable names are replaced by parameters, as discussed in detail in Section 39.10.1. This allows the query plan for the `SELECT` to be prepared just once and then reused for subsequent evaluations with different values of the variables. Thus, what really happens on first use of an expression is essentially a `PREPARE` command. For example, if we have declared two integer variables `x` and `y`, and we write

```
IF x < y THEN ...
```

what happens behind the scenes is equivalent to

```
PREPARE statement_name(integer, integer) AS SELECT $1 < $2;
```

and then this prepared statement is `EXECUTE`d for each execution of the `IF` statement, with the current values of the PL/pgSQL variables supplied as parameter values. The query plan prepared in this way is saved for the life of the database connection, as described in Section 39.10.2. Normally these details are not important to a PL/pgSQL user, but they are useful to know when trying to diagnose a problem.

39.5. 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 39.5.2 and Section 39.5.3.

39.5.1. 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.

If the expression's result data type doesn't match the variable's data type, or the variable has a specific size/precision (like `char(20)`), the result value will be implicitly converted by the PL/pgSQL interpreter using the result type's output-function and the variable type's input-function. Note that this could potentially 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;
```

39.5.2. 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 39.10.1.

When executing a SQL command in this way, PL/pgSQL plans the command just once and re-uses the plan on subsequent executions, for the life of the database connection. The implications of this are discussed in detail in Section 39.10.2.

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;
```

This executes *`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 39.5.5).

> **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 SQL command 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);
```

39.5.3. 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 SQL 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 that you 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. (Note that "the 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 39.5.5) 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.

> **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 39.6.4.

39.5.4. 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 39.10.2) 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 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 prepared 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 '
    || tabname::regclass
    || ' WHERE inserted_by = $1 AND inserted <= $2'
   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; whereas PL/pgSQL normally creates a generic plan and caches it for re-use. In situations where the best plan depends strongly on the parameter values, `EXECUTE` can be significantly faster; while when the plan is not sensitive to parameter values, re-planning will be a waste.

`SELECT INTO` is not currently supported within `EXECUTE`; instead, execute a plain `SELECT` command and specify `INTO` as part of the `EXECUTE` itself.

> **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 39-1. 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 39.11.1, which can save you some effort when translating said code to a more reasonable scheme.)

Dynamic values that are to be inserted into the constructed query require careful handling since they might themselves contain quote characters. An example (this assumes that you are using dollar quoting for the function as a whole, so the quote marks need not be doubled):

```
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 labelled `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`. 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);
```

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 more efficient, because the parameters `newvalue` and `keyvalue` are not converted to text.

A much larger example of a dynamic command and `EXECUTE` can be seen in Example 39-8, which builds and executes a `CREATE FUNCTION` command to define a new function.

39.5.5. 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 DIAGNOSTICS variable = item [ , ... ];
```

This command allows retrieval of system status indicators. Each *`item`* is a key word identifying a state 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 `ROW_COUNT`, the number of rows processed by the last SQL command sent to the SQL engine, and `RESULT_OID`, the OID of the last row inserted by the most recent SQL command. Note that `RESULT_OID` is only useful after an `INSERT` command into a table containing OIDs.

An example:

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

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.

39.5.6. 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.

39.6. Control Structures

Control structures are probably the most useful (and important) part of PL/pgSQL. With PL/pgSQL's control structures, you can manipulate PostgreSQL data in a very flexible and powerful way.

39.6.1. Returning From a Function

There are two commands available that allow you to return data from a function: `RETURN` and `RETURN NEXT`.

39.6.1.1. `RETURN`

```
RETURN expression;
```

`RETURN` with an expression terminates the function and returns the value of *`expression`* to the caller. This form is used for PL/pgSQL functions that do not return a set.

When returning a scalar type, any expression can be used. The expression's result will be automatically cast into the function's return type as described for assignments. To return a composite (row) value, you must write a record or row variable as the *`expression`*.

If you declared the function with output parameters, write just `RETURN` with no expression. The current values of the output parameter variables will be returned.

If you declared the function to return `void`, a `RETURN` statement can be used to exit the function early; but do not write an expression following `RETURN`.

The return value of a function cannot be left undefined. If control reaches the end of the top-level block of the function without hitting a `RETURN` statement, a run-time error will occur. This restriction does not apply to functions with output parameters and functions returning `void`, however. In those cases a `RETURN` statement is automatically executed if the top-level block finishes.

39.6.1.2. `RETURN NEXT` and `RETURN QUERY`

```
RETURN NEXT expression;
RETURN QUERY query;
RETURN QUERY EXECUTE command-string [ USING expression [, ... ] ];
```

When a PL/pgSQL function is declared to return `SETOF` *`sometype`*, the procedure to follow is slightly different. In that case, the individual items to return are specified by a sequence of `RETURN NEXT` or `RETURN QUERY` commands, and then a final `RETURN` command with no argument is used to indicate that the function has finished executing. `RETURN NEXT` can be used with both scalar and composite data types; with a composite result type, an entire "table" of results will be returned. `RETURN QUERY` appends the results of executing a query to the function's result set. `RETURN NEXT` and `RETURN QUERY` can be freely intermixed in a single set-returning function, in which case their results will be concatenated.

`RETURN NEXT` and `RETURN QUERY` do not actually return from the function — they simply append zero or more rows to the function's result set. Execution then continues with the next statement in the PL/pgSQL function. As successive `RETURN NEXT` or `RETURN QUERY` commands are executed, the result set is built up. A final `RETURN`, which should have no argument, causes control to exit the function (or you can just let control reach the end of the function).

`RETURN QUERY` has a variant `RETURN QUERY EXECUTE`, which specifies the query to be executed dynamically. Parameter expressions can be inserted into the computed query string via `USING`, in just the same way as in the `EXECUTE` command.

If you declared the function with output parameters, write just `RETURN NEXT` with no expression. On each execution, the current values of the output parameter variable(s) will be saved for eventual return as a row of the result. Note that you must declare the function as returning `SETOF record` when there are multiple output parameters, or `SETOF` *`sometype`* when there is just one output parameter of type *`sometype`*, in order to create a set-returning function with output parameters.

Here is an example of a function using `RETURN NEXT`:

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
INSERT INTO foo VALUES (1, 2, 'three');
INSERT INTO foo VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION getAllFoo() RETURNS SETOF foo AS
$BODY$
DECLARE
    r foo%rowtype;
BEGIN
    FOR r IN SELECT * FROM foo
    WHERE fooid > 0
    LOOP
        -- can do some processing here
        RETURN NEXT r; -- return current row of SELECT
    END LOOP;
    RETURN;
END
$BODY$
LANGUAGE 'plpgsql' ;

SELECT * FROM getallfoo();
```

> **Note:** The current implementation of `RETURN NEXT` and `RETURN QUERY` stores the entire result set before returning from the function, as discussed above. That means that if a PL/pgSQL function produces a very large result set, performance might be poor: data will be written to disk to avoid memory exhaustion, but the function itself will not return until the entire result set has been generated. A future version of PL/pgSQL might allow users to define set-returning functions that do not have this limitation. Currently, the point at which data begins being written to disk is controlled by the work_mem configuration variable. Administrators who have sufficient memory to store larger result sets in memory should consider increasing this parameter.

39.6.2. Conditionals

`IF` and `CASE` statements let you execute alternative commands based on certain conditions. PL/pgSQL has three forms of `IF`:

- `IF ... THEN`
- `IF ... THEN ... ELSE`
- `IF ... THEN ... ELSIF ... THEN ... ELSE`

and two forms of `CASE`:

- `CASE ... WHEN ... THEN ... ELSE ... END CASE`
- `CASE WHEN ... THEN ... ELSE ... END CASE`

39.6.2.1. `IF-THEN`

```
IF boolean-expression THEN
    statements
END IF;
```

`IF-THEN` statements are the simplest form of `IF`. The statements between `THEN` and `END IF` will be executed if the condition is true. Otherwise, they are skipped.

Example:

```
IF v_user_id <> 0 THEN
    UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

39.6.2.2. `IF-THEN-ELSE`

```
IF boolean-expression THEN
    statements
ELSE
    statements
END IF;
```

`IF-THEN-ELSE` statements add to `IF-THEN` by letting you specify an alternative set of statements that should be executed if the condition is not true. (Note this includes the case where the condition evaluates to NULL.)

