Authors: Egor Rogov, Pavel Luzanov
Translated by Liudmila Mantrova

Contact Us
Please send your feedback to: edu@postgrespro.ru

Agenda

Correctness checks

PL/pgSQL debugger

Debugging messages and their various implementations

Session tracing

Correctness Checks

Compile-time and run-time checks

plpgsql.extra_warnings
plpgsql.extra_errors

additional checks provided by the plpgsql_check extension

Built-in checks

the ASSERT command

Testing

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

But you can also avoid some particular error classes if you enable compile-time and run-time verification of source code. It is controlled by *plpgsql.extra_warnings* and *plpgsql.extra_errors* parameters, as explained in the “PL/pgSQL. Executing Queries” lecture.

https://postgrespro.com/docs/postgrespro/12/plpgsql-development-tips#PLPGSQL-EXTRA-CHECKS

That lecture also introduces the plpgsql_check extension, which 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 command.

https://postgrespro.com/docs/postgresql/12/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

Using the ASSERT command, you can specify conditions that, if broken, indicate an unexpected error. Such conditions are somewhat similar to integrity constraints in a database.

For example, here is a function that returns the number of the entrance given an apartment 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
```

You can check correctness by testing some corner cases:

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

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

But if the input values are invalid, the function will return a meaningless result, which, if passed further to other routines, can lead to their incorrect behavior as well. You won't catch these issues if you test a single function only.

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

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

You can provide protection against such cases by adding the following check:

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

```
CREATE FUNCTION
```

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

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

Now an invalid call will immediately result in an error.

PL/pgSQL Debugger

Postgres PROFESSIONAL

Interface

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

Features

- setting breakpoints
- step-by-step execution
- checking and setting variable values
- no need to modify the code
- debugging applications at run time

As its name suggests, PL/pgSQL Debugger is a debugging utility 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 at run time. 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 pgAdmin) 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

Debug Messages

Not only debugging

- monitoring long-running processes
- writing an application log

Implementation strategies

- directing debug output to the terminal or to the server log file
- saving messages in a table or in a file
- passing information to other processes

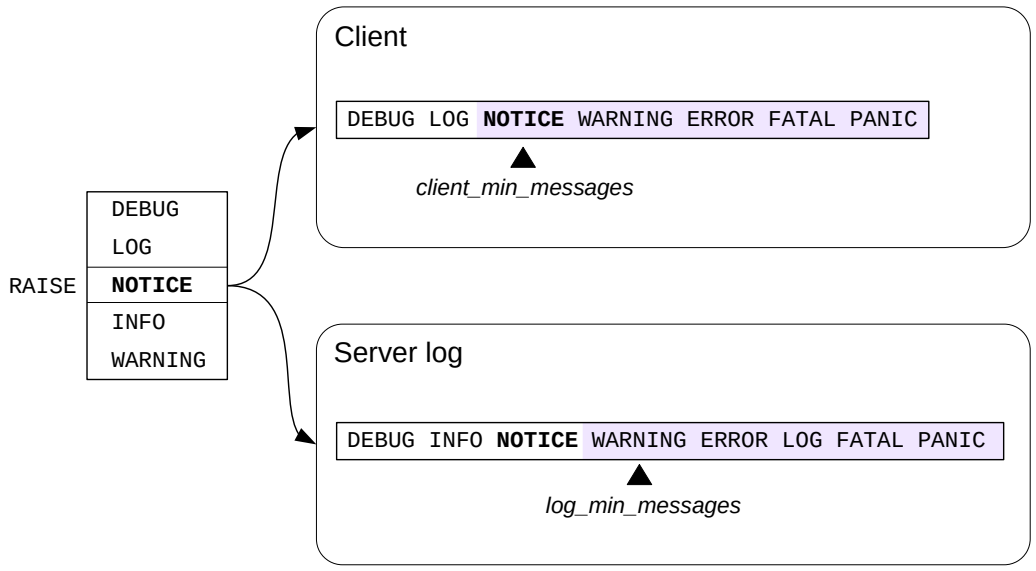
Another debugging approach consists in adding debug messages to the key parts of the code to provide the current context. As you analyze these messages, you can understand what exactly has gone wrong.

Apart from the debugging itself, debug messages can also perform other functions. They can indicate the execution stage of a long-running process. Similar to a database system, an application can write its own log. Having such a log with important data (e.g., report-related data: the name of the report, the user who has collected it, date, parameters, 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 Command



We are already familiar with the `RAISE` command. It can be used both to raise exceptions (which is discussed in detail in lecture “PL/pgSQL. Error Handling”) and to emit messages. Such messages can be either sent to the client or 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` command 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/12/plpgsql-errors-and-messages

The RAISE Command

Let's create a function that takes a table name as a parameter and calculates the number of rows in this table.

