

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов, Игорь Гнатюк
Фото: Олег Бартунов (монастырь Пху и пик Бхрикути, Непал)



**Обратная связь**
Отзывы, замечания и предложения направляйте по адресу:
edu@postgrespro.ru

# Темы

Проверки корректности

Отладчик PL/pgSQL

Служебные сообщения и различные способы их реализации

Трассировка сеансов

# Проверки корректности

Проверки времени компиляции и времени выполнения

- *plpgsql.extra_warnings*
- *plpgsql.extra_errors*
- дополнительные проверки в расширении plpgsql_check

Проверки в коде

- команда ASSERT

Тестирование



Отладка предполагает выполнение программы и изучение возникающих проблем, обычно с помощью специализированного отладчика или с помощью вывода отладочных сообщений.

Но можно заранее исключить определенные классы ошибок, включив проверку исходного кода во время компиляции и выполнения. Это управляется параметрами *plpgsql.extra_warnings* и *plpgsql.extra_errors*, как рассматривалось в теме «PL/pgSQL. Выполнение запросов».

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

Там же рассматривалось применение расширения plpgsql_check, которое обеспечивает более широкий спектр проверок.

Еще одна возможность обезопасить свой код — добавить в него проверки условий, которые всегда должны выполняться (т. н. «sanity check»). Удобный способ для этого — SQL-команда ASSERT.

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

Нельзя не сказать и о важности тестирования кода. Тестирование не только позволяет изначально убедиться, что код работает так, как задумано, но и облегчает его дальнейшую модификацию — дает уверенность в том, что изменения не поломали существующий функционал. Мы не будем затрагивать эту тему; отметим только, что тестирование кода, работающего с базой данных, может оказаться непростым делом из-за необходимости подготовки тестовых примеров.

### Проверки корректности

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

```
CREATE DATABASE
```

```
=> \c plpgsql_debug
```

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

Команда ASSERT позволяет указать условия, нарушения которых являются непредвиденной ошибкой. Можно провести определенную аналогию между такими условиями и ограничениями целостности в базе данных.

Пример: функция, возвращающее номер подъезда по номеру квартиры:

```
=> 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
```

Убедиться в правильности работы функции можно при помощи тестирования, проверив результат на некоторых «интересных» значениях:

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

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

Но при некорректных входных значениях функция будет выдавать бессмысленный результат, который, например, может быть передан дальше в другие подпрограммы, которые из-за этого тоже могут повести себя некорректно. Тестирование только кода данной функции здесь никак не поможет.

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

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

Можно обезопасить себя, добавив проверку:

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

```
CREATE FUNCTION
```

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

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

Теперь некорректный вызов сразу же приведет к ошибке.

# PL/pgSQL Debugger

Состав

- расширение pldbgapi
- встроенная поддержка в некоторых графических средах

Возможности

- установка точек прерывания
- пошаговое выполнение
- проверка и установка значений переменных
- не требуется изменение кода
- отладка работающих приложений



PL/pgSQL Debugger — это отладчик для PL/pgSQL. Представляет собой расширение pldbgapi, которое официально поддерживается разработчиками PostgreSQL.

Расширение pldbgapi — это набор интерфейсных функций для сервера PostgreSQL, которые позволяют устанавливать точки прерывания, пошагово выполнять код программ, проверять и устанавливать значения переменных.

Исходный код отлаживаемых программ изменять не требуется, что дает возможность выполнять отладку работающих приложений. То есть процесс отладки не обязательно запускать отдельно, можно подключиться к уже работающему сеансу и начать его отладку.

Непосредственно пользоваться этими функциями неудобно, они в первую очередь предназначены для вызова из кода графических сред разработки. Некоторые из таких сред (включая pgAdmin) имеют встроенный интерфейс для отладки. Но чтобы им воспользоваться, сначала все равно придется установить расширение pldbgapi в соответствующую базу данных.

Исходный код отладчика доступен по ссылке:

https://github.com/EnterpriseDB/pldebugger

### Отладка с PL/pgSQL Debugger

Выполним отладку одной из функций нашего приложения.