Examples:

```
IF parentid IS NULL OR parentid = ''
THEN
    RETURN fullname;
ELSE
    RETURN hp_true_filename(parentid) || '/' || fullname;
END IF;
```

```
IF v_count > 0 THEN
    INSERT INTO users_count (count) VALUES (v_count);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;
```

39.6.2.3. `IF-THEN-ELSIF`

```
IF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
    ...]]
[ ELSE
    statements ]
END IF;
```

Sometimes there are more than just two alternatives. `IF-THEN-ELSIF` provides a convenient method of checking several alternatives in turn. The `IF` conditions are tested successively until the first one that is true is found. Then the associated statement(s) are executed, after which control passes to the next statement after `END IF`. (Any subsequent `IF` conditions are *not* tested.) If none of the `IF` conditions is true, then the `ELSE` block (if any) is executed.

Here is an example:

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    -- hmm, the only other possibility is that number is null
    result := 'NULL';
END IF;
```

The key word `ELSIF` can also be spelled `ELSEIF`.

An alternative way of accomplishing the same task is to nest `IF-THEN-ELSE` statements, as in the following example:

```
IF demo_row.sex = 'm' THEN
    pretty_sex := 'man';
ELSE
    IF demo_row.sex = 'f' THEN
        pretty_sex := 'woman';
    END IF;
END IF;
```

However, this method requires writing a matching `END IF` for each `IF`, so it is much more cumbersome than using `ELSIF` when there are many alternatives.

39.6.2.4. Simple `CASE`

```
CASE search-expression
    WHEN expression [, expression [ ... ]] THEN
      statements
  [ WHEN expression [, expression [ ... ]] THEN
      statements
    ... ]
  [ ELSE
      statements ]
END CASE;
```

The simple form of `CASE` provides conditional execution based on equality of operands. The *`search-expression`* is evaluated (once) and successively compared to each *`expression`* in the `WHEN` clauses. If a match is found, then the corresponding *`statements`* are executed, and then control passes to the next statement after `END CASE`. (Subsequent `WHEN` expressions are not evaluated.) If no match is found, the `ELSE` *`statements`* are executed; but if `ELSE` is not present, then a `CASE_NOT_FOUND` exception is raised.

Here is a simple example:

```
CASE x
    WHEN 1, 2 THEN
        msg := 'one or two';
    ELSE
        msg := 'other value than one or two';
END CASE;
```

39.6.2.5. Searched `CASE`

```
CASE
    WHEN boolean-expression THEN
      statements
  [ WHEN boolean-expression THEN
      statements
    ... ]
  [ ELSE
      statements ]
END CASE;
```

The searched form of `CASE` provides conditional execution based on truth of Boolean expressions. Each `WHEN` clause's *`boolean-expression`* is evaluated in turn, until one is found that yields `true`. Then the corresponding *`statements`* are executed, and then control passes to the next statement after `END CASE`. (Subsequent `WHEN` expressions are not evaluated.) If no true result is found, the `ELSE` *`statements`* are executed; but if `ELSE` is not present, then a `CASE_NOT_FOUND` exception is raised.

Here is an example:

```
CASE
```

```
    WHEN x BETWEEN 0 AND 10 THEN
        msg := 'value is between zero and ten';
    WHEN x BETWEEN 11 AND 20 THEN
        msg := 'value is between eleven and twenty';
END CASE;
```

This form of `CASE` is entirely equivalent to `IF-THEN-ELSIF`, except for the rule that reaching an omitted `ELSE` clause results in an error rather than doing nothing.

39.6.3. Simple Loops

With the `LOOP`, `EXIT`, `CONTINUE`, `WHILE`, `FOR`, and `FOREACH` statements, you can arrange for your PL/pgSQL function to repeat a series of commands.

39.6.3.1. LOOP

```
[ <<label>> ]
LOOP
    statements
END LOOP [ label ];
```

`LOOP` defines an unconditional loop that is repeated indefinitely until terminated by an `EXIT` or `RETURN` statement. The optional *`label`* can be used by `EXIT` and `CONTINUE` statements within nested loops to specify which loop those statements refer to.

39.6.3.2. EXIT

```
EXIT [ label ] [ WHEN boolean-expression ];
```

If no *`label`* is given, the innermost loop is terminated and the statement following `END LOOP` is executed next. If *`label`* is given, it must be the label of the current or some outer level of nested loop or block. Then the named loop or block is terminated and control continues with the statement after the loop's/block's corresponding `END`.

If `WHEN` is specified, the loop exit occurs only if *`boolean-expression`* is true. Otherwise, control passes to the statement after `EXIT`.

`EXIT` can be used with all types of loops; it is not limited to use with unconditional loops.

When used with a `BEGIN` block, `EXIT` passes control to the next statement after the end of the block. Note that a label must be used for this purpose; an unlabelled `EXIT` is never considered to match a `BEGIN` block. (This is a change from pre-8.4 releases of PostgreSQL, which would allow an unlabelled `EXIT` to match a `BEGIN` block.)

Examples:

```
LOOP
    -- some computations
    IF count > 0 THEN
        EXIT;  -- exit loop
    END IF;
END LOOP;
```

```
LOOP
    -- some computations
    EXIT WHEN count > 0;  -- same result as previous example
END LOOP;

<<ablock>>
BEGIN
    -- some computations
    IF stocks > 100000 THEN
        EXIT ablock;  -- causes exit from the BEGIN block
    END IF;
    -- computations here will be skipped when stocks > 100000
END;
```

39.6.3.3. `CONTINUE`

```
CONTINUE [ label ] [ WHEN boolean-expression ];
```

If no *`label`* is given, the next iteration of the innermost loop is begun. That is, all statements remaining in the loop body are skipped, and control returns to the loop control expression (if any) to determine whether another loop iteration is needed. If *`label`* is present, it specifies the label of the loop whose execution will be continued.

If `WHEN` is specified, the next iteration of the loop is begun only if *`boolean-expression`* is true. Otherwise, control passes to the statement after `CONTINUE`.

`CONTINUE` can be used with all types of loops; it is not limited to use with unconditional loops.

Examples:

```
LOOP
    -- some computations
    EXIT WHEN count > 100;
    CONTINUE WHEN count < 50;
    -- some computations for count IN [50 .. 100]
END LOOP;
```

39.6.3.4. `WHILE`

```
[ <<label>> ]
WHILE boolean-expression LOOP
    statements
END LOOP [ label ];
```

The `WHILE` statement repeats a sequence of statements so long as the *`boolean-expression`* evaluates to true. The expression is checked just before each entry to the loop body.

For example:

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
    -- some computations here
```

```
END LOOP;

WHILE NOT done LOOP
    -- some computations here
END LOOP;
```

39.6.3.5. `FOR` (Integer Variant)

```
[ <<label>> ]
FOR name IN [ REVERSE ] expression .. expression [ BY expression ] LOOP
    statements
END LOOP [ label ];
```

This form of `FOR` creates a loop that iterates over a range of integer values. The variable *`name`* is automatically defined as type `integer` and exists only inside the loop (any existing definition of the variable name is ignored within the loop). The two expressions giving the lower and upper bound of the range are evaluated once when entering the loop. If the `BY` clause isn't specified the iteration step is 1, otherwise it's the value specified in the `BY` clause, which again is evaluated once on loop entry. If `REVERSE` is specified then the step value is subtracted, rather than added, after each iteration.

