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

Обработка ошибок в блоке PL/pgSQL

Имена и коды ошибок

Как происходит поиск обработчика

Накладные расходы на обработку ошибок

Обработка ошибок в блоке

Обработка выполняется, если есть секция EXCEPTION

Выполняется откат к точке сохранения в начале блока

- точка сохранения устанавливается неявно, если в блоке есть секция EXCEPTION

Если имеется обработчик, соответствующий ошибке

- выполняются команды обработчика
- блок завершается успешно

Если нет подходящего обработчика

- блок завершается ошибкой

Если внутри блока кода происходит ошибка времени выполнения, то обычно программа (блок, функция или процедура) аварийно прерывается, а текущая транзакция переходит в состояние сбоя: ее нельзя зафиксировать и можно только откатить.

Но ошибку можно перехватить и обработать. Для этого в блоке можно указать дополнительную секцию EXCEPTION, внутри которой перечислить условия, соответствующие ошибке, и операторы для обработки каждой такой ситуации.

EXCEPTION в целом соответствует конструкции try-catch в некоторых языках программирования (за исключением особенностей, связанных с обработкой транзакций, конечно).

В начале каждого блока с секцией EXCEPTION неявно устанавливается точка сохранения. Перед обработкой ошибки происходит откат к этой точке, в результате отменяются сделанные в блоке изменения, а также снимаются установленные в блоке блокировки.

Из-за наличия точки сохранения в блоках процедур с EXCEPTION запрещено использовать команды COMMIT и ROLLBACK. Зато, хотя команды SAVEPOINT и ROLLBACK TO SAVEPOINT не работают в PL/pgSQL, точку сохранения и откат к ней все-таки можно использовать и в процедурах, и в функциях — неявно.

https://postgrespro.ru/docs/postgresql/16/plpgsql-control-structures#PLPGSQL-ERROR-TRAPPING

Обработка ошибок в блоке

Рассмотрим простой пример.

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

```
CREATE TABLE
```

```
=> INSERT INTO t(id) VALUES (1);
```

```
INSERT 0 1
```

Когда нет ошибок, все операторы блока выполняются обычным образом:

```
=> DO $$
DECLARE
    n integer;
BEGIN
    SELECT id INTO STRICT n FROM t;
    RAISE NOTICE 'Оператор SELECT INTO выполнился, n = %', n;
END
$$;
```

```
NOTICE:  Оператор SELECT INTO выполнился, n = 1
DO
```

Теперь добавим в таблицу «лишнюю» строку, чтобы спровоцировать ошибку.

```
=> INSERT INTO t(id) VALUES (2);
```

```
INSERT 0 1
```

Если в блоке нет секции EXCEPTION, выполнение операторов в блоке прерывается и весь блок считается завершившимся с ошибкой:

```
=> DO $$
DECLARE
    n integer;
BEGIN
    SELECT id INTO STRICT n FROM t;
    RAISE NOTICE 'Оператор SELECT INTO выполнился, n = %', n;
END
$$;
```

```
ERROR:  query returned more than one row
HINT:  Make sure the query returns a single row, or use LIMIT 1.
CONTEXT:  PL/pgSQL function inline_code_block line 5 at SQL statement
```

Чтобы перехватить ошибку, в блоке нужна секция EXCEPTION, определяющая обработчик или несколько обработчиков.

Эта конструкция работает аналогично CASE: условия просматриваются сверху вниз, выбирается первая подходящая ветвь и выполняются ее операторы.

Что будет выведено?

```
=> DO $$
DECLARE
    n integer;
BEGIN
    n := 3;
    INSERT INTO t(id) VALUES (n);
    SELECT id INTO STRICT n FROM t;
    RAISE NOTICE 'Оператор SELECT INTO выполнился, n = %', n;
EXCEPTION
    WHEN no_data_found THEN
        RAISE NOTICE 'Нет данных';
    WHEN too_many_rows THEN
        RAISE NOTICE 'Слишком много данных';
        RAISE NOTICE 'Строк в таблице: %, n = %', (SELECT count(*) FROM t), n;
END
$$;
```

```
NOTICE:  Слишком много данных
NOTICE:  Строк в таблице: 2, n = 3
DO
```

Выполняется обработчик, соответствующий ошибке too_many_rows. Обратите внимание: после обработки в таблице остается 2 строки, так как перед выполнением обработчика произошел откат к неявной точке сохранения, которая устанавливается в начале блока.