Пакет поддержки отладчика postgresql-16-pldebugger уже установлен в виртуальной машине курса. Теперь нам нужно обеспечить загрузку разделяемой библиотеки, для этого установим параметр и перезапустим сервер:

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

```
ALTER SYSTEM
```

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

Установим расширение отладчика в базу данных bookstore:

```
student$ psql bookstore
```

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

```
CREATE EXTENSION
```

Для начала отладочных действий все готово. Запускаем приложение:

```
student$ xdg-open http://localhost
```

Отладку проведем с помощью графической утилиты pgAdmin 4.

```
student$ /usr/pgadmin4/bin/pgadmin4 2>/dev/null
```

## Служебные сообщения

Не только отладка кода

- мониторинг долго выполняющихся процессов
- ведение журнала приложения

Подходы к реализации

- вывод на консоль или в журнал сервера
- запись в таблицу или в файл
- передача информации другим процессам



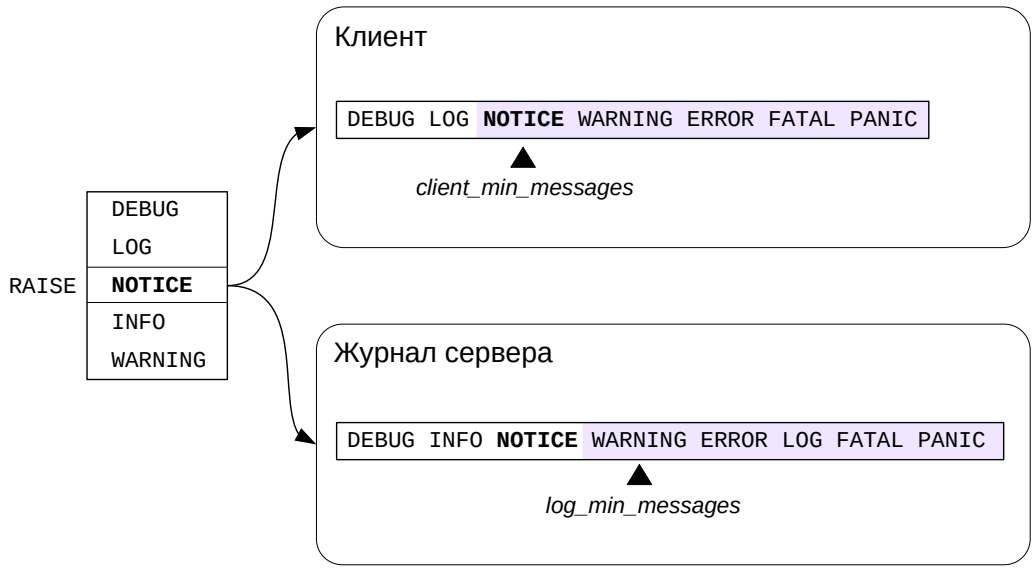
Второй способ отладки предполагает добавление в важные места кода служебных сообщений, выводящих информацию о текущем контексте. Анализируя сообщения, можно понять, что именно пошло не так.

Помимо собственно отладки, служебные сообщения могут выполнять и другие функции. Сообщения помогут определить, на каком этапе выполнения находится долго выполняющийся процесс. По аналогии с журналом сообщений СУБД приложение может записывать в свой собственный журнал важную информацию. Например, данные о запуске отчетов: название отчета, пользователь, когда и с какими параметрами запускался и т. д. Такая информация может сильно облегчить работу специалистов поддержки.

Можно выделить несколько подходов к реализации служебных сообщений в PL/pgSQL. Помимо уже знакомой команды RAISE, позволяющей выводить сообщения на консоль (а также в журнал сообщений сервера), можно отправлять сообщения другому процессу, записывать сообщения в таблицу или в файл.

При выборе того или иного подхода нужно обращать внимание на многие нюансы. Являются ли сообщения транзакционными (отправляются не дожидаясь окончания транзакции или только при ее завершении)? Можно ли одновременно использовать в нескольких сеансах? Как организовать доступ к журналу сообщений и как очищать его от старых записей? Какое влияние оказывают журналирование на производительность? Требуется ли модификация исходного кода?

# Команда RAISE






Мы уже встречались с командой RAISE. Она служит и для вызова исключительных ситуаций, что подробно рассматривалось в теме «PL/pgSQL. Обработка ошибок», и для отправки сообщений. Причем сообщения можно не только отправлять клиенту, но и записывать в журнал сервера.

В простом случае для отладки нужно добавить вызовы RAISE NOTICE в код функции, запустить функцию на выполнение (например в сеансе psql) и проанализировать получаемые по ходу выполнения сообщения. Сообщения RAISE нетранзакционные: они отправляются асинхронно и не зависят от статуса завершения транзакции.