Some examples of integer `FOR` loops:

```
FOR i IN 1..10 LOOP
    -- i will take on the values 1,2,3,4,5,6,7,8,9,10 within the loop
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- i will take on the values 10,9,8,7,6,5,4,3,2,1 within the loop
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
    -- i will take on the values 10,8,6,4,2 within the loop
END LOOP;
```

If the lower bound is greater than the upper bound (or less than, in the `REVERSE` case), the loop body is not executed at all. No error is raised.

If a *`label`* is attached to the `FOR` loop then the integer loop variable can be referenced with a qualified name, using that *`label`*.

39.6.4. Looping Through Query Results

Using a different type of `FOR` loop, you can iterate through the results of a query and manipulate that data accordingly. The syntax is:

```
[ <<label>> ]
FOR target IN query LOOP
    statements
END LOOP [ label ];
```

The *target* is a record variable, row variable, or comma-separated list of scalar variables. The *target* is successively assigned each row resulting from the *query* and the loop body is executed for each row. Here is an example:

```
CREATE FUNCTION cs_refresh_mviews() RETURNS integer AS $$
DECLARE
    mviews RECORD;
BEGIN
    RAISE NOTICE 'Refreshing materialized views...';

    FOR mviews IN SELECT * FROM cs_materialized_views ORDER BY sort_key LOOP

        -- Now "mviews" has one record from cs_materialized_views

        RAISE NOTICE 'Refreshing materialized view %s ...', quote_ident(mviews.mv_name);
        EXECUTE 'TRUNCATE TABLE ' || quote_ident(mviews.mv_name);
        EXECUTE 'INSERT INTO '
                   || quote_ident(mviews.mv_name) || ' '
                   || mviews.mv_query;
    END LOOP;

    RAISE NOTICE 'Done refreshing materialized views.';
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

If the loop is terminated by an `EXIT` statement, the last assigned row value is still accessible after the loop.

The *query* used in this type of `FOR` statement can be any SQL command that returns rows to the caller: `SELECT` is the most common case, but you can also use `INSERT`, `UPDATE`, or `DELETE` with a `RETURNING` clause. Some utility commands such as `EXPLAIN` will work too.

PL/pgSQL variables are substituted into the query text, and the query plan is cached for possible re-use, as discussed in detail in Section 39.10.1 and Section 39.10.2.

The `FOR-IN-EXECUTE` statement is another way to iterate over rows:

```
[ <<label>> ]
FOR target IN EXECUTE text_expression [ USING expression [, ... ] ] LOOP
    statements
END LOOP [ label ];
```

This is like the previous form, except that the source query is specified as a string expression, which is evaluated and replanned on each entry to the `FOR` loop. This allows the programmer to choose the speed of a preplanned query or the flexibility of a dynamic query, just as with a plain `EXECUTE` statement. As with `EXECUTE`, parameter values can be inserted into the dynamic command via `USING`.

Another way to specify the query whose results should be iterated through is to declare it as a cursor. This is described in Section 39.7.4.

39.6.5. Looping Through Arrays

The `FOREACH` loop is much like a `FOR` loop, but instead of iterating through the rows returned by a SQL query, it iterates through the elements of an array value. (In general, `FOREACH` is meant for looping through components of a composite-valued expression; variants for looping through composites besides arrays may be added in future.) The `FOREACH` statement to loop over an array is:

```
[ <<label>> ]
FOREACH target [ SLICE number ] IN ARRAY expression LOOP
    statements
END LOOP [ label ];
```

Without `SLICE`, or if `SLICE 0` is specified, the loop iterates through individual elements of the array produced by evaluating the *`expression`*. The *`target`* variable is assigned each element value in sequence, and the loop body is executed for each element. Here is an example of looping through the elements of an integer array:

```
CREATE FUNCTION sum(int[]) RETURNS int8 AS $$
DECLARE
  s int8 := 0;
  x int;
BEGIN
  FOREACH x IN ARRAY $1
  LOOP
    s := s + x;
  END LOOP;
  RETURN s;
END;
$$ LANGUAGE plpgsql;
```

The elements are visited in storage order, regardless of the number of array dimensions. Although the *`target`* is usually just a single variable, it can be a list of variables when looping through an array of composite values (records). In that case, for each array element, the variables are assigned from successive columns of the composite value.

With a positive `SLICE` value, `FOREACH` iterates through slices of the array rather than single elements. The `SLICE` value must be an integer constant not larger than the number of dimensions of the array. The *`target`* variable must be an array, and it receives successive slices of the array value, where each slice is of the number of dimensions specified by `SLICE`. Here is an example of iterating through one-dimensional slices:

```
CREATE FUNCTION scan_rows(int[]) RETURNS void AS $$
DECLARE
  x int[];
BEGIN
  FOREACH x SLICE 1 IN ARRAY $1
  LOOP
    RAISE NOTICE 'row = %', x;
  END LOOP;
END;
$$ LANGUAGE plpgsql;

SELECT scan_rows(ARRAY[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]);

NOTICE:  row = {1,2,3}
NOTICE:  row = {4,5,6}
NOTICE:  row = {7,8,9}
NOTICE:  row = {10,11,12}
```

39.6.6. Trapping Errors

By default, any error occurring in a PL/pgSQL function aborts execution of the function, and indeed of the surrounding transaction as well. You can trap errors and recover from them by using a `BEGIN` block with an `EXCEPTION` clause. The syntax is an extension of the normal syntax for a `BEGIN` block:

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
EXCEPTION
    WHEN condition [ OR condition ... ] THEN
        handler_statements
    [ WHEN condition [ OR condition ... ] THEN
          handler_statements
      ... ]
END;
```

If no error occurs, this form of block simply executes all the *statements*, and then control passes to the next statement after `END`. But if an error occurs within the *statements*, further processing of the *statements* is abandoned, and control passes to the `EXCEPTION` list. The list is searched for the first *condition* matching the error that occurred. If a match is found, the corresponding *handler_statements* are executed, and then control passes to the next statement after `END`. If no match is found, the error propagates out as though the `EXCEPTION` clause were not there at all: the error can be caught by an enclosing block with `EXCEPTION`, or if there is none it aborts processing of the function.

The *condition* names can be any of those shown in Appendix A. A category name matches any error within its category. The special condition name `OTHERS` matches every error type except `QUERY_CANCELED`. (It is possible, but often unwise, to trap `QUERY_CANCELED` by name.) Condition names are not case-sensitive. Also, an error condition can be specified by `SQLSTATE` code; for example these are equivalent:

```
WHEN division_by_zero THEN ...
WHEN SQLSTATE '22012' THEN ...
```

If a new error occurs within the selected *handler_statements*, it cannot be caught by this `EXCEPTION` clause, but is propagated out. A surrounding `EXCEPTION` clause could catch it.