```
=> 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
```

To execute a command dynamically, it is better to save its text in a variable in advance. If an error occurs, you can check the contents of this variable.

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

```
NOTICE:  cmd: SELECT COUNT(*) FROM pg_class
 get_count
-----------
       395
(1 row)
```

The line that starts with "NOTICE" provides debug information.

RAISE can be used to track the execution of a long-running query.

Suppose there are three explicitly defined execution stages in the code, and we would like to know the exact stage we are at during routine execution.

```
=> CREATE PROCEDURE long_running()
AS $$
BEGIN
    RAISE NOTICE 'long_running. Stage 1/3...';
    PERFORM pg_sleep(2);

    RAISE NOTICE 'long_running. Stage 2/3...';
    PERFORM pg_sleep(3);

    RAISE NOTICE 'long_running. Stage 3/3...';
    PERFORM pg_sleep(1);

    RAISE NOTICE 'long_running. Done.';
END;
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

The RAISE command displays messages at once, without waiting until the function execution is complete:

```
=> 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 is convenient if the function can be called in a separate session. But if it is called from the application, it is easier to take a look at the server log.

Let's create the raise_msg procedure to produce a message of the level set in the user-defined app.raise_level parameter:

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

```
CREATE PROCEDURE
```

For the purposes of our example, let's set this parameter at the session level and make the long_running procedure use raise_msg:

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

```
SET
```

```
=> CREATE OR REPLACE PROCEDURE long_running()
AS $$
BEGIN
    CALL raise_msg('long_running. Stage 1/3...');
    PERFORM pg_sleep(2);

    CALL raise_msg('long_running. Stage 2/3...');
    PERFORM pg_sleep(3);

    CALL raise_msg('long_running. Stage 3/3...');
    PERFORM pg_sleep(1);

    CALL raise_msg('long_running. Done.');
END;
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

Now debug messages won't be displayed in normal circumstances (if app.raise_level = NONE):

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

```
CALL
```

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

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

```
SET
```

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

```
NOTICE:  student, 2021-10-19 17:03:51.338091+03, long_running. Stage 1/3...
NOTICE:  student, 2021-10-19 17:03:53.34025+03, long_running. Stage 2/3...
NOTICE:  student, 2021-10-19 17:03:56.344236+03, long_running. Stage 3/3...
NOTICE:  student, 2021-10-19 17:03:57.346025+03, long_running. Done.
CALL
```

To direct the application's debug messages into the server log, set the app.raise_level to LOG:

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

```
SET
```

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

```
CALL
```

Let's have a look at the server log:

```
student$ tail -n 20 /var/log/postgresql/postgresql-12-main.log | grep long_running
```

```
2021-10-19 17:03:57.401 MSK [29165] student@plpgsql_debug LOG:  student, 2021-10-19 17:03:57.40142+03, long_running. Stage 1/3...
        SQL statement "CALL raise_msg('long_running. Stage 1/3...')"
        PL/pgSQL function long_running() line 3 at CALL
2021-10-19 17:03:57.401 MSK [29165] student@plpgsql_debug STATEMENT:  CALL long_running();
2021-10-19 17:03:59.404 MSK [29165] student@plpgsql_debug LOG:  student, 2021-10-19 17:03:59.404887+03, long_running. Stage 2/3...
        SQL statement "CALL raise_msg('long_running. Stage 2/3...')"
        PL/pgSQL function long_running() line 6 at CALL
2021-10-19 17:03:59.404 MSK [29165] student@plpgsql_debug STATEMENT:  CALL long_running();
2021-10-19 17:04:02.406 MSK [29165] student@plpgsql_debug LOG:  student, 2021-10-19 17:04:02.405975+03, long_running. Stage 3/3...
        SQL statement "CALL raise_msg('long_running. Stage 3/3...')"
        PL/pgSQL function long_running() line 9 at CALL
2021-10-19 17:04:02.406 MSK [29165] student@plpgsql_debug STATEMENT:  CALL long_running();
2021-10-19 17:04:03.408 MSK [29165] student@plpgsql_debug LOG:  student, 2021-10-19 17:04:03.408104+03, long_running. Done.
        SQL statement "CALL raise_msg('long_running. Done.')"
        PL/pgSQL function long_running() line 12 at CALL
2021-10-19 17:04:03.408 MSK [29165] student@plpgsql_debug STATEMENT:  CALL long_running();
```