Для управления отправкой сообщений используются уровень сообщения (DEBUG, LOG, NOTICE, INFO, WARNING) и параметры сервера. Он значений параметров зависит, будет ли сообщение отправлено клиенту (*client_min_messages*) и/или записано в журнал сервера (*log_min_messages*). Сообщение будет отправлено, если уровень команды RAISE равен значению соответствующего параметра или больше его (находится правее на рисунке).

Значения параметров по умолчанию настроены так, что сообщения с уровнем NOTICE отправляются только клиенту, с уровнем LOG — только в журнал, а с уровнем WARNING — и клиенту, и в журнал.

Сообщения с уровнем INFO всегда отправляются клиенту, их нельзя перехватить параметром *client_min_messages*.

https://postgrespro.ru/docs/postgresql/16/plpgsql-errors-and-messages

### Команда RAISE

Создадим функцию для подсчета количества строк в таблице, имя которой передается во входном параметре.

```
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
```

Для динамического выполнения текст команды лучше предварительно записывать в переменную. В случае ошибки можно проанализировать значение этой переменной.

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

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

Строка, начинающаяся с «NOTICE» — наша отладочная информация.

---

RAISE можно использовать для отслеживания хода выполнения долгого запроса.

Предположим, что внутри кода, решающего некую задачу, четко выделяются три этапа выполнения. И по ходу работы подпрограммы мы хотим понимать, на каком из них находимся. Для этого на каждом этапе будем вызывать такую процедуру:

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

```
CREATE PROCEDURE
```

Основная процедура по завершении очередного этапа работы вызывает процедуру 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
```

Команда RAISE выдает сообщения сразу, а не по окончании работы подпрограммы:

```
=> 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
```

---

Такой подход удобен, когда можно вызвать функцию в отдельном сеансе. Если же функция вызывается из

приложения, то удобнее писать и затем смотреть в журнал сервера.

Перепишем процедуру debug_message для выдачи сообщения с уровнем, установленным в пользовательском параметре app.raise_level:

```
=> CREATE OR REPLACE PROCEDURE debug_message(msg text)
AS $$
BEGIN
    CASE current_setting('app.raise_level', true)
        WHEN 'NOTICE'  THEN RAISE NOTICE  '%, %, %', user, clock_timestamp(), msg;
        WHEN 'DEBUG'   THEN RAISE DEBUG   '%, %, %', user, clock_timestamp(), msg;
        WHEN 'LOG'     THEN RAISE LOG     '%, %, %', user, clock_timestamp(), msg;
        WHEN 'INFO'    THEN RAISE INFO    '%, %, %', user, clock_timestamp(), msg;
        WHEN 'WARNING' THEN RAISE WARNING '%, %, %', user, clock_timestamp(), msg;
        ELSE NULL; -- все прочие значения отключают вывод сообщений
    END CASE;
END
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

Для целей примера установим параметр на уровне сеанса:

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

```
SET
```

Теперь в «обычной» жизни (app.raise_level = NONE) отладочные сообщения не будут выдаваться:

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

```
CALL
```

Запуская функцию в отдельном сеансе, мы можем получить отладочные сообщения, выставив app.raise_level в NOTICE:

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

```
SET
```

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

```
NOTICE:  student, 2024-01-10 20:03:20.407379+03, long_running. Stage 1/3
NOTICE:  student, 2024-01-10 20:03:22.410281+03, long_running. Stage 2/3
NOTICE:  student, 2024-01-10 20:03:25.413033+03, long_running. Stage 3/3
NOTICE:  student, 2024-01-10 20:03:26.414919+03, long_running. Done
CALL
```

Если же мы хотим включить отладку в приложении с записью в журнал сервера, то переключаем app.raise_level в LOG:

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

```
SET
```

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

```
CALL
```

Смотрим в журнал сервера:

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

```
2024-01-10 20:03:26.504 MSK [45623] student@plpgsql_debug LOG:  student, 2024-01-10
20:03:26.504088+03, long_running. Stage 1/3
        SQL statement "CALL debug_message('long_running. Stage 1/3')"
2024-01-10 20:03:28.504 MSK [45623] student@plpgsql_debug LOG:  student, 2024-01-10
20:03:28.504593+03, long_running. Stage 2/3
        SQL statement "CALL debug_message('long_running. Stage 2/3')"
2024-01-10 20:03:31.508 MSK [45623] student@plpgsql_debug LOG:  student, 2024-01-10
20:03:31.508215+03, long_running. Stage 3/3
        SQL statement "CALL debug_message('long_running. Stage 3/3')"
2024-01-10 20:03:32.509 MSK [45623] student@plpgsql_debug LOG:  student, 2024-01-10
20:03:32.509706+03, long_running. Done
        SQL statement "CALL debug_message('long_running. Done')"
```

Управляя параметрами app.raise_level, log_min_messages и client_min_messages, можно добиться различного поведения при выводе отладочных сообщений.