When an error is caught by an `EXCEPTION` clause, the local variables of the PL/pgSQL function remain as they were when the error occurred, but all changes to persistent database state within the block are rolled back. As an example, consider this fragment:

```
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
BEGIN
    UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
    x := x + 1;
    y := x / 0;
EXCEPTION
```

```
    WHEN division_by_zero THEN
        RAISE NOTICE 'caught division_by_zero';
        RETURN x;
END;
```

When control reaches the assignment to `y`, it will fail with a `division_by_zero` error. This will be caught by the `EXCEPTION` clause. The value returned in the `RETURN` statement will be the incremented value of `x`, but the effects of the `UPDATE` command will have been rolled back. The `INSERT` command preceding the block is not rolled back, however, so the end result is that the database contains `Tom Jones` not `Joe Jones`.

> **Tip:** A block containing an `EXCEPTION` clause is significantly more expensive to enter and exit than a block without one. Therefore, don't use `EXCEPTION` without need.

Within an exception handler, the `SQLSTATE` variable contains the error code that corresponds to the exception that was raised (refer to Table A-1 for a list of possible error codes). The `SQLERRM` variable contains the error message associated with the exception. These variables are undefined outside exception handlers.

Example 39-2. Exceptions with `UPDATE`/`INSERT`

This example uses exception handling to perform either `UPDATE` or `INSERT`, as appropriate:

```
CREATE TABLE db (a INT PRIMARY KEY, b TEXT);

CREATE FUNCTION merge_db(key INT, data TEXT) RETURNS VOID AS
$$
BEGIN
    LOOP
        -- first try to update the key
        UPDATE db SET b = data WHERE a = key;
        IF found THEN
            RETURN;
        END IF;
        -- not there, so try to insert the key
        -- if someone else inserts the same key concurrently,
        -- we could get a unique-key failure
        BEGIN
            INSERT INTO db(a,b) VALUES (key, data);
            RETURN;
        EXCEPTION WHEN unique_violation THEN
            -- Do nothing, and loop to try the UPDATE again.
        END;
    END LOOP;
END;
$$
LANGUAGE plpgsql;

SELECT merge_db(1, 'david');
SELECT merge_db(1, 'dennis');
```

This example assumes the `unique_violation` error is caused by the `INSERT`, and not by an `INSERT` trigger function on the table.

39.7. Cursors

Rather than executing a whole query at once, it is possible to set up a *cursor* that encapsulates the query, and then read the query result a few rows at a time. One reason for doing this is to avoid memory overrun when the result contains a large number of rows. (However, PL/pgSQL users do not normally need to worry about that, since `FOR` loops automatically use a cursor internally to avoid memory problems.) A more interesting usage is to return a reference to a cursor that a function has created, allowing the caller to read the rows. This provides an efficient way to return large row sets from functions.

39.7.1. 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, which in general is:

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

(`FOR` can be replaced by `IS` for Oracle compatibility.) 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.

Some examples:

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) IS SELECT * FROM tenk1 WHERE unique1 = key;
```

All three of these variables have the data type `refcursor`, but the first can be used with any query, while the second has a fully specified query already *bound* to it, and the last has a parameterized query bound to it. (`key` will be replaced by an integer parameter value when the cursor is opened.) The variable `curs1` is said to be *unbound* since it is not bound to any particular query.

39.7.2. 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.

> **Note:** Bound cursor variables can also be used without explicitly opening the cursor, via the `FOR` statement described in Section 39.7.4.

39.7.2.1. `OPEN FOR` *`query`*

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] 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. The `SCROLL` and `NO SCROLL` options have the same meanings as for a bound cursor.

An example:

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

39.7.2.2. `OPEN FOR EXECUTE`

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

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 (see Section 39.10.2), 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 `USING`. The `SCROLL` and `NO SCROLL` options have the same meanings as for a bound cursor.

An example:

```
OPEN curs1 FOR EXECUTE 'SELECT * FROM ' || quote_ident(tabname)
                                        || ' WHERE col1 = $1' USING keyvalue;
```

In this example, the table name is inserted into the query textually, so use of `quote_ident()` is recommended to guard against SQL injection. The comparison value for `col1` is inserted via a `USING` parameter, so it needs no quoting.

39.7.2.3. Opening a Bound Cursor

```
OPEN bound_cursorvar [ ( argument_values ) ];
```

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. Notice that `SCROLL` and `NO SCROLL` cannot be specified, as the cursor's scrolling behavior was already determined.

Note that because variable substitution is done on the bound cursor's query, there are two ways to pass values into the cursor: either with an explicit argument to `OPEN`, or implicitly by referencing a PL/pgSQL variable in the query. However, only variables declared before the bound cursor was declared will be substituted into it. In either case the value to be passed is determined at the time of the `OPEN`.

Examples:

```
OPEN curs2;
OPEN curs3(42);
```

39.7.3. Using Cursors

Once a cursor has been opened, it can be manipulated with the statements described here.

These manipulations need not occur in the same function that opened the cursor to begin with. You can return a `refcursor` value out of a function and let the caller operate on the cursor. (Internally, a `refcursor` value is simply the string name of a so-called portal containing the active query for the cursor. This name can be passed around, assigned to other `refcursor` variables, and so on, without disturbing the portal.)

All portals are implicitly closed at transaction end. Therefore a `refcursor` value is usable to reference an open cursor only until the end of the transaction.

39.7.3.1. FETCH

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

`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.

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`. *`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.

Examples:

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;
```

39.7.3.2. MOVE

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

`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.

The `direction` clause can be any of the variants allowed in the SQL FETCH command, namely `NEXT`, `PRIOR`, `FIRST`, `LAST`, `ABSOLUTE` *count*, `RELATIVE` *count*, `ALL`, `FORWARD` [*count* | `ALL`], or `BACKWARD` [*count* | `ALL`]. Omitting *direction* is the same as specifying `NEXT`. *direction* values that require moving backward are likely to fail unless the cursor was declared or opened with the `SCROLL` option.

Examples:

```
MOVE curs1;
MOVE LAST FROM curs3;
MOVE RELATIVE -2 FROM curs4;
MOVE FORWARD 2 FROM curs4;
```

39.7.3.3. UPDATE/DELETE WHERE CURRENT OF

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

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 (in particular, no grouping) and it's best to use `FOR UPDATE` in the cursor. For more information see the DECLARE reference page.

An example:

```
UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;
```

39.7.3.4. CLOSE

```
CLOSE cursor;
```

`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.

An example:

```
CLOSE curs1;
```

39.7.3.5. 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.

> **Note:** A bound cursor variable is initialized to the string value representing its name, so that the portal name is the same as the cursor variable name, unless the programmer overrides it by assignment before opening the cursor. But an unbound cursor variable defaults to the null value initially, so it will receive an automatically-generated unique name, unless overridden.

The following example shows one way a cursor name can be supplied by the caller:

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

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;
```

The following example uses automatic cursor name generation:

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM test;
    RETURN ref;
END;
' LANGUAGE plpgsql;

-- need to be in a transaction to use cursors.
BEGIN;
SELECT reffunc2();

      reffunc2
--------------------
 <unnamed cursor 1>
(1 row)

FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

The following example shows one way to return multiple cursors from a single function:

```
CREATE FUNCTION myfunc(refcursor, refcursor) RETURNS SETOF refcursor AS $$
BEGIN
```

```
    OPEN $1 FOR SELECT * FROM table_1;
    RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM table_2;
    RETURN NEXT $2;
END;
$$ LANGUAGE plpgsql;

-- need to be in a transaction to use cursors.
BEGIN;

SELECT * FROM myfunc('a', 'b');

FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

39.7.4. Looping Through a Cursor's Result

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

```
[ <<label>> ]
FOR recordvar IN bound_cursorvar [ ( argument_values ) ] 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.

39.8. Errors and Messages

Use the `RAISE` statement to report messages and raise errors.