Также заметьте, что локальная переменная функции сохранила то значение, которое было на момент возникновения ошибки.

Тонкий момент: если ошибка произойдет в секции DECLARE или в самом обработчике внутри EXCEPTION, то в этом блоке ее перехватить не получится.

```
=> DO $$
DECLARE
    n integer := 1 / 0; -- ошибка в этом месте не перехватывается
BEGIN
    RAISE NOTICE 'Все успешно';
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'Деление на ноль';
END
$$;
```

```
ERROR:  division by zero
CONTEXT:  SQL expression "1 / 0"
PL/pgSQL function inline_code_block line 3 during statement block local variable
initialization
```

Имена и коды ошибок

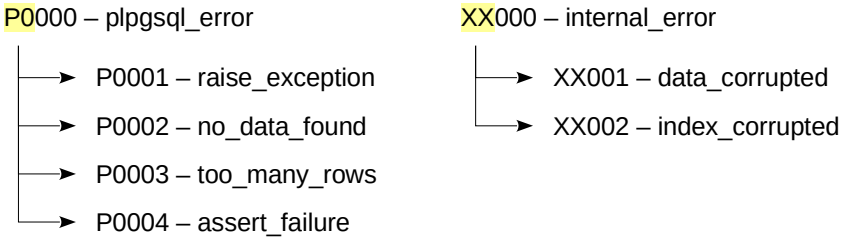
Информация об ошибке

имя

пятисимвольный код

дополнительные сведения: сообщение, детальное сообщение, подсказка, имена объектов, связанных с ошибкой

Двухуровневая иерархия



Каждой возможной ошибке соответствует имя и код (строка из пяти символов). В условии WHEN можно указывать как имя, так и код.

Ошибки организованы в своеобразную двухуровневую иерархию. Категория ошибки имеет код, заканчивающийся на три нуля, и соответствует любой ошибке с теми же первыми двумя символами.

Например, код 23000 соответствует категории, которая включает все ошибки, связанные с нарушением ограничений целостности (такие, как 23502 — нарушение ограничения NOT NULL или 23505 — нарушение уникальности).

Таким образом, кроме обычных ошибок можно использовать имя или код категории. Можно также использовать специальное имя OTHERS для того, чтобы перехватить любую ошибку (кроме самых фатальных).

Кроме имени и кода любая ошибка может иметь дополнительные сведения, облегчающие ее диагностику: сообщение, детальное сообщение, подсказку.

Все ошибки описаны в приложении А документации:

https://postgrespro.ru/docs/postgresql/16/errcodes-appendix

Ошибки можно не только перехватывать, но и программно вызывать.

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

Имена и коды ошибок

Имена ошибок мы уже видели, а для указания кода служит конструкция SQLSTATE.

В обработчике можно получить код ошибки и сообщение с помощью предопределенных переменных SQLSTATE и SQLERRM (вне блока EXCEPTION эти переменные не определены).

```
=> DO $$
DECLARE
    n integer;
BEGIN
    SELECT id INTO STRICT n FROM t;
EXCEPTION
    WHEN SQLSTATE 'P0003' OR no_data_found THEN -- можно несколько
        RAISE NOTICE '%: %', SQLSTATE, SQLERRM;
END
$$;
```

```
NOTICE:  P0003: query returned more than one row
DO
```

Какой обработчик будет использован?

```
=> DO $$
DECLARE
    n integer;
BEGIN
    SELECT id INTO STRICT n FROM t;
EXCEPTION
    WHEN no_data_found THEN
        RAISE NOTICE 'Нет данных. %: %', SQLSTATE, SQLERRM;
    WHEN plpgsql_error THEN
        RAISE NOTICE 'Другая ошибка. %: %', SQLSTATE, SQLERRM;
    WHEN too_many_rows THEN
        RAISE NOTICE 'Слишком много данных. %: %', SQLSTATE, SQLERRM;
END
$$;
```

```
NOTICE:  Другая ошибка. P0003: query returned more than one row
DO
```

Выбирается первый подходящий обработчик, в данном случае — plpgsql_error (напомним, что это не отдельная ошибка, а категория). До последнего обработчика дело никогда не дойдет.