By modifying app.raise_level, log_min_messages, and client_min_messages parameters, you can switch between different modes of logging debug messages.

What is important, the application code remains the same.

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/postgrespro/12/runtime-config-logging#RUNTIME-CONFIG-LOGGING-WHAT

Session Status

Let's see how we can use the application_name parameter for debugging. The first session modifies this parameter, while the second one periodically polls the pg_stat_activity view.

Here is a new version of the procedure:

```
=> CREATE OR REPLACE PROCEDURE long_running()
AS $$
BEGIN
    SET LOCAL application_name TO "long_running. Stage 1/3...";
    PERFORM pg_sleep(2);

    SET LOCAL application_name TO "long_running. Stage 2/3...";
    PERFORM pg_sleep(3);

    SET LOCAL application_name TO "long_running. Stage 3/3...";
    PERFORM pg_sleep(1);

    SET LOCAL application_name TO "long_running. Done.";
END;
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

In the first session, run the following commands:

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

In the second session, refresh the row in the pg_stats_activity view every two seconds:

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

```
  pid  | usename |      application_name
-------+---------+---------------------------
 29165 | student | long_running. Stage 1/3...
(1 row)
```

```
=> \g
```

```
  pid  | usename |      application_name
-------+---------+---------------------------
 29165 | student | long_running. Stage 2/3...
(1 row)
```

```
=> \g
```

```
  pid  | usename |      application_name
-------+---------+---------------------------
 29165 | student | long_running. Stage 3/3...
(1 row)
```

```
=> \g
```

```
  pid  | usename | application_name
-------+---------+------------------
 29165 | student | psql
(1 row)
```

```
CALL
```

Process → Table

The dblink extension

- is part of the server
- incurs additional costs for opening a new connection

Autonomous transactions

- commercial distributions (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/postgrespro/12/dblink

Commercial distributions, such as Postgres Pro Enterprise, implement autonomous transactions, which incur lower overhead than dblink usage.

Writing Debug Messages into a Table: the dblink Extension

Install the extension:

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

```
CREATE EXTENSION
```

Create a table for logging messages.

In this table, it is useful to store insertion time and the name of the user who performed the operation. The id column is required to sort table rows in the order of their insertion.

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

```
CREATE TABLE
```

Create a procedure to automatically add new entries into the table log. The procedure opens a new session, inserts a row in a separate transaction, and closes the session.

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

```
CREATE PROCEDURE
```

Now create another version of the long_running procedure.

```
=> CREATE OR REPLACE PROCEDURE long_running()
AS $$
BEGIN
    CALL write_log('long_running. Stage 1/3...');
    PERFORM pg_sleep(2);

    CALL write_log('long_running. Stage 2/3...');
    PERFORM pg_sleep(3);

    CALL write_log('long_running. Stage 3/3...');
    PERFORM pg_sleep(1);

    CALL write_log('long_running. Done.');
END;
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

To check our implementation, let’s start the long_running procedure in a separate transaction that will be aborted at the very last moment.

```
=> BEGIN;
```

```
BEGIN
```

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

```
CALL
```

```
=> ROLLBACK;
```

```
ROLLBACK
```

Let’s make sure that all the calls of write_log are saved in the table. Using ts values, you can determine time intervals between the calls.