```
RAISE [ level ] 'format' [, expression [, ... ]] [ USING option = expression [, ... ] ];
RAISE [ level ] condition_name [ USING option = expression [, ... ] ];
RAISE [ level ] SQLSTATE 'sqlstate' [ USING option = expression [, ... ] ];
RAISE [ level ] USING option = expression [, ... ];
RAISE ;
```

The *`level`* option specifies the error severity. Allowed levels are `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`, and `EXCEPTION`, with `EXCEPTION` being the default. `EXCEPTION` raises an error (which normally aborts the current transaction); the other levels only generate messages of different priority levels. Whether messages of a particular priority are reported to the client, written to the server log, or both is controlled by the log_min_messages and client_min_messages configuration variables. See Chapter 18 for more information.

After `level` if any, you can write a `format` (which must be a simple string literal, not an expression). The format string specifies the error message text to be reported. The format string can be followed by optional argument expressions to be inserted into the message. Inside the format string, `%` is replaced by the string representation of the next optional argument's value. Write `%%` to emit a literal `%`.

In this example, the value of `v_job_id` will replace the `%` in the string:

```
RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;
```

You can attach additional information to the error report by writing `USING` followed by `option` = `expression` items. The allowed `option` keywords are `MESSAGE`, `DETAIL`, `HINT`, and `ERRCODE`, while each `expression` can be any string-valued expression. `MESSAGE` sets the error message text (this option can't be used in the form of `RAISE` that includes a format string before `USING`). `DETAIL` supplies an error detail message, while `HINT` supplies a hint message. `ERRCODE` specifies the error code (SQLSTATE) to report, either by condition name as shown in Appendix A, or directly as a five-character SQLSTATE code.

This example will abort the transaction with the given error message and hint:

```
RAISE EXCEPTION 'Nonexistent ID --> %', user_id
      USING HINT = 'Please check your user ID';
```

These two examples show equivalent ways of setting the SQLSTATE:

```
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = 'unique_violation';
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';
```

There is a second `RAISE` syntax in which the main argument is the condition name or SQLSTATE to be reported, for example:

```
RAISE division_by_zero;
RAISE SQLSTATE '22012';
```

In this syntax, `USING` can be used to supply a custom error message, detail, or hint. Another way to do the earlier example is

```
RAISE unique_violation USING MESSAGE = 'Duplicate user ID: ' || user_id;
```

Still another variant is to write `RAISE USING` or `RAISE level USING` and put everything else into the `USING` list.

The last variant of `RAISE` has no parameters at all. This form can only be used inside a `BEGIN` block's `EXCEPTION` clause; it causes the error currently being handled to be re-thrown.

> **Note:** Before PostgreSQL 9.1, `RAISE` without parameters was interpreted as re-throwing the error from the block containing the active exception handler. Thus an `EXCEPTION` clause nested within that handler could not catch it, even if the `RAISE` was within the nested `EXCEPTION` clause's block. This was deemed surprising as well as being incompatible with Oracle's PL/SQL.

If no condition name nor SQLSTATE is specified in a `RAISE EXCEPTION` command, the default is to use `RAISE_EXCEPTION` (`P0001`). If no message text is specified, the default is to use the condition name or SQLSTATE as message text.

> **Note:** When specifying an error code by SQLSTATE code, you are not limited to the predefined error codes, but can select any error code consisting of five digits and/or upper-case ASCII letters, other than `00000`. It is recommended that you avoid throwing error codes that end in three zeroes, because these are category codes and can only be trapped by trapping the whole category.

39.9. Trigger Procedures

PL/pgSQL can be used to define trigger procedures. A trigger procedure is created with the `CREATE FUNCTION` command, declaring it 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 arguments specified in `CREATE TRIGGER` — trigger arguments are passed via `TG_ARGV`, as described below.

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 procedure 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.

Row-level triggers fired `BEFORE` can return null to signal the trigger manager to skip the rest of the operation for this row (i.e., subsequent triggers are not fired, and the `INSERT`/`UPDATE`/`DELETE` does not occur for this row). If a nonnull value is returned then the operation proceeds with that row value. Returning a row value different from the original value of `NEW` alters the row that will be inserted or updated. Thus, if the trigger function wants the triggering action to succeed normally without altering the row value, `NEW` (or a value equal thereto) has to be returned. To alter the row to be stored, it is possible to replace single values directly in `NEW` and return the modified `NEW`, or to build a complete new record/row to return. In the case of a before-trigger on `DELETE`, the returned value has no direct effect, but it has to be nonnull to allow the trigger action to proceed. Note that `NEW` is null in `DELETE` triggers, so returning that is usually not sensible. The usual idiom in `DELETE` triggers is to return `OLD`.

`INSTEAD OF` triggers (which are always row-level triggers, and may only be used on views) can return null to signal that they did not perform any updates, and that the rest of the operation for this row should be skipped (i.e., subsequent triggers are not fired, and the row is not counted in the rows-affected status for the surrounding `INSERT`/`UPDATE`/`DELETE`). Otherwise a nonnull value should be returned, to signal that the trigger performed the requested operation. For `INSERT` and `UPDATE` operations, the return value should be `NEW`, which the trigger function may modify to support `INSERT RETURNING` and `UPDATE RETURNING` (this will also affect the row value passed to any subsequent triggers). For `DELETE` operations, the return value should be `OLD`.

The return value of a row-level trigger fired `AFTER` or a statement-level trigger fired `BEFORE` or `AFTER` is always ignored; it might as well be null. However, any of these types of triggers might still abort the entire operation by raising an error.

Example 39-3 shows an example of a trigger procedure in PL/pgSQL.

Example 39-3. A PL/pgSQL Trigger Procedure

This example trigger ensures that any time a row is inserted or updated in the table, the current user name and time are stamped into the row. And it checks that an employee's name is given and that the salary is a positive value.

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);
```

```
CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN
        -- Check that empname and salary are given
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION 'empname cannot be null';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION '% cannot have null salary', NEW.empname;
        END IF;

        -- Who works for us when she must pay for it?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
        END IF;

        -- Remember who changed the payroll when
        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
$emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

Another way to log changes to a table involves creating a new table that holds a row for each insert, update, or delete that occurs. This approach can be thought of as auditing changes to a table. Example 39-4 shows an example of an audit trigger procedure in PL/pgSQL.

Example 39-4. A PL/pgSQL Trigger Procedure For Auditing

This example 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,
        -- make 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.*;
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
            RETURN 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 PROCEDURE process_emp_audit();
```

A variation of the previous example uses a view joining the main table to the audit table, to show when each entry was last modified. This approach still records the full audit trail of changes to the table, but also presents a simplified view of the audit trail, showing just the last modified timestamp derived from the audit trail for each entry. Example 39-5 shows an example of an audit trigger on a view in PL/pgSQL.

