

PL/pgSQL Debugging



16

Copyright

© Postgres Professional, 2017–2025

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov, Igor Gnatyuk

Translated by: Liudmila Mantrova, Alexander Meleshko, Elena Sharafutdinova

Photo by: Oleg Bartunov (Phu monastery, Bhrikuti summit, Nepal)

Use of Course Materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

Feedback

Please send your feedback, comments and suggestions to:

edu@postgrespro.ru

Disclaimer

In no event shall Postgres Professional company be liable for any damages or loss, including loss of profits, that arise from direct or indirect, special or incidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

Correctness Checks

PL/pgSQL Debugger

Debug Messages and their Implementations

Session Tracing

Correctness Checks



Compile-time and runtime checks

plpgsql.extra_warnings

plpgsql.extra_errors

additional checks provided by the `plpgsql_check` extension

Built-in checks

`ASSERT` statement

Testing

3

Debugging implies executing a program and analyzing the occurred issues, typically by running a debugger or by displaying debug messages.

But you can also avoid some particular error classes if you enable compile-time and runtime verification of source code. It is controlled by *plpgsql.extra_warnings* and *plpgsql.extra_errors* parameters, as explained in the PL/pgSQL. Query execution lesson.

<https://postgrespro.com/docs/postgresql/16/plpgsql-development-tips#PLPGSQL-EXTRA-CHECKS>

The extension `plpgsql_check`, mentioned earlier in the course, offers a wider range of checks.

Another way to make your code more secure is to check for conditions that must always hold true (the so-called sanity checks). A convenient way to do it is to use the `ASSERT` statement.

<https://postgrespro.com/docs/postgresql/16/plpgsql-errors-and-messages#PLPGSQL-STATEMENTS-ASSERT>

We must also mention the importance of testing the code. Apart from making sure from the very beginning that the code works as expected, testing also facilitates further maintenance: it ensures that the existing functionality is not broken by the introduced changes. We will not expand on this topic; but it's worth noting that testing the code that accesses a database can turn out to be quite tricky as you have to prepare test cases.

Correctness Checks

```
=> CREATE DATABASE plpgsql_debug;
```

```
CREATE DATABASE
```

```
=> \c plpgsql_debug
```

You are now connected to database "plpgsql_debug" as user "student".

The ASSERT statement specifies conditions that, when violated, raise an error. These are somewhat similar to database integrity constraints.

Example: a function that takes an apartment number in a block and returns the entrance number:

```
=> CREATE FUNCTION entrance(  
    floors integer,  
    flats_per_floor integer,  
    flat_no integer  
)  
RETURNS integer  
AS $$  
BEGIN  
    RETURN floor((flat_no - 1)::real / (floors * flats_per_floor)) + 1;  
END  
$$ LANGUAGE plpgsql IMMUTABLE;
```

```
CREATE FUNCTION
```

Let's test the function by providing some sample values:

```
=> SELECT entrance(9, 4, 1), entrance(9, 4, 36), entrance(9, 4, 37);
```

```
entrance | entrance | entrance  
-----+-----+-----  
1 | 1 | 2  
(1 row)
```

But with some invalid input values, the function will return invalid output. This output may then get forwarded to other programs, possibly breaking them too. Testing only the function code here will not suffice.

```
=> SELECT entrance(9, 4, 0);
```

```
entrance  
-----  
0  
(1 row)
```

For additional safety, let's add a check:

```
=> CREATE OR REPLACE FUNCTION entrance(  
    floors integer,  
    flats_per_floor integer,  
    flat_no integer  
)  
RETURNS integer  
AS $$  
BEGIN  
    ASSERT floors > 0 AND flats_per_floor > 0 AND flat_no > 0,  
        'Bad input';  
    RETURN floor((flat_no - 1)::real / (floors * flats_per_floor)) + 1;  
END  
$$ LANGUAGE plpgsql IMMUTABLE;
```

```
CREATE FUNCTION
```

```
=> SELECT entrance(9, 4, 0);
```

```
ERROR: Bad input
```

```
CONTEXT: PL/pgSQL function entrance(integer,integer,integer) line 3 at ASSERT
```

Now, a call with incorrect parameters will immediately return an error.

Interface

- the API is provided as an extension (pldbgapi)
- built-in support is available in some GUIs

Features

- setting breakpoints
- step-by-step execution
- checking and setting variable values
- no need to modify the code
- debugging applications on the fly

As its name suggests, PL/pgSQL Debugger is a debugging tool for PL/pgSQL. It is delivered as the pldbgapi extension, which is officially supported by PostgreSQL developers.