Важно, что для этого не нужно менять код приложения.

# Процесс → процесс (IPC)

NOTIFY → LISTEN

- команды SQL
- транзакционное выполнение, неудобно для отладки

Статус сеанса

- параметр *application_name*
- виден в представлении pg_stat_activity и выводе команды ps
- можно использовать в журнальных сообщениях



Серверные процессы в PostgreSQL могут обмениваться информацией между собой.

Среди встроенных решений можно отметить следующие.

- Использование команд NOTIFY для отправки сообщений в одном процессе и LISTEN для получения в другом.Но эти команды являются транзакционными, поэтому они неудобны для отладки:
  1. Сообщения отправляются только при фиксации транзакции, а не сразу после выполнения NOTIFY. Поэтому невозможно следить за ходом процесса.
  2. Если транзакция завершится неуспешно, то сообщения вообще не будут отправлены.
- Использование параметра *application_name*.
  Сеанс с долго выполняющимся процессом может периодически записывать статус выполнения в *application_name*. В другом сеансе администратор может опрашивать представление pg_stat_activity, содержащем подробную информацию о всех выполняющихся сеансах. Также значение *application_name* обычно видно в выводе команды ps.
  Значение *application_name* также можно записывать в журнал сервера (настраивая параметр *log_line_prefix*). Это облегчит поиск нужных строк в журнале.

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

### Статус сеанса

Посмотрим, как использовать для отладки параметр application_name. Первый сеанс меняет значение этого параметра, второй — периодически опрашивает представление pg_stat_activity.

Новый вариант процедуры:

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

```
CREATE PROCEDURE
```

Запускаем в первом сеансе:

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

Во втором с паузой в 2 секунды обновляем строку из pg_stat_activity:

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

```
    pid  | usename |                application_name
---------+---------+------------------------------------------------
 45623 | student | long_running. Stage long_running. Stage 1/3...
(1 row)
```

```
  => \g
```

```
    pid  | usename |                application_name
---------+---------+------------------------------------------------
 45623 | student | long_running. Stage long_running. Stage 2/3...
(1 row)
```

```
  => \g
```

```
    pid  | usename |                application_name
---------+---------+------------------------------------------------
 45623 | student | long_running. Stage long_running. Stage 3/3...
(1 row)
```

```
  => \g
```

```
    pid  | usename | application_name
---------+---------+------------------
 45623 | student | psql
(1 row)
```

```
CALL
```

# Процесс → таблица

Расширение dblink

- входит в состав сервера
- накладные расходы на создание соединения

Автономные транзакции

- Postgres Pro Enterprise



Еще один способ сохранения отладочных сообщений — запись в таблицу базы данных.

К плюсам данного подхода относится то, что параллельная работа и доступ журналу обеспечиваются средствами самой СУБД.

Однако нужно позаботиться о том, чтобы вставка в таблицу была нетранзакционной. Для этого можно использовать расширение dblink, идущее в составе сервера PostgreSQL. Это расширение позволяет открыть новое соединение к той же самой базе данных, поэтому вставка в таблицу выполняется в другой — независимой — транзакции.

К минусам данного подхода относится то, что открытие нового соединения требует дополнительных ресурсов сервера.

Подробнее работа с dblink рассматривается в курсе DEV2.

https://postgrespro.ru/docs/postgresql/16/dblink

В Postgres Pro Enterprise доступен механизм автономных транзакций, не имеющий таких существенных накладных расходов, которые характерны для dblink (см. курс PGPRO «Возможности Postgres Pro Enterprise»).

### Запись в таблицу: расширение dblink

Установим расширение:

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

```
CREATE EXTENSION
```

Создаем таблицу для записи сообщений.

В таблице полезно сохранять информацию о пользователе и времени вставки. Столбец id нужен для гарантированной сортировки результата в порядке добавления строк.