Example 39-5. A PL/pgSQL View Trigger Procedure For Auditing

This example uses a trigger on the view to make it updatable, and ensure that any insert, update or delete of a row in the view is recorded (i.e., audited) in the `emp_audit` table. The current time and user name are recorded, together with the type of operation performed, and the view displays the last modified time of each row.

```
CREATE TABLE emp (
    empname           text PRIMARY KEY,
    salary            integer
);

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

CREATE VIEW emp_view AS
    SELECT e.empname,
           e.salary,
           max(ea.stamp) AS last_updated
      FROM emp e
      LEFT JOIN emp_audit ea ON ea.empname = e.empname
     GROUP BY 1, 2;

CREATE OR REPLACE FUNCTION update_emp_view() RETURNS TRIGGER AS $$
    BEGIN
        --
        -- Perform the required operation on emp, and create a row in emp_audit
        -- to reflect the change made to emp.
        --
        IF (TG_OP = 'DELETE') THEN
```

```
            DELETE FROM emp WHERE empname = OLD.empname;
            IF NOT FOUND THEN RETURN NULL; END IF;

            OLD.last_updated = now();
            INSERT INTO emp_audit VALUES('D', user, OLD.*);
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            UPDATE emp SET salary = NEW.salary WHERE empname = OLD.empname;
            IF NOT FOUND THEN RETURN NULL; END IF;

            NEW.last_updated = now();
            INSERT INTO emp_audit VALUES('U', user, NEW.*);
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp VALUES(NEW.empname, NEW.salary);

            NEW.last_updated = now();
            INSERT INTO emp_audit VALUES('I', user, NEW.*);
            RETURN NEW;
        END IF;
    END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_view
    FOR EACH ROW EXECUTE PROCEDURE update_emp_view();
```

One use of triggers is to maintain a summary table of another table. The resulting summary can be used in place of the original table for certain queries — often with vastly reduced run times. This technique is commonly used in Data Warehousing, where the tables of measured or observed data (called fact tables) might be extremely large. Example 39-6 shows an example of a trigger procedure in PL/pgSQL that maintains a summary table for a fact table in a data warehouse.

Example 39-6. A PL/pgSQL Trigger Procedure For Maintaining A Summary Table

The schema detailed here is partly based on the *Grocery Store* example from *The Data Warehouse Toolkit* by Ralph Kimball.

```
--
-- Main tables - time dimension and sales fact.
--
CREATE TABLE time_dimension (
    time_key                    integer NOT NULL,
    day_of_week                 integer NOT NULL,
    day_of_month                integer NOT NULL,
    month                       integer NOT NULL,
    quarter                     integer NOT NULL,
    year                        integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (
    time_key                    integer NOT NULL,
    product_key                 integer NOT NULL,
    store_key                   integer NOT NULL,
    amount_sold                 numeric(12,2) NOT NULL,
    units_sold                  integer NOT NULL,
    amount_cost                 numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);

--
-- Summary table - sales by time.
--
CREATE TABLE sales_summary_bytime (
    time_key                    integer NOT NULL,
    amount_sold                 numeric(15,2) NOT NULL,
    units_sold                  numeric(12) NOT NULL,
    amount_cost                 numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON sales_summary_bytime(time_key);

--
-- Function and trigger to amend summarized column(s) on UPDATE, INSERT, DELETE.
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER
AS $maint_sales_summary_bytime$
    DECLARE
        delta_time_key          integer;
        delta_amount_sold       numeric(15,2);
        delta_units_sold        numeric(12);
        delta_amount_cost       numeric(15,2);
    BEGIN

        -- Work out the increment/decrement amount(s).
        IF (TG_OP = 'DELETE') THEN

            delta_time_key = OLD.time_key;
            delta_amount_sold = -1 * OLD.amount_sold;
            delta_units_sold = -1 * OLD.units_sold;
            delta_amount_cost = -1 * OLD.amount_cost;

        ELSIF (TG_OP = 'UPDATE') THEN

            -- forbid updates that change the time_key -
            -- (probably not too onerous, as DELETE + INSERT is how most
            -- changes will be made).
            IF ( OLD.time_key != NEW.time_key) THEN
                RAISE EXCEPTION 'Update of time_key : % -> % not allowed',
                                                    OLD.time_key, NEW.time_key;
            END IF;

            delta_time_key = OLD.time_key;
            delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
            delta_units_sold = NEW.units_sold - OLD.units_sold;
            delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

        ELSIF (TG_OP = 'INSERT') THEN

            delta_time_key = NEW.time_key;
            delta_amount_sold = NEW.amount_sold;
            delta_units_sold = NEW.units_sold;
            delta_amount_cost = NEW.amount_cost;
        END IF;


        -- Insert or update the summary row with the new values.
        <<insert_update>>
        LOOP
            UPDATE sales_summary_bytime
                SET amount_sold = amount_sold + delta_amount_sold,
                    units_sold = units_sold + delta_units_sold,
                    amount_cost = amount_cost + delta_amount_cost
                WHERE time_key = delta_time_key;

            EXIT insert_update WHEN found;

            BEGIN
                INSERT INTO sales_summary_bytime (
                            time_key,
                            amount_sold,
                            units_sold,
                            amount_cost)
                    VALUES (
                            delta_time_key,
                            delta_amount_sold,
                            delta_units_sold,
                            delta_amount_cost
                           );

                EXIT insert_update;

            EXCEPTION
                WHEN UNIQUE_VIOLATION THEN
                    -- do nothing
            END;
        END LOOP insert_update;

        RETURN NULL;

    END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
AFTER INSERT OR UPDATE OR DELETE ON sales_fact
    FOR EACH ROW EXECUTE PROCEDURE maint_sales_summary_bytime();

INSERT INTO sales_fact VALUES(1,1,1,10,3,15);
INSERT INTO sales_fact VALUES(1,2,1,20,5,35);
INSERT INTO sales_fact VALUES(2,2,1,40,15,135);
INSERT INTO sales_fact VALUES(2,3,1,10,1,13);
SELECT * FROM sales_summary_bytime;
DELETE FROM sales_fact WHERE product_key = 1;
SELECT * FROM sales_summary_bytime;
UPDATE sales_fact SET units_sold = units_sold * 2;
SELECT * FROM sales_summary_bytime;
```

39.10. PL/pgSQL Under the Hood

This section discusses some implementation details that are frequently important for PL/pgSQL users to know.

39.10.1. Variable Substitution

SQL statements and expressions within a PL/pgSQL function can refer to variables and parameters of the function. Behind the scenes, PL/pgSQL substitutes query parameters for such references. Parameters will only be substituted in places where a parameter or column reference is syntactically allowed. As an extreme case, consider this example of poor programming style:

```
INSERT INTO foo (foo) VALUES (foo);
```

The first occurrence of `foo` must syntactically be a table name, so it will not be substituted, even if the function has a variable named `foo`. The second occurrence must be the name of a column of the table, so it will not be substituted either. Only the third occurrence is a candidate to be a reference to the function's variable.

> **Note:** PostgreSQL versions before 9.0 would try to substitute the variable in all three cases, leading to syntax errors.

Since the names of variables are syntactically no different from the names of table columns, there can be ambiguity in statements that also refer to tables: is a given name meant to refer to a table column, or a variable? Let's change the previous example to

```
INSERT INTO dest (col) SELECT foo + bar FROM src;
```

Here, `dest` and `src` must be table names, and `col` must be a column of `dest`, but `foo` and `bar` might reasonably be either variables of the function or columns of `src`.

By default, PL/pgSQL will report an error if a name in a SQL statement could refer to either a variable or a table column. You can fix such a problem by renaming the variable or column, or by qualifying the ambiguous reference, or by telling PL/pgSQL which interpretation to prefer.