The pldbgapi extension is a collection of interface functions for the PostgreSQL server that enable you to set breakpoints, execute the application code step-by-step, check and set variable values.

There is no need to modify the source code of the application to debug, so debugging can be performed on the fly. In other words, you do not have to restart the process with an attached debugger, you can simply connect to the current session and start debugging it.

It is inconvenient to use these functions directly; they are mainly targeted for IDEs with graphical user interface. Some of these IDEs (including DBeaver) have a convenient built-in debugging user interface. But in order to use it, you still have to install the pldbgapi extension into the corresponding database first.

The source code of pldbgapi is available at:

<https://github.com/EnterpriseDB/pldebugger>

Debugging with PL/pgSQL Debugger

Let's debug one of the functions of our application.

The VM comes with the debugger support package postgresql-16-pldebugger already installed. The debugger needs its shared library to be loaded. Add it to the list and restart the server:

```
=> ALTER SYSTEM SET shared_preload_libraries = 'plugin_debugger';
```

```
ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 16 main restart
```

Install the debugger extension into the bookstore database:

```
student$ psql bookstore
```

```
| => CREATE EXTENSION pldbgapi SCHEMA public;
```

```
| CREATE EXTENSION
```

We will use DBeaver, a GUI tool, for debugging.

```
student$ /opt/dbeaver/dbeaver 2>/dev/null
```

Not only debugging

- monitoring of long-running processes
- application logging

Implementation strategies

- monitoring long-running processes
- writing messages to a table or a file
- passing information to other processes

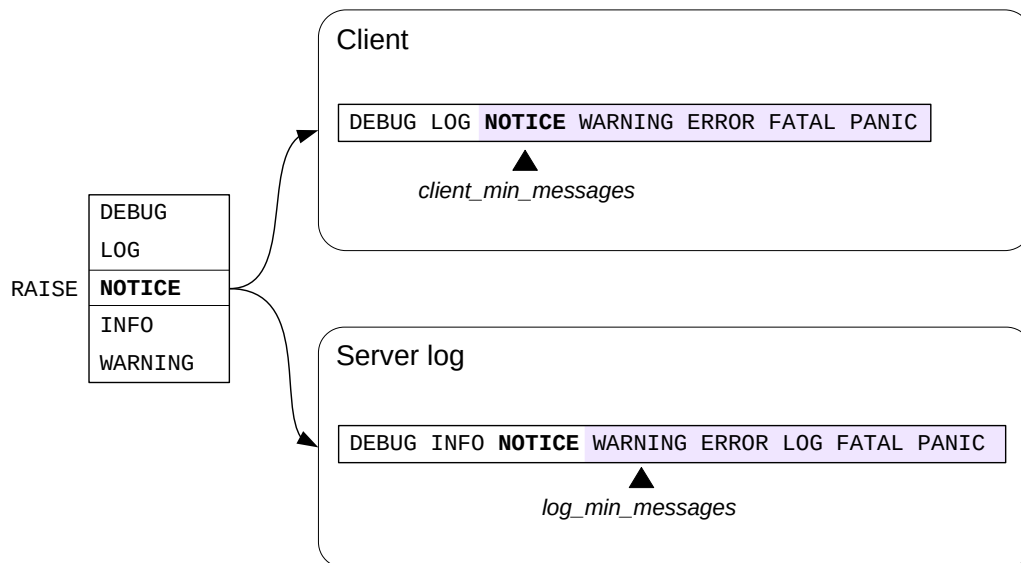
Another debugging approach consists in adding debug messages to the key parts of the code to provide the current context. As you analyze these messages, you can understand what exactly has gone wrong.

Apart from the debugging itself, debug messages can also perform other functions. Messages help to track down what stage the process is currently in. Similar to a database log, an application can write its own log. Having such a log with important data (for example, report-related data: the name of the report, the user who has collected it, date, arguments, etc.) can greatly facilitate technical support.

We can single out several strategies of implementing debug messages in PL/pgSQL. Apart from using the already familiar RAISE command, which can display messages in the terminal (or save them into the server log), it is also possible to send messages to another process, as well as write them into a table or into a file.

When choosing the approach to use, you have to take a lot of different aspects into account. Are messages transactional (are they sent before the end of the transaction or only after it has been committed)? Can you send them from several sessions simultaneously? How can you set up access to the log file and clean up old log entries? How does logging affect performance? Do you have to modify the source code?

RAISE Statement



8

We are already familiar with the RAISE statement. It can be used both to raise exceptions (which is discussed in detail in the PL/pgSQL. Error Handling lesson) and to emit messages. Such messages can not only be sent to the client, but also written to the server log.