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

```
CREATE TABLE
```

Теперь перепишем процедуру так, чтобы она добавляла записи в таблицу log. Процедура открывает новый сеанс, выполняет вставку в отдельной транзакции и закрывает сеанс.

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

```
CREATE PROCEDURE
```

Для проверки запустим процедуру long_running в отдельной транзакции, которую в конце откатим.

```
=> BEGIN;
```

```
BEGIN
```

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

```
CALL
```

```
=> ROLLBACK;
```

```
ROLLBACK
```

Убедимся, что в таблице сохранились все вызовы write_log. По значениям ts можно проверить, сколько времени прошло между вызовами.

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

```
 username |    ts    |         message
----------+----------+-------------------------
 student  | 20:03:40 | long_running. Stage 1/3
 student  | 20:03:42 | long_running. Stage 2/3
 student  | 20:03:45 | long_running. Stage 3/3
 student  | 20:03:46 | long_running. Done
(4 rows)
```

# Процесс → файл

Расширение adminpack

- входит в состав сервера
- в том числе позволяет записывать текстовые файлы

Недоверенные языки

- например, PL/Perl



Записывать сообщения можно и в файл операционной системы.

Для этого, например, можно использовать расширение adminpack, позволяющее записывать данные в любой файл, доступ к которому есть у пользователя ОС postgres.

Другой вариант — написать функцию на недоверенном языке (таком, как PL/Perl — plperlu), которая будет выполнять ту же задачу. Различные языки серверного программирования рассматриваются в курсе DEV2.

https://postgrespro.ru/docs/postgresql/16/adminpack

### Запись в файл: pg_file_write

Установим расширение:

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

```
CREATE EXTENSION
```

В очередной раз перепишем процедуру debug_message, теперь она будет записывать отладочную информацию в файл. Пользователь postgres, запустивший экземпляр СУБД, должен иметь возможность записи в этот файл, поэтому расположим его в домашнем каталоге этого пользователя.

```
=> 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
```

Функция записывает отдельной строкой в файл журнала сообщение, переданное параметром, вместе с информацией о том, кто и когда записал строку.

---

Для проверки запустим long_running в отдельной транзакции, которую в конце откатим.

```
=> BEGIN;
```

```
BEGIN
```

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

```
CALL
```

```
=> ROLLBACK;
```

```
ROLLBACK
```

Проверим, что записи в журнале появились. Чтобы пользователь student мог получить доступ к этому файлу, нужно использовать команду sudo:

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

```
student, 2024-01-10 20:03:47.732497+03, long_running. Stage 1/3
student, 2024-01-10 20:03:49.735254+03, long_running. Stage 2/3
student, 2024-01-10 20:03:52.737086+03, long_running. Stage 3/3
student, 2024-01-10 20:03:53.739024+03, long_running. Done
```

# Трассировка SQL

## Стандартная трассировка в журнал сообщений

- накладные расходы на запись в журнал
- большой размер файла журнала
- требуются инструменты для анализа
- нужен доступ к журналу (безопасность)

## Настройки

| | |
|---|---|
| долго выполняющиеся команды | *log_min_duration_statement* |
| какие команды записывать | *log_statement* |
| контекст сообщения | *log_line_prefix* |
| … | |



В некоторых случаях может оказаться полезной трассировка всего происходящего в коде. Штатными средствами можно получить в журнале сообщений сервера SQL-запросы сеансов. Надо учитывать:

- Высоконагруженное приложение может выполнять огромное число запросов. Их запись в файл журнала может оказывать влияни на производительность подсистемы ввода-вывода.
- В большинстве случаев требуются специальные инструменты для анализа такого объема данных. Стандарт де-факто — pgBadger. https://github.com/darold/pgbadger
- К файлу журнала на сервере может не быть доступа у разработчиков приложений. К тому же в журнале производственной системы могут оказаться команды, содержащие конфиденциальную информацию.

Для настройки трассировки имеется ряд параметров, основные из которых приведены на слайде. Полный список:

https://postgrespro.ru/docs/postgresql/16/runtime-config-logging

Необязательно устанавливать конфигурационные параметры для всего кластера. Их действие можно ограничить отдельными сеансами при помощи команд SET, ALTER DATABASE, ALTER ROLE (как рассказывалось в темах «Обзор базового инструментария. Установка и управление, psql» и «Организация данных. Логическая структура»).