The simplest solution is to rename the variable or column. A common coding rule is to use a different naming convention for PL/pgSQL variables than you use for column names. For example, if you consistently name function variables `v_`*`something`* while none of your column names start with `v_`, no conflicts will occur.

Alternatively you can qualify ambiguous references to make them clear. In the above example, `src.foo` would be an unambiguous reference to the table column. To create an unambiguous reference to a variable, declare it in a labeled block and use the block's label (see Section 39.2). For example,

```
<<block>>
DECLARE
    foo int;
BEGIN
    foo := ...;
    INSERT INTO dest (col) SELECT block.foo + bar FROM src;
```

Here `block.foo` means the variable even if there is a column `foo` in `src`. Function parameters, as well as special variables such as `FOUND`, can be qualified by the function's name, because they are implicitly declared in an outer block labeled with the function's name.

Sometimes it is impractical to fix all the ambiguous references in a large body of PL/pgSQL code. In such cases you can specify that PL/pgSQL should resolve ambiguous references as the variable (which is compatible with PL/pgSQL's behavior before PostgreSQL 9.0), or as the table column (which is compatible with some other systems such as Oracle).

To change this behavior on a system-wide basis, set the configuration parameter `plpgsql.variable_conflict` to one of `error`, `use_variable`, or `use_column` (where `error` is the factory default). This parameter affects subsequent compilations of statements in PL/pgSQL functions, but not statements already compiled in the current session. To set the parameter before PL/pgSQL has been loaded, it is necessary to have added "`plpgsql`" to the custom_variable_classes list in `postgresql.conf`. Because changing this setting can cause unexpected changes in the behavior of PL/pgSQL functions, it can only be changed by a superuser.

You can also set the behavior on a function-by-function basis, by inserting one of these special commands at the start of the function text:

```
#variable_conflict error
#variable_conflict use_variable
#variable_conflict use_column
```

These commands affect only the function they are written in, and override the setting of `plpgsql.variable_conflict`. An example is

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    #variable_conflict use_variable
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = curtime, comment = comment
          WHERE users.id = id;
    END;
$$ LANGUAGE plpgsql;
```

In the `UPDATE` command, `curtime`, `comment`, and `id` will refer to the function's variable and parameters whether or not `users` has columns of those names. Notice that we had to qualify the reference to `users.id` in the `WHERE` clause to make it refer to the table column. But we did not have to qualify the reference to `comment` as a target in the `UPDATE` list, because syntactically that must be a column of `users`. We could write the same function without depending on the `variable_conflict` setting in this way:

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    <<fn>>
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = fn.curtime, comment = stamp_user.comment
          WHERE users.id = stamp_user.id;
    END;
$$ LANGUAGE plpgsql;
```

Variable substitution does not happen in the command string given to `EXECUTE` or one of its variants. If you need to insert a varying value into such a command, do so as part of constructing the string value, or use `USING`, as illustrated in Section 39.5.4.

Variable substitution currently works only in `SELECT`, `INSERT`, `UPDATE`, and `DELETE` commands, because the main SQL engine allows query parameters only in these commands. To use a non-constant name or value in other statement types (generically called utility statements), you must construct the utility statement as a string and `EXECUTE` it.

39.10.2. Plan Caching

The PL/pgSQL interpreter parses the function's source text and produces an internal binary instruction tree the first time the function is called (within each session). The instruction tree fully translates the PL/pgSQL statement structure, but individual SQL expressions and SQL commands used in the function are not translated immediately.

As each expression and SQL command is first executed in the function, the PL/pgSQL interpreter creates a prepared execution plan (using the SPI manager's `SPI_prepare` and `SPI_saveplan` functions). Subsequent visits to that expression or command reuse the prepared plan. Thus, a function with conditional code that contains many statements for which execution plans might be required will only prepare and save those plans that are really used during the lifetime of the database connection. This can substantially reduce the total amount of time required to parse and generate execution plans for the statements in a PL/pgSQL function. A disadvantage is that errors in a specific expression or command cannot be detected until that part of the function is reached in execution. (Trivial syntax errors will be detected during the initial parsing pass, but anything deeper will not be detected until execution.)

A saved plan will be re-planned automatically if there is any schema change to any table used in the query, or if any user-defined function used in the query is redefined. This makes the re-use of prepared plans transparent in most cases, but there are corner cases where a stale plan might be re-used. An example is that dropping and re-creating a user-defined operator won't affect already-cached plans; they'll continue to call the original operator's underlying function, if that has not been changed. When necessary, the cache can be flushed by starting a fresh database session.

Because PL/pgSQL saves execution plans in this way, SQL commands that appear directly in a PL/pgSQL function must refer to the same tables and columns on every execution; that is, you cannot use a parameter as the name of a table or column in an SQL command. To get around this restriction, you can construct dynamic commands using the PL/pgSQL `EXECUTE` statement — at the price of constructing a new execution plan on every execution.

Another important point is that the prepared plans are parameterized to allow the values of PL/pgSQL variables to change from one use to the next, as discussed in detail above. Sometimes this means that a plan is less efficient than it would be if generated for a specific variable value. As an example, consider

```
SELECT * INTO myrec FROM dictionary WHERE word LIKE search_term;
```

where `search_term` is a PL/pgSQL variable. The cached plan for this query will never use an index on `word`, since the planner cannot assume that the `LIKE` pattern will be left-anchored at run time. To use an index the query must be planned with a specific constant `LIKE` pattern provided. This is another situation where `EXECUTE` can be used to force a new plan to be generated for each execution.

The mutable nature of record variables presents another problem in this connection. When fields of a record variable are used in expressions or statements, the data types of the fields must not change from one call of the function to the next, since each expression will be planned using the data type that is present when the expression is first reached. `EXECUTE` can be used to get around this problem when necessary.

If the same function is used as a trigger for more than one table, PL/pgSQL prepares and caches plans independently for each such table — that is, there is a cache for each trigger function and table combination, not just for each function. This alleviates some of the problems with varying data types; for instance, a trigger function will be able to work successfully with a column named `key` even if it happens to have different types in different tables.

Likewise, functions having polymorphic argument types have a separate plan cache for each combination of actual argument types they have been invoked for, so that data type differences do not cause unexpected failures.

Plan caching can sometimes have surprising effects on the interpretation of time-sensitive values. For example there is a difference between what these two functions do:

```
CREATE FUNCTION logfunc1(logtxt text) RETURNS void AS $$
    BEGIN
        INSERT INTO logtable VALUES (logtxt, 'now');
    END;
$$ LANGUAGE plpgsql;
```

and:

```
CREATE FUNCTION logfunc2(logtxt text) RETURNS void AS $$
    DECLARE
        curtime timestamp;
    BEGIN
        curtime := 'now';
        INSERT INTO logtable VALUES (logtxt, curtime);
    END;
$$ LANGUAGE plpgsql;
```

In the case of `logfunc1`, the PostgreSQL main parser knows when preparing the plan for the `INSERT` that the string `'now'` should be interpreted as `timestamp`, because the target column of `logtable` is of that type. Thus, `'now'` will be converted to a constant when the `INSERT` is planned, and then used in all invocations of `logfunc1` during the lifetime of the session. Needless to say, this isn't what the programmer wanted.