Ошибку можно принудительно вызвать по ее коду или имени.

Здесь мы используем специальное имя others, соответствующее любой ошибке, которую можно перехватить (за исключением прерванного клиентом выполнения и нарушения отладочной проверки assert — их можно перехватить отдельно, но обычно это не имеет смысла).

```
=> DO $$
BEGIN
    RAISE no_data_found;
EXCEPTION
    WHEN others THEN
        RAISE NOTICE '%: %', SQLSTATE, SQLERRM;
END
$$;
```

```
NOTICE:  P0002: no_data_found
DO
```

При необходимости можно задействовать и пользовательские коды ошибок, отсутствующие в справочнике, а также указать некоторую дополнительную информацию (в примере — только часть из возможного):

```
=> DO $$
BEGIN
    RAISE SQLSTATE 'ERR01' USING
        message := 'Сбой матрицы',
        detail  := 'При выполнении произошел непоправимый сбой матрицы',
        hint := 'Обратитесь к системному администратору';
END
$$;
```

```
ERROR:  Сбой матрицы
DETAIL:  При выполнении произошел непоправимый сбой матрицы
HINT:  Обратитесь к системному администратору
CONTEXT:  PL/pgSQL function inline_code_block line 3 at RAISE
```

В обработчике эту информацию нельзя получить из переменных; если нужно проанализировать такие данные в коде, есть специальная конструкция:

```
=> DO $$
DECLARE
    message text;
    detail text;
    hint text;
BEGIN
    RAISE SQLSTATE 'ERR01' USING
        message := 'Сбой матрицы',
        detail  := 'При выполнении произошел непоправимый сбой матрицы',
        hint := 'Обратитесь к системному администратору';
EXCEPTION
    WHEN others THEN
        GET STACKED DIAGNOSTICS
            message := MESSAGE_TEXT,
            detail := PG_EXCEPTION_DETAIL,
            hint := PG_EXCEPTION_HINT;
        RAISE NOTICE E'\nmessage = %\ndetail = %\nhint = %',
            message, detail, hint;
END
$$;
```

```
NOTICE:
message = Сбой матрицы
detail = При выполнении произошел непоправимый сбой матрицы
hint = Обратитесь к системному администратору
DO
```

Поиск обработчика

Необработанная ошибка «поднимается» на уровень выше

- в объемлющий блок PL/pgSQL, если он есть
- в вызывающую подпрограмму, если она есть

Поиск обработчика определяется стеком вызовов

- то есть не определен статически, а зависит от выполнения программы

Никем не обработанная ошибка отправляется клиенту

- транзакция переходит в состояние сбоя, клиент должен ее откатить
- информация об ошибке записывается в журнал сообщений сервера

Если ни одно из условий, перечисленных в секции EXCEPTION, не сработает, то ошибка «поднимается» на уровень выше.

Если ошибка возникла в блоке, вложенном в другой блок, то сервер будет пытаться найти обработчик в объемлющем блоке. Если и там не найдется подходящего обработчика, весь объемлющий блок будет считаться ошибочным и ошибка поднимется еще выше, и так далее.

Если мы перебрали все объемлющие блоки и не обнаружили подходящего обработчика возникшей ошибки, ошибка «поднимается» на уровень той подпрограммы, которая вызвала наш блок. То есть, чтобы разобраться, в каком порядке будут просматриваться обработчики ошибки, надо проанализировать стек вызовов.

Если ни один из возможных обработчиков ошибки не сработает:

- сообщение об ошибке обычно попадает в журнал сообщений сервера (это зависит от настроек сервера, см. тему «PL/pgSQL. Отладка»);
- об ошибке сообщается клиенту, который инициировал вызов кода в базе данных. Клиент ставится перед фактом: транзакция переходит в состояние сбоя, и ее можно только откатить.

Как именно клиент будет обрабатывать ошибку, зависит от него самого. Например, psql выведет сообщение об ошибке и всю доступную диагностическую информацию. Клиент, ориентированный на конечного пользователя, может вывести знаменитое «обратитесь к системному администратору» и т. д.

