EPAM Systems, RD Dep., RD Dep.

POSTGRESQL DB FOR DWH AND ETL BUILDING

Core PL/pgSQL

CONTENTS

1. INTRODUCTION TO SERVER-SIDE PROGRAMMING

A PostgreSQL Server is a powerful framework that can be used for all kinds of data processing, and even some non-data server tasks. It is a server platform that allows you to easily mix and match functions and libraries from several popular languages.

PostgreSQL has all of the native server-side programming features available in most larger database systems such as triggers, which are automated actions invoked automatically each time data is changed. However, it has uniquely deep abilities to override the built-in behavior down to very basic operators.

This unique PostgreSQL ability comes from its catalog-driven design, which stores information about data types, functions, and access methods. The ability of PostgreSQL to load user-defined functions via dynamic loading makes it rapidly changeable without having to recompile the database itself. Some examples of this customization include the following:

- Writing user-defined functions (UDF) to carry out complex computations
- Adding complicated constraints to make sure that the data in the server meets guidelines
- Creating triggers in many languages to make related changes to other tables, audit changes, forbid the action from taking place if it does not meet certain criteria, prevent changes to the database, enforce and execute business rules, or replicate data
- Defining new data types and operators in the database
- Using the geography types defined in the PostGIS package
- Adding your own index access methods for either the existing or new data types, making some queries much more efficient

PostgreSQL allows user-defined functions to be written in other languages besides SQL and C. These other languages are generically called procedural languages (PLs). For a function written in a procedural language, the database server has no built-in knowledge about how to interpret the function's source text.

There are currently four procedural languages available in the standard PostgreSQL distribution:

- PL/pgSQL,
- PL/Tcl,
- PL/Perl,
- PL/Python.

There are additional procedural languages available that are not included in the core distribution.

How exactly does PostgreSQL handle languages?

HANDLER: This function is actually the glue between PostgreSQL and any external language that you want to use. It is in charge of mapping PostgreSQL data structures to whatever is needed by the language, and it helps pass the code around.

VALIDATOR: This is the police officer of the infrastructure. If it is available, it will be in charge of delivering tasty syntax errors to the end user. Many languages are able to parse code before actually executing it. PostgreSQL can use that and tell you whether a function is correct or not when you create it. Unfortunately, not all languages can do this, so in some cases, you will still be left with problems showing up at runtime.

INLINE: If this is present, PostgreSQL will be able to run anonymous code blocks utilizing this handler function.

PostgreSQL includes its own programming language named PL/pgSQL that is aimed to integrate easily with SQL commands. PL stands for procedural language, and this is just one of the many languages available for writing server code. pgSQL is the shorthand for PostgreSQL.

1.1 OVERVIEW OF PL/PGSQL

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, procedures, and triggers,
- adds control structures to the SQL language,
- can perform complex computations,
- inherits all user-defined types, functions, procedures, 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.

1.1.1 STRUCTURE OF PL/PGSQL

Functions written in PL/pgSQL are defined to the server by executing CREATE FUNCTION commands. The function body is simply a string literal so far as CREATE FUNCTION is concerned. It is often helpful to use dollar quoting to write the function body, rather than the normal single quote syntax. Without dollar quoting, any single quotes or backslashes in the function body must be escaped by doubling them. Almost all the examples in this chapter use dollar-quoted literals for their function bodies.

PL/pgSQL is a block-structured language. The complete text of a function body 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.

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.

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.

When you create a PL/pgSQL function:

1. No execution plan is saved.
2. No checks for existence of tables, columns, or other functions are performed.
3. You do not know whether your function works or not, until you execute it (often more than one time, if there are multiple code paths).

1.1.2 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. The general syntax of a variable declaration is:

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

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. 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. Equal (=) can be used instead of PL/SQL-compliant :=.

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.

%TYPE provides the data type of a variable or table column. You can use this to declare variables that will hold database values.

```
variable%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.

1.1.3 ROW AND RECORD TYPES

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.

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

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 will be a row variable.

Record variables are similar to row-type variables, but they have no predefined structure.