In a simple debugging case, you have to add RAISE NOTICE calls to the function code, start the function execution (for example, in a psql session), and analyze the received messages as the execution progresses. RAISE messages are non-transactional: they are emitted asynchronously and do not depend on the transaction status.

Message delivery is controlled by message levels (DEBUG, LOG, NOTICE, INFO, WARNING) and server parameters. Parameter values determine whether a message will be sent to the client (`client_min_messages`) and/or written to the server log (`log_min_messages`). A message will be sent if the RAISE message level is not lower than the value of the corresponding parameter (is shown to the right of the parameter value on this slide).

In the default configuration, NOTICE messages are only sent to the client, LOG messages are only written to the log file, and WARNING messages are both sent to the client and written to the log file.

INFO messages are always sent to the client; they cannot be trapped using the `client_min_messages` parameter.

<https://postgrespro.com/docs/postgresql/16/plpgsql-errors-and-messages>

RAISE Statement

Let's create a function that takes a table name as input and returns the number of rows in that table.

```
student$ psql plpgsql_debug

=> CREATE FUNCTION get_count(tabname text) RETURNS bigint
AS $$
DECLARE
    cmd text;
    retval bigint;
BEGIN
    cmd := 'SELECT COUNT(*) FROM ' || quote_ident(tabname);
    RAISE NOTICE 'cmd: %', cmd;
    EXECUTE cmd INTO retval;
    RETURN retval;
END
$$ LANGUAGE plpgsql STABLE;

CREATE FUNCTION
```

For dynamic execution, it is better to save the command text to a variable. In case of an error, you can analyze the variable value.

```
=> SELECT get_count('pg_class');

NOTICE: cmd: SELECT COUNT(*) FROM pg_class
get_count
-----
         413
(1 row)
```

The string starting with "NOTICE" is our debug information.

RAISE can be used to track the progress of a long query.

Let's consider a piece of code that has three distinct stages. As the code executes, we want to understand which stage we are at. To do this, we will call this procedure at each stage:

```
=> CREATE PROCEDURE debug_message(msg text)
AS $$
BEGIN
    RAISE NOTICE '%', msg;
END
$$ LANGUAGE plpgsql;

CREATE PROCEDURE
```

Upon completing each stage, the main procedure calls debug_message:

```
=> CREATE PROCEDURE long_running()
AS $$
BEGIN
    CALL debug_message('long_running. Stage 1/3...');
    PERFORM pg_sleep(2);
    CALL debug_message('long_running. Stage 2/3...');
    PERFORM pg_sleep(3);
    CALL debug_message('long_running. Stage 3/3...');
    PERFORM pg_sleep(1);
    CALL debug_message('long_running. Done');
END
$$ LANGUAGE plpgsql;

CREATE PROCEDURE
```

The RAISE statement issues messages immediately, rather than at the end of the main procedure:

```
=> CALL long_running();

NOTICE: long_running. Stage 1/3...
NOTICE: long_running. Stage 2/3...
NOTICE: long_running. Stage 3/3...
NOTICE: long_running. Done
CALL
```

This approach works fine when you call a function within a session. If, however, the function is called by an application, writing to and then reading from the server message log is much more convenient.

Let's make debug_message emit a message with a specific level. The level will be defined by a custom parameter app.raise_level:

```
=> CREATE OR REPLACE PROCEDURE debug_message(msg text)
AS $$
BEGIN
    CASE current_setting('app.raise_level', true)
        WHEN 'NOTICE' THEN RAISE NOTICE '%, %, %', user, clock_timestamp(), msg;
        WHEN 'DEBUG' THEN RAISE DEBUG '%, %, %', user, clock_timestamp(), msg;
        WHEN 'LOG' THEN RAISE LOG '%, %, %', user, clock_timestamp(), msg;
        WHEN 'INFO' THEN RAISE INFO '%, %, %', user, clock_timestamp(), msg;
        WHEN 'WARNING' THEN RAISE WARNING '%, %, %', user, clock_timestamp(), msg;
        ELSE NULL; -- other values turn off the messages
    END CASE;
END
$$ LANGUAGE plpgsql;

CREATE PROCEDURE
```

For this example, set it just for the current session:

```
=> SET app.raise_level TO 'NONE';

SET
```

This way, the procedure will not post any debug messages during “production” (app.raise_level = NONE):

```
=> CALL long_running();

CALL
```