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

 username |    ts    |          message
----------+----------+---------------------------
 student  | 17:04:10 | long_running. Stage 1/3...
 student  | 17:04:12 | long_running. Stage 2/3...
 student  | 17:04:16 | long_running. Stage 3/3...
 student  | 17:04:17 | long_running. Done.
(4 rows)
```

Process → File

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—plperlu) that will perform the same task. Various server-side programming languages are covered in the DEV2 course.

https://postgrespro.com/docs/postgrespro/12/adminpack

Writing into a File: pg_file_write

Let's install the extension:

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

```
CREATE EXTENSION
```

Now let's create the write_file procedure that will be writing debug information into a file. The postgres user that has started the database instance must have the write access to this file, so let's save it in this user's home directory.

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

```
CREATE PROCEDURE
```

The function writes the message onto a separate line of the log file, together with the information about who and when has written this line.

Let's create another version of the long_running procedure.

```
=> CREATE OR REPLACE PROCEDURE long_running()
AS $$
BEGIN
    CALL write_file('long_running. Stage 1/3...');
    PERFORM pg_sleep(2);

    CALL write_file('long_running. Stage 2/3...');
    PERFORM pg_sleep(3);

    CALL write_file('long_running. Stage 3/3...');
    PERFORM pg_sleep(1);

    CALL write_file('long_running. Done.');
END;
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

To check our implementation, let's start long_running in a separate transaction that will be aborted at the very last moment.

```
=> BEGIN;
```

```
BEGIN
```

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

```
CALL
```

```
=> ROLLBACK;
```

```
ROLLBACK
```

Let's check that the messages have appeared in the log (on behalf of the postgres OS user):

```
postgres$ cat /var/lib/postgresql/log.txt
```

```
student, 2021-10-19 17:04:17.259374+03, long_running. Stage 1/3...
student, 2021-10-19 17:04:19.261617+03, long_running. Stage 2/3...
student, 2021-10-19 17:04:22.264959+03, long_running. Stage 3/3...
student, 2021-10-19 17:04:23.26653+03, long_running. Done.
```

To access this file on behalf of the student user, you have to use the sudo command.

Tracing SQL

Standard tracing into the log file

- logging overhead
- a big size of the log file
- profiling tools are required
- access to the log file is required (security)

Settings

long-running statements	*log_min_duration_statement*
the statements to log	*log_statement*
message context	*log_line_prefix*
…	

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 can 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/12/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 we have explained in lectures “Basic Tools. Installation and Management, psql” and “Data Organization. Logical Structure”).

Tracing SQL

The auto_explain extension

- logging execution plans
- tracing nested statements

Settings

plans of long-running commands	*auto_explain.log_min_duration*
nested statements	*auto_explain.log_nested_statements*
…	

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 operators), 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/12/auto-explain

Tracing PL/pgSQL

The plpgsql_check extension

- overhead incurred by logging
- loads of returned data

The main settings

enabling tracing	*plpgsql_check.enable_tracer* *plpgsql_check.tracer*
message levels	*plpgsql_check.tracer_errlevel*

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 can be not that easy. There are no built-in features 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 lecture “PL/pgSQL. Executing Queries”).

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

https://github.com/okbob/plpgsql_check

Tracing Sessions

A simple example of tracing is setting the log_statement parameter to all (log all commands, including DDL commands, data modification operations, and queries).

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

```
SET
```

Let's run an arbitrary query...

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

```
NOTICE:  cmd: SELECT COUNT(*) FROM pg_views
 get_count
-----------
       124
(1 row)
```

...and disable tracing:

```
=> RESET log_statement;
```

```
RESET
```

The information about the executed commands appears in the server log:

```
student$ tail -n 2 /var/log/postgresql/postgresql-12-main.log
```

```
2021-10-19 17:04:23.405 MSK [29165] student@plpgsql_debug LOG:  statement: SELECT get_count('pg_views');
2021-10-19 17:04:23.458 MSK [29165] student@plpgsql_debug LOG:  statement: RESET log_statement;
```

However, the log contains only the top-level command; the query run within the get_count function is not there.