# Трассировка SQL-команд

## Расширение auto_explain

- запись в журнал планов выполнения запросов
- трассировка вложенных запросов

## Настройки

| | |
|---|---|
| планы долгих команд | *auto_explain.log_min_duration* |
| вложенные запросы | *auto_explain.log_nested_statements* |
| … | |



При включении трассировки SQL-команды попадают в журнал в том виде, в каком они отправлены на сервер. Если была вызвана подпрограмма PL/pgSQL, то в журнал будет записан только вызов этой подпрограммы (например, оператор SELECT или CALL), но не будет тех команд, которые выполняются внутри подпрограммы.

Чтобы увидеть в журнале запросы не только верхнего уровня, но и вложенные, потребуется штатное расширение auto_explain.

Как следует из названия расширения, его основная задача — записывать в журнал не только текст команды, но и план ее выполнения. Это тоже может оказаться полезно, хотя относится не к трассировке как таковой, а к оптимизации запросов (которая рассматривается в курсе QPT).

https://postgrespro.ru/docs/postgresql/16/auto-explain

# Трассировка PL/pgSQL

## Расширение plpgsql_check

- накладные расходы на запись сообщений
- большой объем выдачи

## Основные настройки

| | |
|---|---|
| включение трассировки | *plpgsql_check.enable_tracer*<br>*plpgsql_check.tracer* |
| уровень сообщений | *plpgsql_check.tracer_errlevel* |



Чтобы разобраться по журналу, какой процедурный код выполнялся, придется сопоставить SQL-запросы с подпрограммами на PL/pgSQL, а это может оказаться непросто. Стандартной возможности для трассировки PL/pgSQL-кода не предусмотрено, но расширение Павла Стехуле plpgsql_check (которое упоминалось в теме «PL/pgSQL. Выполнение запросов») позволяет ее выполнить.

Подобная трассировка вызывает большие накладные расходы и должна использоваться только для отладки, но не в промышленной эксплуатации.

https://github.com/okbob/plpgsql_check

### Трассировка сеансов

Простой пример включения трассировки — установка параметра log_statement в значение all (записывать все команды, включая DDL, модификацию данных и запросы).

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

```
SET
```

Выполним какой-нибудь запрос:

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

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

И выключим трассировку:

```
=> RESET log_statement;
```

```
RESET
```

Информация о выполненных командах окажется в журнале сервера:

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

```
2024-01-10 20:03:53.915 MSK [45623] student@plpgsql_debug LOG:  statement: SELECT
get_count('pg_views');
2024-01-10 20:03:53.966 MSK [45623] student@plpgsql_debug LOG:  statement: RESET
log_statement;
```

Однако в журнал попадает только команда верхнего уровня, но не запрос внутри функции get_count.

Воспользуемся расширением auto_explain. Это расширение не нужно устанавливать в базу данных, но требуется загрузить в память. Загрузку можно настроить для всего экземпляра с помощью параметра shared_preload_libraries, либо проделать однократно для текущего процесса:

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

```
LOAD
```

Установим трассировку всех команд независимо от длительности выполнения:

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

```
SET
```

Включим трассировку вложенных запросов:

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

```
SET
```

Сообщения выводятся с помощью того же механизма, что использует команда RAISE. По умолчанию используется уровень LOG, что обычно соответствует выводу в журнал. Изменив параметр, можно получать трассировку непосредственно в консоли:

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

```
SET
```

Повторим запрос:

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

```
NOTICE:  cmd: SELECT COUNT(*) FROM pg_views
NOTICE:  duration: 0.073 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: 0.481 ms  plan:
Query Text: SELECT get_count('pg_views');
Result  (cost=0.00..0.26 rows=1 width=8)
 get_count
-----------
       141
(1 row)
```

Мы видим не только вызов функции, но и вложенный запрос вместе с планами выполнения.

# Итоги

PL/pgSQL Debugger — API отладчика, используется в графических средах разработки

Служебные сообщения — вывод на консоль, запись в журнал сообщений сервера, в таблицу или в файл, передача другим процессам

Возможность трассировки сеансов

# Практика

1. Измените функцию get_catalog так, чтобы динамически формируемый текст запроса записывался в журнал сообщений сервера.

   В приложении выполните несколько раз поиск, заполняя разные поля, и убедитесь, что команды SQL формируются правильно.

2. Включите трассировку команд SQL на уровне сервера.