Starting the function in a separate session, we can get debug messages by setting the app.raise_level to NOTICE:

```
=> SET app.raise_level TO 'NOTICE';

SET

=> CALL long_running();

NOTICE: student, 2025-11-27 14:25:05.09135+03, long_running. Stage 1/3...
NOTICE: student, 2025-11-27 14:25:07.092756+03, long_running. Stage 2/3...
NOTICE: student, 2025-11-27 14:25:10.09479+03, long_running. Stage 3/3...
NOTICE: student, 2025-11-27 14:25:11.095751+03, long_running. Done
CALL
```

To enable both debugging in the application and logging, set the message level to LOG:

```
=> SET app.raise_level TO 'LOG';

SET

=> CALL long_running();

CALL
```

Check the server log:

```
student$ tail -n 30 /var/log/postgresql/postgresql-16-main.log | grep 'long_running\.'

2025-11-27 14:25:11.215 MSK [40142] student@plpgsql_debug LOG: student, 2025-11-27
14:25:11.215472+03, long_running. Stage 1/3...
        SQL statement "CALL debug_message('long_running. Stage 1/3...')"
```

```
2025-11-27 14:25:13.216 MSK [40142] student@plpgsql_debug LOG: student, 2025-11-27
14:25:13.216795+03, long_running. Stage 2/3...
        SQL statement "CALL debug_message('long_running. Stage 2/3...')"
```

```
2025-11-27 14:25:16.219 MSK [40142] student@plpgsql_debug LOG: student, 2025-11-27
14:25:16.219773+03, long_running. Stage 3/3...
        SQL statement "CALL debug_message('long_running. Stage 3/3...')"
```

```
2025-11-27 14:25:17.220 MSK [40142] student@plpgsql_debug LOG: student, 2025-11-27
14:25:17.220785+03, long_running. Done
        SQL statement "CALL debug_message('long_running. Done')"
```

By adjusting app.raise_level, log_min_messages, and client_min_messages, you can achieve different debug message output behavior.

Very importantly, all this is done without tampering with the application code.

Process → Process (IPC)

NOTIFY → LISTEN

SQL commands

transactional execution is inconvenient for debugging

Session status

the *application_name* parameter

is visible in the `pg_stat_activity` view and in the output of the `ps` command

can be used in log messages

In PostgreSQL, server processes can communicate between each other. Among the built-in solutions, the following are worth noting.

- Sending messages via the NOTIFY command in one process and getting them via LISTEN in another. But since these commands are transactional, it is inconvenient to use them for debugging:
 1. Messages are sent only at commit time, not right after the NOTIFY command execution, so it is impossible to track the execution progress.
 2. If the transaction fails, messages won't be sent at all.
- Using the *application_name* parameter. A session with a long-running process can periodically write its execution status into the *application_name* parameter. In a separate session, a DBA can poll the `pg_stat_activity` view, which contains detailed information about all active sessions. The *application_name* value is usually also visible in the output of the `ps` command. The *application_name* value can also be written to the server log (if you set up the *log_line_prefix* parameter). As a result, relevant log entries will be easier to find.

<https://postgrespro.com/docs/postgresql/16/runtime-config-logging#RUNTIME-CONFIG-LOGGING-WHAT>

Session Status

application_name is a useful parameter for debugging. The first session changes its value, the second session checks the pg_stat_activity view in a loop.

```
| => \c plpgsql_debug
```

```
| You are now connected to database "plpgsql_debug" as user "student".
```

New version of the procedure:

```
=> CREATE OR REPLACE PROCEDURE debug_message(msg text)
AS $$
BEGIN
    PERFORM set_config('application_name', format('%s', msg), /* is_local */ true);
END
$$ LANGUAGE plpgsql;
```

CREATE PROCEDURE

In the first session:

```
=> CALL long_running();
```

In the second session, read the value from pg_stat_activity every two seconds:

```
| => SELECT pid, username, application_name
| FROM pg_stat_activity
| WHERE datname = 'plpgsql_debug' AND pid <> pg_backend_pid();
```

```
| pid | username | application_name
|-----+-----+-----
| 40142 | student | long_running. Stage 1/3...
| (1 row)
```

```
| => \g
```

```
| pid | username | application_name
|-----+-----+-----
| 40142 | student | long_running. Stage 2/3...
| (1 row)
```

```
| => \g
```

```
| pid | username | application_name
|-----+-----+-----
| 40142 | student | long_running. Stage 3/3...
| (1 row)
```

```
| => \g
```

```
| pid | username | application_name
|-----+-----+-----
| 40142 | student | psql
| (1 row)
```

CALL

The dblink extension

is part of the server

incurs additional costs for opening a new connection

Autonomous transactions

Postgres Pro Enterprise

Another way to save debug messages is to write them into a database table.

One of the advantages of this approach is that log access and concurrent execution are managed by the database system itself.