Поиск обработчика

Рассмотрим несколько примеров поиска обработчика в случае вложенных блоков. Что будет выведено?

```
=> DO $$
BEGIN
    BEGIN
        SELECT 1/0;
        RAISE NOTICE 'Вложенный блок выполнен';
    EXCEPTION
        WHEN division_by_zero THEN
            RAISE NOTICE 'Ошибка во вложенном блоке';
    END;
    RAISE NOTICE 'Внешний блок выполнен';
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'Ошибка во внешнем блоке';
END
$$;
```

```
NOTICE:  Ошибка во вложенном блоке
NOTICE:  Внешний блок выполнен
DO
```

Ошибка обрабатывается в том блоке, в котором она возникла. Внешний блок выполняется так, как будто никакой ошибки не было.

А так?

```
=> DO $$
BEGIN
    BEGIN
        SELECT 1/0;
        RAISE NOTICE 'Вложенный блок выполнен';
    EXCEPTION
        WHEN no_data_found THEN
            RAISE NOTICE 'Ошибка во вложенном блоке';
    END;
    RAISE NOTICE 'Внешний блок выполнен';
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'Ошибка во внешнем блоке';
END
$$;
```

```
NOTICE:  Ошибка во внешнем блоке
DO
```

Обработчик во внутреннем блоке не подходит; блок завершается с ошибкой, которая обрабатывается уже во внешнем блоке.

Не забывайте, что в блоке с секцией EXCEPTION происходит откат к точке сохранения, неявно установленной в начале блока. В данном случае будут отменены все изменения, сделанные в обоих блоках.

А так?

```
=> DO $$
BEGIN
    BEGIN
        SELECT 1/0;
        RAISE NOTICE 'Вложенный блок выполнен';
    EXCEPTION
        WHEN no_data_found THEN
            RAISE NOTICE 'Ошибка во вложенном блоке';
    END;
    RAISE NOTICE 'Внешний блок выполнен';
EXCEPTION
    WHEN no_data_found THEN
        RAISE NOTICE 'Ошибка во внешнем блоке';
END
$$;
```

```
ERROR:  division by zero
CONTEXT:  SQL statement "SELECT 1/0"
PL/pgSQL function inline_code_block line 4 at SQL statement
```

Так не срабатывает ни один обработчик, и вся транзакция обрывается.

Обычно не нужно стремиться обработать все возможные ошибки в серверном коде. Нет ничего плохого в том, чтобы передать возникшую ошибку клиенту. В целом, обрабатывать ошибку имеет смысл на том уровне, на котором можно сделать что-то осмысленное в возникшей ситуации. Поэтому обрабатывать ошибку внутри базы данных стоит, когда можно что-то сделать именно на серверной стороне (например, повторить операцию при ошибке сериализации). Про журналирование сообщений об ошибках мы еще будем говорить в теме «PL/pgSQL. Отладка».

Теперь рассмотрим пример с подпрограммами.

```
=> CREATE PROCEDURE foo()
AS $$
BEGIN
    CALL bar();
END
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

```
=> CREATE PROCEDURE bar()
AS $$
BEGIN
    CALL baz();
END
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

```
=> CREATE PROCEDURE baz()
AS $$
BEGIN
    PERFORM 1 / 0;
END
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

Что произойдет при вызове?

```
=> CALL foo();
```

```
ERROR:  division by zero
CONTEXT:  SQL statement "SELECT 1 / 0"
PL/pgSQL function baz() line 3 at PERFORM
SQL statement "CALL baz()"
PL/pgSQL function bar() line 3 at CALL
SQL statement "CALL bar()"
PL/pgSQL function foo() line 3 at CALL
```

То, что мы видим в сообщении об ошибке — это стек вызовов: сверху вниз = изнутри наружу.

Заметьте, что в этом сообщении, как и во многих других, вместо слова procedure используется function.

В обработчике ошибки тоже можно получить доступ к стеку, правда, в виде одной строки:

```
=> CREATE OR REPLACE PROCEDURE bar()
AS $$
DECLARE
    msg text;
    ctx text;
BEGIN
    CALL baz();
EXCEPTION
    WHEN others THEN
        GET STACKED DIAGNOSTICS
             msg := MESSAGE_TEXT,
             ctx := PG_EXCEPTION_DETAIL;
        RAISE NOTICE E'\nОшибка: %\nСтек ошибки:\n%\n', msg, ctx;
END
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