```
name RECORD;
```

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.

1.1.4 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. 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 EXECUTEd for each execution of the IF statement, with the current values of the PL/pgSQL variables supplied as parameter values. Normally these details are not important to a PL/pgSQL user, but they are useful to know when trying to diagnose a problem.

2. FUNCTIONS AND PROCEDURES

2.1 ANONYMOUS CODE BLOCKS

In addition to functions, PostgreSQL allows the use of anonymous code blocks. The idea is to run code that is needed only once. This kind of code execution is especially useful for dealing with administrative tasks. Anonymous code blocks don't take parameters and are not permanently stored in the database, since they don't have names.

Example:

```
DO
$$
BEGIN
RAISE NOTICE 'current time: %', now();
END;
$$ LANGUAGE 'plpgsql';
```

In this example, the code only issues a message and quits. This string is passed to PostgreSQL using simple dollar quoting.

DO — execute an anonymous code block. The code block is treated as though it were the body of a function with no parameters, returning void. It is parsed and executed a single time.

The name of the procedural language the code is written in LANGUAGE. If omitted, the default is plpgsql.

2.2 USER-DEFINED FUNCTIONS

Since all modern programming languages include user-defined functions, people often assume that database functions are cut from the same cloth and if you know how to write functions and when to write functions in an application programming language, you can apply this knowledge to PostgreSQL. This could not be further from truth.

PostgreSQL has both built-in (internal) functions and user-defined functions.

Internal functions are written in the C language and are integrated with the PostgreSQL server. For each data type supported by PostgreSQL, there are a number of functions

that perform different operations with variables or column values of that type. Similar to imperative languages, there are functions for mathematical operations, functions to operate on strings, functions to operate on date/time, and many others. Moreover, the list of available functions and supported types expands with each new release.

User-defined functions are functions that you, the user, create. PostgreSQL supports three kinds of user-defined functions:

- Query language functions, that is, functions written in SQL
- C functions (written in C or C-like languages, like C++)
- Procedural language functions, written in one of the supported procedural languages (referred to as PL)

```
CREATE OR REPLACE FUNCTION func_name(arg1 arg1_datatype DEFAULT arg1_default)
RETURNS some type | set of some type | TABLE (..) AS
$$
BODY of function
$$
LANGUAGE language_of_function
```

CREATE FUNCTION defines a new function. ***CREATE OR REPLACE FUNCTION*** will either create a new function or replace an existing definition. To be able to define a function, the user must have the USAGE privilege on the language.

If a schema name is included, then the function is created in the specified schema. Otherwise it is created in the current schema. The name of the new function must not match any existing function or procedure with the same input argument types in the same schema. However, functions and procedures of different argument types can share a name (this is called overloading).

To replace the current definition of an existing function, use ***CREATE OR REPLACE FUNCTION***. If you drop and then recreate a function, the new function is not the same entity as the old; you will have to drop existing rules, views, triggers, etc. that refer to the old function. Use ***CREATE OR REPLACE FUNCTION*** to change a function definition without breaking objects that refer to the function.

Also, ***ALTER FUNCTION*** can be used to change most of the auxiliary properties of an existing function.

Most often, a function will have one or multiple parameters, but it might have none. For example, the internal function now(), which returns the current timestamp, does not have any parameters. We can assign default values to any function parameter, to be used if no specific value is explicitly passed.

In addition, there are multiple ways to define function parameters. For the example, parameters could be named, but they can also be positioned ($1, $2, etc.). Some parameters may be defined as OUT or INOUT, instead of specifying a return type.

```
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;
```

It is possible for a function to return no value; in this case, a function is specified as RETURNS VOID. These exist because previously, PostgreSQL did not have support for stored procedures, so the only way to package multiple statements together was inside a function.

2.2.1 CREATE FUNCTION ADDITIONAL OPTIONS

Functional definitions often include additional qualifiers to optimize execution and to enforce security:

- ***LANGUAGE*** - The language must be one installed in your database. Obtain a list with the SELECT lanname FROM pg_language; query.
- ***VOLATILITY*** - This setting clues the query planner as to whether outputs can be cached and used across multiple calls. Your choices are:
    - ***IMMUTABLE*** - The function will always return the same output for the same input. Think of arithmetic functions. Only immutable functions can be used in the definition of indexes.

  - ***STABLE*** - The function will return the same value for the same inputs within the same query.
  - ***VOLATILE*** - The function can return different values with each call, even with the same inputs. Think of functions that change data or depend on environment settings like system time. This is the default.

  Keep in mind that the volatility setting is merely a hint to the planner. The default value of ***VOLATILE*** ensures that the planner will always recompute the result. If you use one of the other values, the planner can still choose to forgo caching should it decide that recomputing is more cost-effective.

- ***STRICT*** - A function marked with this qualifier will always return NULL if any inputs are NULL. The planner skips evaluating the function altogether with any NULL inputs. When writing SQL functions, be cautious when marking a function as ***STRICT***, because it could prevent the planner from taking advantage of indexes.
- ***ROWS*** - Applies only to functions returning sets of records. The value provides an estimate of how many rows will be returned. The planner will take this value into consideration when coming up with the best strategy.
- ***SECURITY DEFINER*** - This causes execution to take place within the security context of the owner of the function. If omitted, the function executes under the context of the user calling the function. This qualifier is useful for giving people rights to update a table via a function when they do not have direct update privileges.
- ***PARALLEL*** - This qualifier allows the planner to run in parallel mode. By default, a function is marked as ***PARALLEL UNSAFE***, which prevents any queries containing the function from being distributed into separate work processes.

2.2.2 RETURNING FROM A FUNCTION

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

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.

In a function that returns a scalar type, the expression's result will automatically be cast into the function's return type as described for assignments. But to return a composite (row) value, you must write an expression delivering exactly the requested column set. This may require use of explicit casting.