In the case of `logfunc2`, the PostgreSQL main parser does not know what type `'now'` should become and therefore it returns a data value of type `text` containing the string `now`. During the ensuing assignment to the local variable `curtime`, the PL/pgSQL interpreter casts this string to the `timestamp` type by calling the `text_out` and `timestamp_in` functions for the conversion. So, the computed time stamp is updated on each execution as the programmer expects.

39.11. Tips for Developing in PL/pgSQL

One good way to develop in PL/pgSQL is to use the text editor of your choice to create your functions, and in another window, use psql to load and test those functions. If you are doing it this way, it is a good idea to write the function using `CREATE OR REPLACE FUNCTION`. That way you can just reload the file to update the function definition. For example:

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $$
```

```
          ....
$$ LANGUAGE plpgsql;
```

While running psql, you can load or reload such a function definition file with:

```
\i filename.sql
```

and then immediately issue SQL commands to test the function.

Another good way to develop in PL/pgSQL is with a GUI database access tool that facilitates development in a procedural language. One example of such a tool is pgAdmin, although others exist. These tools often provide convenient features such as escaping single quotes and making it easier to recreate and debug functions.

39.11.1. Handling of Quotation Marks

The code of a PL/pgSQL function is specified in `CREATE FUNCTION` as a string literal. If you write the string literal in the ordinary way with surrounding single quotes, then any single quotes inside the function body must be doubled; likewise any backslashes must be doubled (assuming escape string syntax is used). Doubling quotes is at best tedious, and in more complicated cases the code can become downright incomprehensible, because you can easily find yourself needing half a dozen or more adjacent quote marks. It's recommended that you instead write the function body as a "dollar-quoted" string literal (see Section 4.1.2.4). In the dollar-quoting approach, you never double any quote marks, but instead take care to choose a different dollar-quoting delimiter for each level of nesting you need. For example, you might write the `CREATE FUNCTION` command as:

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $PROC$
          ....
$PROC$ LANGUAGE plpgsql;
```

Within this, you might use quote marks for simple literal strings in SQL commands and `$$` to delimit fragments of SQL commands that you are assembling as strings. If you need to quote text that includes `$$`, you could use `$Q$`, and so on.

The following chart shows what you have to do when writing quote marks without dollar quoting. It might be useful when translating pre-dollar quoting code into something more comprehensible.

1 quotation mark

To begin and end the function body, for example:

```
CREATE FUNCTION foo() RETURNS integer AS '
          ....
' LANGUAGE plpgsql;
```

Anywhere within a single-quoted function body, quote marks *must* appear in pairs.

2 quotation marks

For string literals inside the function body, for example:

```
a_output := ''Blah'';
SELECT * FROM users WHERE f_name=''foobar'';
```

In the dollar-quoting approach, you'd just write:

```
a_output := 'Blah';
SELECT * FROM users WHERE f_name='foobar';
```

which is exactly what the PL/pgSQL parser would see in either case.

4 quotation marks

When you need a single quotation mark in a string constant inside the function body, for example:

```
a_output := a_output || ″ AND name LIKE ″″foobar″″ AND xyz″
```

The value actually appended to `a_output` would be: `AND name LIKE 'foobar' AND xyz`.

In the dollar-quoting approach, you'd write:

```
a_output := a_output || $$ AND name LIKE 'foobar' AND xyz$$
```

being careful that any dollar-quote delimiters around this are not just `$$`.

6 quotation marks

When a single quotation mark in a string inside the function body is adjacent to the end of that string constant, for example:

```
a_output := a_output || ″ AND name LIKE ″″foobar″″″
```

The value appended to `a_output` would then be: `AND name LIKE 'foobar'`.

In the dollar-quoting approach, this becomes:

```
a_output := a_output || $$ AND name LIKE 'foobar'$$
```

10 quotation marks

When you want two single quotation marks in a string constant (which accounts for 8 quotation marks) and this is adjacent to the end of that string constant (2 more). You will probably only need that if you are writing a function that generates other functions, as in Example 39-8. For example:

```
a_output := a_output || ″ if v_″ ||
    referrer_keys.kind || ″ like ″″″″″
    || referrer_keys.key_string || ″″″″″
    then return ″″″  || referrer_keys.referrer_type
    || ″″″; end if;″;
```

The value of `a_output` would then be:

```
if v_... like ″...″ then return ″...″; end if;
```

In the dollar-quoting approach, this becomes:

```
a_output := a_output || $$ if v_$$ || referrer_keys.kind || $$ like '$$
    || referrer_keys.key_string || $$'
    then return '$$  || referrer_keys.referrer_type
    || $$'; end if;$$;
```

where we assume we only need to put single quote marks into `a_output`, because it will be re-quoted before use.

39.12. 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, 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 39.10.1. 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 39.11.1.)
- 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 39.6.3.5.)
- 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.

39.12.1. Porting Examples

Example 39-7 shows how to port a simple function from PL/SQL to PL/pgSQL.

Example 39-7. 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 varchar,
                                                  v_version varchar)
RETURN varchar 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 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 39-8 shows how to port a function that creates another function and how to handle the ensuing quoting problems.

Example 39-8. 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 VARCHAR,
                 v_domain IN VARCHAR, v_url IN VARCHAR) RETURN VARCHAR 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 39-9 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 39.12.3 there is a PL/pgSQL implementation of `instr` that you can use to make your porting easier.

Example 39-9. 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 VARCHAR,
    v_host OUT VARCHAR,  -- This will be passed back
    v_path OUT VARCHAR,  -- This one too
    v_query OUT VARCHAR) -- 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 39-10 shows how to port a procedure that uses numerous features that are specific to Oracle.

Example 39-10. 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;❶
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;❷

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

    IF a_running_job_count > 0 THEN
        COMMIT; -- free lock❸
        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, sysdate);
    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:

❶ There is no `PRAGMA` statement in PostgreSQL.

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

❸ 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';❶
    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 ❷
            -- don't worry if it already exists
    END;
END;
$$ LANGUAGE plpgsql;
```

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

❷ 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.

39.12.2. Other Things to Watch For

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

39.12.2.1. 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.

39.12.2.2. 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 39.5.4. Constructs of the type `EXECUTE 'SELECT * FROM $1';` will not work reliably unless you use these functions.

39.12.2.3. 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;
```

39.12.3. 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.
--
-- Searches string1 beginning at the nth character for the mth occurrence
-- of string2.  If n is negative, search backwards.  If m is not passed,
-- assume 1 (search starts at first character).
--

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


CREATE FUNCTION instr(string varchar, string_to_search 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 IN temp_str);

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN pos + beg_index - 1;
        END IF;
    ELSE
        ss_length := char_length(string_to_search);
        length := char_length(string);
        beg := length + beg_index - ss_length + 2;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            pos := position(string_to_search IN temp_str);

            IF pos > 0 THEN
                RETURN beg;
            END IF;

            beg := beg - 1;
        END LOOP;

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


CREATE FUNCTION instr(string varchar, string_to_search 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 beg_index > 0 THEN
        beg := beg_index;
        temp_str := substring(string FROM beg_index);

        FOR i IN 1..occur_index LOOP
            pos := position(string_to_search IN temp_str);

            IF i = 1 THEN
                beg := beg + pos - 1;
            ELSE
                beg := beg + pos;
            END IF;

            temp_str := substring(string FROM beg + 1);
        END LOOP;

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN beg;
        END IF;
    ELSE
        ss_length := char_length(string_to_search);
        length := char_length(string);
        beg := length + beg_index - ss_length + 2;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            pos := position(string_to_search IN temp_str);

            IF pos > 0 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;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```