Let's use the auto_explain extension. You do not have to install this extension into the database, but you need to load it into memory. It can be done for the whole instance using the shared_preload_libraries parameter, or for the current process only:

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

```
LOAD
```

Enable tracing for all commands, regardless of their execution time:

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

```
SET
```

Enable tracing of nested statements:

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

```
SET
```

The messages are displayed using the same mechanism that is employed by RAISE. By default, the LOG level is used, which usually directs the output to the log file. You can modify this parameter to display tracing information right in the terminal:

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

```
SET
```

Repeat the query:

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

```
NOTICE:  cmd: SELECT COUNT(*) FROM pg_views
NOTICE:  duration: 0.093 ms  plan:
Query Text: SELECT COUNT(*) FROM pg_views
Aggregate  (cost=18.25..18.26 rows=1 width=8)
  ->  Seq Scan on pg_class c  (cost=0.00..17.94 rows=124 width=0)
        Filter: (relkind = 'v'::"char")
NOTICE:  duration: 0.439 ms  plan:
Query Text: SELECT get_count('pg_views');
Result  (cost=0.00..0.26 rows=1 width=8)
 get_count
-----------
       124
(1 row)
```

We see both the function call and the nested query, together with the execution plans.

Summary

PL/pgSQL Debugger is a debugger API used in GUI IDEs

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

It is possible to trace a session

Practice

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.

Task 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 lecture “Basic Tools. Installation and Management, psql.” 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.

Task 1. get_catalog Function

Let's construct the text of the dynamic query in a separate variable, and write this variable into the server log before execution. To provide more detailed information, let's extend the debug output with parameter values passed to the function.

Debug messages can be found in the log by searching for the "DEBUG get_catalog" string.

You can delete or comment out the RAISE LOG command after debugging.

```
=> 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
```

Task 2. Enabling and Disabling Tracing of SQL Queries

To enable tracing of all queries at the server level, you can run the following commands:

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

```
ALTER SYSTEM
```

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

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

To disable tracing, run:

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

```
ALTER SYSTEM
```

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

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

The last two commands can be found in the server log:

```
student$ tail -n 6 /var/log/postgresql/postgresql-12-main.log
```

```
2021-10-19 17:06:17.658 MSK [35602] LOG:  received SIGHUP, reloading configuration files
2021-10-19 17:06:17.658 MSK [35602] LOG:  parameter "log_min_duration_statement" changed to "0"
2021-10-19 17:06:17.716 MSK [45424] student@bookstore LOG:  duration: 6.186 ms  statement: ALTER SYSTEM RESET log_min_duration_statement;
2021-10-19 17:06:17.737 MSK [45424] student@bookstore LOG:  duration: 0.106 ms  statement: SELECT pg_reload_conf();
2021-10-19 17:06:17.737 MSK [35602] LOG:  received SIGHUP, reloading configuration files
2021-10-19 17:06:17.737 MSK [35602] LOG:  parameter "log_min_duration_statement" removed from configuration file, reset to default
```

Practice

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.

Task 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.tracers* parameters to “on” at the session level.

Task 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” lecture.

Task 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".
```

Load the extension (in this particular case, you do not have to install it into the database using the CREATE EXTENSION command):

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

```
LOAD
```

Enable tracing:

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

```
SET
```

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

```
SET
```

Create several functions that call 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
```

A tracing example:

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

```
NOTICE:  #0   ->> start of function foo(integer) (oid=24840)
NOTICE:  #0       "n" => '3'
NOTICE:  #1     ->> start of function bar(integer) (oid=24841)
NOTICE:  #1           call by foo(integer) line 3 at RETURN
NOTICE:  #1         "n" => '2'
NOTICE:  #2       ->> start of function baz(integer) (oid=24842)
NOTICE:  #2             call by bar(integer) line 3 at RETURN
NOTICE:  #2           "n" => '1'
NOTICE:  #2       <<- end of function baz (elapsed time=0.018 ms)
NOTICE:  #1     <<- end of function bar (elapsed time=0.106 ms)
NOTICE:  #0   <<- end of function foo (elapsed time=0.437 ms)
 foo
-----
   1
(1 row)
```

In addition to function start and end events, parameter values and elapsed time are displayed (the extension also provides profiling features, but we won't cover them here).

Disable tracing:

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

```
SET
```

Task 2. Including a Function Name into Debug Messages

Let's create a procedure that displays the upper part of the call stack (excluding the tracing procedure itself). The message is displayed with an indent that indicates the stack depth.

```
=> 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
```

A tracing 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
```