   Поработайте в приложении и проверьте, какие команды попадают в журнал сообщений.

   Выключите трассировку.



2. Для включения трассировки установите значение параметра *log_min_duration_statement* в 0 и перечитайте конфигурацию.
В журнал будут записываться все команды и время их выполнения.

Проще всего это сделать командой ALTER SYSTEM SET. Другие способы рассматривались в теме «Обзор базового инструментария. Установка и управление, psql». Не забудьте перечитать измененный конфигурационный файл.

После просмотра журнала следует вернуть значение параметра *log_min_duration_statement* в значение по умолчанию (−1), чтобы отключить трассировку. Удобный способ — команда ALTER SYSTEM RESET.

## 1. Функция get_catalog

Текст динамического запроса формируем в отдельной переменной, которую перед выполнением запишем в журнал сервера. Для более полной информации включим в сообщение значения переданных в функцию параметров.

Отладочные строки в журнале можно найти по подстроке «DEBUG get_catalog».

После отладки команду RAISE LOG можно удалить или закомментировать.

```
=> 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. Включение и выключение трассировки SQL-запросов

Чтобы включить трассировку всех запросов на уровне сервера, можно выполнить:

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

```
ALTER SYSTEM
```

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

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

Чтобы выключить:

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

```
ALTER SYSTEM
```

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

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

Последние две команды попали в журнал сообщений:

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

```
2024-01-10 20:17:02.865 MSK [85964] LOG:  received SIGHUP, reloading configuration files
2024-01-10 20:17:02.865 MSK [85964] LOG:  parameter "log_min_duration_statement" changed
to "0"
2024-01-10 20:17:02.947 MSK [88741] student@bookstore LOG:  duration: 29.495 ms
statement: ALTER SYSTEM RESET log_min_duration_statement;
2024-01-10 20:17:02.968 MSK [88741] student@bookstore LOG:  duration: 0.094 ms
statement: SELECT pg_reload_conf();
2024-01-10 20:17:02.968 MSK [85964] LOG:  received SIGHUP, reloading configuration files
2024-01-10 20:17:02.968 MSK [85964] LOG:  parameter "log_min_duration_statement" removed
from configuration file, reset to default
```

# Практика

1. Включите трассировку PL/pgSQL-кода средствами расширения plpgsql_check и проверьте ее работу на примере нескольких подпрограмм, вызывающих одна другую.
2. При выводе отладочных сообщений из PL/pgSQL-кода важно понимать, к какой подпрограмме они относятся. В демонстрации имя функции выводилось вручную. Реализуйте функционал, автоматически добавляющий к тексту сообщений имя текущей функции или процедуры.



1. Для включения трассировки загрузите расширение plpgsql_check в память сеанса командой LOAD, затем установите в сеансе оба параметра *plpgsql_check.enable_tracer* и *plpgsql_check.tracer* в значение «on».

2. Имя подпрограммы можно получить, разобрав стек вызовов. Воспользуйтесь подходом из практического задания 3 к теме «Обработка ошибок».

## 1. Трассировка с помощью plpgsql_check

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

```
CREATE DATABASE
```

```
=> \c plpgsql_debug
```

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

Загрузим расширение (в данном случае устанавливать его в базу данных командой CREATE EXTENSION не нужно):

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

```
LOAD
```

Включим трассировку:

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

```
SET
```

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

```
SET
```

Несколько функций, вызывающих друг друга:

```
=> 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
```

Пример работы трассировки:

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

```
NOTICE:  #0   ->> start of function foo(integer) (oid=409613, tnl=1)
NOTICE:  #0       "n" => '3'
NOTICE:  #1     ->> start of function bar(integer) (oid=409614, 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=409615, 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=0.017 ms)
NOTICE:  #1     <<- end of function bar (elapsed time=0.094 ms)
NOTICE:  #0   <<- end of function foo (elapsed time=0.254 ms)
 foo
-----
   1
(1 row)
```

Выводятся не только события начала и окончания работы функций, но и значения параметров, а также затраченное на выполнение время (в расширении есть и возможность профилирования, которую мы не рассматриваем).

Выключим трассировку:

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

```
SET
```

## 2. Имя функции в отладочных сообщениях

Напишем процедуру, которая выводит верхушку стека вызовов (за исключением самой процедуры трассировки). Сообщение выводится с отступом, который соответствует глубине стека.

```
=> 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
```

Пример работы трассировки:

```
=> 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
```