But you have to make sure that insertion operations on this table are non-transactional. It can be done using the dblink extension, which is provided as part of the PostgreSQL server. This extension enables you to open another connection to the same database, so insertion is performed in a separate transaction.

As for the disadvantages, opening a new connection takes additional server resources.

We cover dblink usage in more detail in the DEV2 course.

<https://postgrespro.com/docs/postgresql/16/dblink>

Postgres Pro Enterprise implements autonomous transactions with lower overhead than dblink usage (See the PGPRO course about Postgres Pro Enterprise features).

Writing to a Table: the dblink Extension

Install the extension:

```
=> CREATE EXTENSION dblink;
```

```
CREATE EXTENSION
```

Create a table to write into.

It makes sense to store information about the user and the time of record. The id column is there to make sure that the table is sorted in the order in which the rows are added.

```
=> CREATE TABLE log (  
    id          integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,  
    username    text,  
    ts          timestampz,  
    message     text  
);
```

```
CREATE TABLE
```

Now, rewrite the procedure to add records to the table. The procedure opens a new session, performs an insert in a separate transaction, and closes the session.

```
=> CREATE OR REPLACE PROCEDURE debug_message(msg text)  
AS $$  
DECLARE  
    cmd text;  
BEGIN  
    cmd := format(  
        'INSERT INTO log (username, ts, message)  
        VALUES (%L, %L::timestampz, %L)',  
        user, clock_timestamp()::text, debug_message.msg  
    );  
    PERFORM dblink('dbname=' || current_database(), cmd);  
END  
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

Let's test it: run our long_running procedure in a separate transaction, then roll it back.

```
=> BEGIN;
```

```
BEGIN
```

```
=> CALL long_running();
```

```
CALL
```

```
=> ROLLBACK;
```

```
ROLLBACK
```

Check if the table has all the debug_message records. The timestamps in ts tell us how much time has passed between the calls.

```
=> SELECT username, to_char(ts, 'HH24:MI:SS') as ts, message  
FROM log  
ORDER BY id;
```

username	ts	message
student	14:25:25	long_running. Stage 1/3...
student	14:25:27	long_running. Stage 2/3...
student	14:25:30	long_running. Stage 3/3...
student	14:25:31	long_running. Done

(4 rows)

The adminpack extension

is part of the server
among other things, allows writing text files

Untrusted languages

for example, PL/Perl

You can also write debug messages into an OS file.

It can be done using the adminpack extension, which allows writing data to any file that can be accessed by the *postgres* OS user.

Another option is to create a function in an untrusted language (such as PL/Perl — plperl) that will perform the same task. Various server-side programming languages are covered in the DEV2 course.

<https://postgrespro.com/docs/postgresql/16/adminpack>

Writing to a File: pg_file_write

Install the extension:

```
=> CREATE EXTENSION adminpack;
```

CREATE EXTENSION

Rewrite debug_message again, now making it write debug information to a file. The postgres OS user who started the DBMS instance must be allowed write to this file. Let's place it in the user's home directory.

```
=> CREATE OR REPLACE PROCEDURE debug_message(msg text)
AS $$
DECLARE
    filename CONSTANT text := '/var/lib/postgresql/log.txt';
    message text;
BEGIN
    message := format(E'%s, %s, %s\n',
        session_user, clock_timestamp()::text, debug_message.msg
    );
    PERFORM pg_file_write(filename, message, /* append */ true);
END
$$ LANGUAGE plpgsql;
```

CREATE PROCEDURE

The function writes a separate line to the log file: the message passed by the parameter, the user responsible, and the timestamp.

Test it: run long_running in a separate transaction, then roll it back.

```
=> BEGIN;
```

BEGIN

```
=> CALL long_running();
```

CALL

```
=> ROLLBACK;
```

ROLLBACK

Check the file contents. To access the file as student, you need to use sudo:

```
student$ sudo cat /var/lib/postgresql/log.txt
```

```
student, 2025-11-27 14:25:31.951922+03, long_running. Stage 1/3...
student, 2025-11-27 14:25:33.953793+03, long_running. Stage 2/3...
student, 2025-11-27 14:25:36.955905+03, long_running. Stage 3/3...
student, 2025-11-27 14:25:37.95683+03, long_running. Done
```


Standard tracing into the log file

- logging overhead
- large log file
- profiling tools are required
- access to the log is required (security)

Settings

long-running statements	<i>log_min_duration_statement</i>
the statements to log	<i>log_statement</i>
message context	<i>log_line_prefix</i>
...	

16

In some cases, it may be useful to trace everything that happens in the code. Using the built-in functionality, you can save the executed SQL queries into the server log file. Make sure to take into account the following specifics:

- A high-load application can execute a huge number of queries. Writing them into a file can affect performance of the I/O subsystem.
- In most cases, you have to use special tools to analyze such data sets. A de-facto standard is pgBadger.
<https://github.com/darold/pgbadger>
- Application developers may have no access to the log file on the server. Besides, in production systems, log files may contain commands with confidential information.

PostgreSQL provides several parameters to configure tracing; the main ones are listed on the slide. The full list is available here:

<https://postgrespro.com/docs/postgresql/16/runtime-config-logging>

You do not have to set configuration parameters for the whole cluster. Their scope can be limited to particular sessions using SET, ALTER DATABASE, and ALTER ROLE commands (as explained in Basic Tools. Installation and Management, psql and Data Organization. Logical Structure lessons).

The auto_explain extension

- logging execution plans
- tracing nested statements

Settings

plans of long-running commands	<i>auto_explain.log_min_duration</i>
nested statements	<i>auto_explain.log_nested_statements</i>
...	

When tracing is enabled, SQL commands make it into the log in their exact form that has been sent to the server. If a PL/pgSQL routine was called, the log will contain only this top-level call (for example, SELECT or CALL statement), but not the commands executed within the routine.

To log nested queries in addition to top-level commands, you have to use the auto_explain extension.

As suggested by its name, the main objective of this extension is to log both the text of the command and its execution plan. It can turn out to be useful, although it is not exactly tracing, but rather query optimization (which is covered in the QPT course).

<https://postgrespro.com/docs/postgresql/16/auto-explain>

The plpgsql_check extension

overhead incurred by logging
loads of output data

The main settings

enabling tracing	<code>plpgsql_check.enable_tracer</code>
	<code>plpgsql_check.tracer</code>
message levels	<code>plpgsql_check.tracer_errlevel</code>

To figure out which code has been executed as you are looking at the log, you have to match SQL queries with PL/pgSQL routines, and it may be not that easy. There are no built-in tools for tracing PL/pgSQL code, but you can do it with the help of the `plpgsql_check` extension developed by Pavel Stehule (we have already mentioned this extension in PL/pgSQL. Executing Queries lesson).

Such tracing causes significant overhead and should only be used for debugging, not in production operations.

https://github.com/okbob/plpgsql_check

Session Tracing

One simple way to enable tracing is to set the `log_statement` parameter to 'all' (write all commands, including DDL, data modification, and queries).

```
=> SET log_statement = 'all';
```

SET

Run any query:

```
=> SELECT get_count('pg_views');
```

```
NOTICE: cmd: SELECT COUNT(*) FROM pg_views
get_count
-----
      141
(1 row)
```

Disable tracing:

```
=> RESET log_statement;
```

RESET

Information about the executed commands will appear in the server log:

```
student$ tail -n 2 /var/log/postgresql/postgresql-16-main.log
```

```
2025-11-27 14:25:38.285 MSK [40142] student@plpgsql_debug LOG: statement: SELECT
get_count('pg_views');
2025-11-27 14:25:38.418 MSK [40142] student@plpgsql_debug LOG: statement: RESET
log_statement;
```

Only the top level command is logged, not the query from within `get_count`.

This is where the `auto_explain` extension comes in. This extension does not need to be installed, but must be loaded into memory. You can either set it up to load for the whole instance with `shared_preload_libraries` parameter, or just load it for the current session:

```
=> LOAD 'auto_explain';
```

LOAD

Enable tracing for all commands, regardless of execution time.

```
=> SET auto_explain.log_min_duration = 0;
```

SET

Enable tracing for nested queries:

```
=> SET auto_explain.log_nested_statements = on;
```

SET

The messages are raised the same way as with the `RAISE` statement. The `LOG` level is used by default, usually implying that the messages are recorded to the server log. You can change the log level to output tracing messages directly to the screen:

```
=> SET auto_explain.log_level = 'NOTICE';
```

SET

Try the query again:

```
=> SELECT get_count('pg_views');
```

```

NOTICE: cmd: SELECT COUNT(*) FROM pg_views
NOTICE: duration: 0.090 ms plan:
Query Text: SELECT COUNT(*) FROM pg_views
Aggregate (cost=19.52..19.53 rows=1 width=8)
-> Seq Scan on pg_class c (cost=0.00..19.16 rows=141 width=0)
    Filter: (relkind = 'v'::"char")
NOTICE: duration: 1.244 ms plan:
Query Text: SELECT get_count('pg_views');
Result (cost=0.00..0.26 rows=1 width=8)
get_count
-----
      141
(1 row)

```

Now we see not only the function call, but also the nested query and the execution plans.

Takeaways



PL/pgSQL Debugger is a debugger API used in graphical development environments

Debugging output can be displayed in the terminal, written into the server log, a table, or a file; it can also be sent to other processes

Sessions can be traced

1. Modify the `get_catalog` function, so that the dynamically constructed text of the query is written into the server log.
In the application, perform search several times by filling out different fields; make sure that SQL commands are constructed correctly.
2. Enable tracing of SQL statements at the server level.
Perform some actions in the application and check which commands are logged.
Disable tracing.

2. To enable tracing, set the `log_min_duration_statement` parameter to 0 and reload the configuration. All commands will be logged, together with their execution time.

The easiest way to do it is to use the `ALTER SYSTEM SET` command. Other ways of setting parameters are covered in Basic Tools. Installation and Management, `psql` lesson. Remember to reload the server configuration file.

After having a look at the log file, you should reset `log_min_duration_statement` to the default value (-1) to disable tracing. It is convenient to use `ALTER SYSTEM RESET` for this purpose.

1. The get_catalog Function

The text of a dynamic query is stored in a separate variable, which is saved to the server log before executing. Include parameter values in the message for clarity.

Debug strings are marked with the “DEBUG get_catalog” substring in the log.

After debugging, the RAISE LOG command can be removed or commented out.

```
=> CREATE OR REPLACE FUNCTION get_catalog(
    author_name text,
    book_title text,
    in_stock boolean
)
RETURNS TABLE(book_id integer, display_name text, onhand_qty integer)
AS $$
DECLARE
    title_cond text := '';
    author_cond text := '';
    qty_cond text := '';
    cmd text := '';
BEGIN
    IF book_title != '' THEN
        title_cond := format(
            ' AND cv.title ILIKE %L', '%' || book_title || '%'
        );
    END IF;
    IF author_name != '' THEN
        author_cond := format(
            ' AND cv.authors ILIKE %L', '%' || author_name || '%'
        );
    END IF;
    IF in_stock THEN
        qty_cond := ' AND cv.onhand_qty > 0';
    END IF;
    cmd := '
        SELECT cv.book_id,
               cv.display_name,
               cv.onhand_qty
        FROM   catalog_v cv
        WHERE  true'
        || title_cond || author_cond || qty_cond || '
        ORDER BY display_name';

    RAISE LOG 'DEBUG get_catalog (%, %, %): %',
        author_name, book_title, in_stock, cmd;
    RETURN QUERY EXECUTE cmd;
END
$$ STABLE LANGUAGE plpgsql;

CREATE FUNCTION
```

2. Enabling and Disabling SQL Query Tracing

To enable tracing of all queries at the server level, run:

```
=> ALTER SYSTEM SET log_min_duration_statement = 0;
```

ALTER SYSTEM

```
=> SELECT pg_reload_conf();
```

```
pg_reload_conf
-----
t
(1 row)
```

To disable:

```
=> ALTER SYSTEM RESET log_min_duration_statement;
```

ALTER SYSTEM

```
=> SELECT pg_reload_conf();
```



```
pg_reload_conf
```

```
-----
```

```
t
```

```
(1 row)
```

The last two commands got into the message log:

```
student$ tail -n 6 /var/log/postgresql/postgresql-16-main.log
```

```
2025-11-27 14:30:45.656 MSK [72018] LOG:  received SIGHUP, reloading configuration files
```

```
2025-11-27 14:30:45.657 MSK [72018] LOG:  parameter "log_min_duration_statement" changed  
to "0"
```

```
2025-11-27 14:30:45.791 MSK [75185] student@bookstore LOG:  duration: 9.576 ms
```

```
statement: ALTER SYSTEM RESET log_min_duration_statement;
```

```
2025-11-27 14:30:45.836 MSK [75185] student@bookstore LOG:  duration: 0.099 ms
```

```
statement: SELECT pg_reload_conf();
```

```
2025-11-27 14:30:45.837 MSK [72018] LOG:  received SIGHUP, reloading configuration files
```

```
2025-11-27 14:30:45.838 MSK [72018] LOG:  parameter "log_min_duration_statement" removed  
from configuration file, reset to default
```

1. Enable tracing of the PL/pgSQL code using the `plpgsql_check` extension; check how it works on the example of several routines that call each other.
2. When getting debug messages from the PL/pgSQL code, it is convenient to know the exact routine they are related to. In the demo, the function name was entered manually. Implement the functionality that automatically adds the name of the current function or procedure to the message.

1. To enable tracing, load the `plpgsql_check` extension into the session memory using the `LOAD` command, and then set both the `plpgsql_check.enable_tracer` and `plpgsql_check.tracer` parameters to *on* at the session level.

2. You can get the routine name by parsing the call stack. Use the results of Task 3 that you have completed as part of the practice for the Error Handling lesson.

1. Tracing with plpgsql_check