```
-- functions returning a scalar type
RETURN 1 + 2;
RETURN scalar_var;


-- functions returning a composite type
RETURN composite_type_var;
RETURN (1, 2, 'three'::text);  -- must cast columns to correct types
```

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.

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,

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

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 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.

2.3 USER-DEFINED PROCEDURES

The term stored procedure has traditionally been used to actually talk about a function. Thus, it is essential that we understand the difference between a function and a procedure.

A function is part of a normal SQL statement and is not allowed to start or commit transactions. You can run it inside a SELECT statement.

A procedure, in contrast, is able to control transactions and even run multiple transactions one after the other. However, you cannot run it inside a SELECT statement. Instead, you have to invoke CALL.

Therefore, there is a fundamental distinction between functions and procedures. The terminology that you will find on the internet is not always clear. However, you have to be aware of those important differences. In PostgreSQL, functions have been around since the very beginning. However, the concept of a procedure, as outlined in this section, is new and has only been introduced in PostgreSQL 11.

```
CREATE PROCEDURE insert_data(a integer, b integer)
LANGUAGE SQL
AS $$
INSERT INTO tbl VALUES (a);
INSERT INTO tbl VALUES (b);
$$;

CALL insert_data(1, 2);
```

A procedure is a database object similar to a function. The key differences are:

- Procedures are defined with the ***CREATE PROCEDURE*** command, not ***CREATE FUNCTION***.
- Procedures do not return a function value; hence ***CREATE PROCEDURE*** lacks a ***RETURNS*** clause. However, procedures can instead return data to their callers via output parameters.
- While a function is called as part of a query or DML command, a procedure is called in isolation using the ***CALL*** command.
- A procedure can commit or roll back transactions during its execution (then automatically beginning a new transaction), so long as the invoking ***CALL*** command is not part of an explicit transaction block. A function cannot do that.
- Certain function attributes, such as strictness, don't apply to procedures. Those attributes control how the function is used in a query, which isn't relevant to procedures.

A procedure does not have a return value. A procedure can therefore end without a RETURN statement. If you wish to use a RETURN statement to exit the code early, write just RETURN with no expression.

If the procedure has output parameters, the final values of the output parameter variables will be returned to the caller.

3. TRANSACTION MANAGEMENT

As you know, everything that PostgreSQL exposes in userland is a transaction.

The same, of course, applies if you are writing functions. A function is always part of the transaction you are in. It is not autonomous, just like an operator or any other operation.

The most important difference between how functions and stored procedures are executed is that you can commit or roll back a transaction within a procedure body.

At the start of the procedure execution(as well as in anonymous code blocks), a new transaction starts, and any ***COMMIT*** or ***ROLLBACK*** command within a function body will terminate the current transaction and start a new one. One of the use cases is the bulk data load.