```
=> CREATE DATABASE plpgsql_debug;
```

CREATE DATABASE

```
=> \c plpgsql_debug
```

You are now connected to database "plpgsql_debug" as user "student".

Download the extension (no need to install it with the CREATE EXTENSION command in this case):

```
=> LOAD 'plpgsql_check';
```

LOAD

Enabling tracing:

```
=> SET plpgsql_check.enable_tracer = on;
```

SET

```
=> SET plpgsql_check.tracer = on;
```

SET

Multiple functions calling each other:

```
=> CREATE FUNCTION foo(n integer) RETURNS integer
AS $$
BEGIN
    RETURN bar(n-1);
END
$$ LANGUAGE plpgsql;
```

CREATE FUNCTION

```
=> CREATE FUNCTION bar(n integer) RETURNS integer
AS $$
BEGIN
    RETURN baz(n-1);
END
$$ LANGUAGE plpgsql;
```

CREATE FUNCTION

```
=> CREATE FUNCTION baz(n integer) RETURNS integer
AS $$
BEGIN
    RETURN n;
END
$$ LANGUAGE plpgsql;
```

CREATE FUNCTION

Example:

```
=> SELECT foo(3);
```

```
NOTICE: #0  -> start of function foo(integer) (oid=16387, tnl=1)
NOTICE: #0      "n" => '3'
NOTICE: #1  -> start of function bar(integer) (oid=16388, tnl=1)
NOTICE: #1      context: PL/pgSQL function foo(integer) line 3 at RETURN
NOTICE: #1      "n" => '2'
NOTICE: #2  -> start of function baz(integer) (oid=16389, tnl=1)
NOTICE: #2      context: PL/pgSQL function bar(integer) line 3 at RETURN
NOTICE: #2      "n" => '1'
NOTICE: #2  <<- end of function baz (elapsed time=0.030 ms)
NOTICE: #1  <<- end of function bar (elapsed time=0.159 ms)
NOTICE: #0  <<- end of function foo (elapsed time=0.392 ms)
foo
-----
 1
(1 row)
```

Not only the start and end events of the functions are displayed, but also the parameter values, as well as the time spent on execution (the extension can also do profiling, but we will not get into that here).

Disable tracing:

```
=> SET plpgsql_check.tracer = off;
```

```
SET
```

2. Function Names in the Debug Log

Create a procedure that returns the top of the call stack (with the exception of the tracing procedure itself). The message is displayed with indentation proportional to the depth of the stack.

```
=> CREATE PROCEDURE raise_msg(msg text)
AS $$
DECLARE
    ctx text;
    stack text[];
BEGIN
    GET DIAGNOSTICS ctx := PG_CONTEXT;
    stack := regexp_split_to_array(ctx, E'\n');
    RAISE NOTICE '?: %',
        repeat('. ', array_length(stack,1)-2) || stack[3], msg;
END
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

Example:

```
=> CREATE TABLE t(n integer);
```

```
CREATE TABLE
```

```
=> CREATE FUNCTION on_insert() RETURNS trigger
AS $$
BEGIN
    CALL raise_msg('NEW = ' || NEW::text);
    RETURN NEW;
END
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

```
=> CREATE TRIGGER t_before_row
BEFORE INSERT ON t
FOR EACH ROW
EXECUTE FUNCTION on_insert();
```

```
CREATE TRIGGER
```

```
=> CREATE PROCEDURE insert_into_t()
AS $$
BEGIN
    CALL raise_msg('start');
    INSERT INTO t SELECT id FROM generate_series(1,3) id;
    CALL raise_msg('end');
END
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

```
=> CALL insert_into_t();
```

```
NOTICE: . PL/pgSQL function insert_into_t() line 3 at CALL: start
NOTICE: . . . PL/pgSQL function on_insert() line 3 at CALL: NEW = (1)
NOTICE: . . . PL/pgSQL function on_insert() line 3 at CALL: NEW = (2)
NOTICE: . . . PL/pgSQL function on_insert() line 3 at CALL: NEW = (3)
NOTICE: . PL/pgSQL function insert_into_t() line 5 at CALL: end
CALL
```

```
=> \c postgres
```

You are now connected to database "postgres" as user "student".

```
=> DROP DATABASE plpgsql_debug;
```

```
DROP DATABASE
```