```
CREATE PROCEDURE transaction_test()
LANGUAGE plpgsql
AS $$
BEGIN
    FOR i IN 0..9 LOOP
        INSERT INTO test1 (a) VALUES (i);
        IF i % 2 = 0 THEN
            COMMIT;
        ELSE
            ROLLBACK;
        END IF;
    END LOOP;
END;
$$;

CALL transaction_test();
```

In cases where transactions are committed in a loop, it might be desirable to start new transactions automatically with the same characteristics as the previous one. The commands ***COMMIT AND CHAIN*** and ***ROLLBACK AND CHAIN*** accomplish this.

Special considerations apply to cursor loops.

Normally, cursors are automatically closed at transaction commit. However, a cursor created as part of a loop like this is automatically converted to a holdable cursor by the first COMMIT or ROLLBACK. That means that the cursor is fully evaluated at the first COMMIT or ROLLBACK rather than row by row. The cursor is still removed automatically after the loop, so this is mostly invisible to the user.

Transaction commands are not allowed in cursor loops driven by commands that are not read-only (for example UPDATE ... RETURNING).

A transaction cannot be ended inside a block with exception handlers.

4. EXCEPTION PROCESSING

For programming languages, in every program, and in every module, error handling is an important thing. Everything is expected to go wrong once in a while, and therefore it is vital, and of key importance, to handle errors properly and professionally. In PL/pgSQL, you can use EXCEPTION blocks to handle errors.

The idea is that if the BEGIN block does something wrong, the EXCEPTION block will take care of it, and handle the problem correctly. Just like many other languages, you can react to different types of errors and catch them separately.

In the following example, the code might run into a division-by-zero problem. The goal is to catch this error and react accordingly:

```
CREATE FUNCTION error_test1(int, int) RETURNS int AS
$$
BEGIN
RAISE NOTICE 'debug message: % / %', $1, $2;
BEGIN
RETURN $1 / $2;
EXCEPTION
WHEN division_by_zero THEN
RAISE NOTICE 'division by zero detected: %', sqlerrm;
WHEN others THEN
RAISE NOTICE 'some other error: %', sqlerrm;
END;
RAISE NOTICE 'all errors handled';
RETURN 0;
END;
$$ LANGUAGE 'plpgsql';
```

The ***BEGIN*** block can clearly throw an error because there can be a division by zero. However, the ***EXCEPTION*** block catches the error that we are looking at and also takes care of all other potential problems that can unexpectedly pop up.

Technically, this is more or less the same as a *savepoint*, and therefore the error does not cause the entire transaction to fail completely. Only the block that is causing the error will be subject to a mini rollback.

By inspecting the *sqlerrm* variable, you can also have direct access to the error message itself.

Within an exception handler, one may also retrieve information about the current exception by using the ***GET STACKED DIAGNOSTICS*** command.

PostgreSQL catches the exception and shows the message in the ***EXCEPTION*** block. It is kind enough to show us the line that is the error. This makes it a whole lot easier to debug and fix the code if it is broken.

In some cases, it also makes sense to raise your own exception.

4.1 RAISE

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

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.

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 %. The number of arguments must match the number of % placeholders in the format string, or an error is raised during the compilation of the function.

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

You can attach additional information to the error report by writing ***USING*** followed by option = expression items. Each expression can be any string-valued expression. The allowed option key words are:

- 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.
- 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.
- COLUMN
- CONSTRAINT
- DATATYPE
- TABLE
- SCHEMA: Supplies the name of a related object.

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.

4.2 ASSERT

The ASSERT statement is a convenient shorthand for inserting debugging checks into PL/pgSQL functions.

```
ASSERT condition [ , message ];
```

The condition is a Boolean expression that is expected to always evaluate to true; if it does, the ASSERT statement does nothing further. If the result is false or null, then an ASSERT_FAILURE exception is raised. (If an error occurs while evaluating the condition, it is reported as a normal error.)

If the optional message is provided, it is an expression whose result (if not null) replaces the default error message text “assertion failed”, should the condition fail. The message expression is not evaluated in the normal case where the assertion succeeds.

Testing of assertions can be enabled or disabled via the configuration parameter plpgsql.check_asserts, which takes a Boolean value; the default is on. If this parameter is off then ASSERT statements do nothing.

Note that ASSERT is meant for detecting program bugs, not for reporting ordinary error conditions. Use the RAISE statement, described above, for that.

5. SOURCE BOOKS AND ARTICLES

1. PostgreSQL 13.4 Documentation, The PostgreSQL Global Development Group, 1996–2021
2. Henrietta Dombrovskaya, Boris Novikov, Anna Bailliekova PostgreSQL Query Optimization: The Ultimate Guide to Building Efficient Queries, Apress, 2021
3. Hans-Jürgen Schönig, Mastering PostgreSQL 12 Third Edition, Packt Publishing, 2019