Проверим:

```
=> CALL foo();
```

```
NOTICE:
Ошибка: division by zero
Стек ошибки:


CALL
```

Поскольку блок с секцией EXCEPTION устанавливает неявную точку сохранения, то и в этом блоке, и во всех блоках выше по стеку вызовов, процедуры лишаются возможности использовать команды COMMIT и ROLLBACK.

```
=> CREATE OR REPLACE PROCEDURE baz()
AS $$
BEGIN
    COMMIT;
END
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

```
=> CALL foo();
```

```
NOTICE:
Ошибка: invalid transaction termination
Стек ошибки:


CALL
```

Накладные расходы

Любой блок с секцией EXCEPTION выполняется медленнее

- из-за установки неявной точки сохранения

Дополнительные расходы при возникновении ошибки

- из-за отката к точке сохранения

Обработку ошибок можно и нужно использовать,
но не стоит употреблять без необходимости

- PL/pgSQL и так интерпретируется и использует SQL для вычисления выражений
- для многих задач эта скорость более чем удовлетворительна
- проблемы с производительностью обычно связаны с запросами, а не с кодом PL/pgSQL

Одно только указание секции EXCEPTION уже влечет за собой накладные расходы из-за того, что в начале блока будет неявно устанавливаться точка сохранения. А если ошибка действительно возникает, то накладные расходы еще более увеличиваются из-за того, что выполняется откат транзакции к точке сохранения.

Поэтому если есть простой способ обойтись без обработки исключений, лучше от нее отказаться. Не стоит строить логику работы программы на «жонглировании» исключениями.

Однако если обработка ошибок действительно нужна, не стоит сомневаться — обрабатывать ошибки можно и нужно несмотря на накладные расходы.

Во-первых, язык PL/pgSQL довольно медленный сам по себе из-за интерпретации и обращений к SQL для вычисления выражений.

Во-вторых, очень часто эта скорость вполне удовлетворительна. Да, можно и быстрее, если написать на C, но зачем?

И в-третьих, основные проблемы с производительностью, как правило, связаны со скоростью выполнения запросов из-за неправильно построенных планов, а вовсе не со скоростью работы процедурного кода (см. курс QPT «Оптимизация запросов»).

Так что если есть альтернативное решение, которое и проще, и быстрее — конечно лучше воспользоваться им.

Накладные расходы

Чтобы оценить накладные расходы, рассмотрим простой пример.

Пусть имеется таблица с текстовым полем, в которое пользователи заносят произвольные данные (обычно это признак неудачного дизайна, но иногда приходится). Нам нужно выделить числа в отдельный столбец числового типа.

```
=> CREATE TABLE data(comment text, n integer);
```

```
CREATE TABLE
```

```
=> INSERT INTO data(comment)
SELECT CASE
        WHEN random() < 0.01 THEN 'не число' --  1%
        ELSE (random()*1000)::integer::text  -- 99%
    END
FROM generate_series(1,1_000_000);
```

```
INSERT 0 1000000
```

Решим задачу с помощью обработки ошибок, возникающих при преобразовании текстовых данных к числу:

```
=> CREATE FUNCTION safe_to_integer_ex(s text) RETURNS integer
AS $$
BEGIN
    RETURN s::integer;
EXCEPTION
    WHEN invalid_text_representation THEN
        RETURN NULL;
END
$$ IMMUTABLE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Проверим:

```
=> \timing on
```

```
Timing is on.
```

```
=> UPDATE data SET n = safe_to_integer_ex(comment);
```

```
UPDATE 1000000
Time: 10499,120 ms (00:10,499)
```

```
=> \timing off
```

```
Timing is off.
```

```
=> SELECT count(*) FROM data WHERE n IS NOT NULL;
```

```
 count
--------
 989931
(1 row)
```

В другом варианте функции вместо обработки ошибки будем проверять формат с помощью регулярного выражения (слегка упрощенного):

```
=> CREATE FUNCTION safe_to_integer_re(s text) RETURNS integer
AS $$
BEGIN
    RETURN CASE
        WHEN s ~ '^\d+$' THEN s::integer
        ELSE NULL
    END;
END
$$ IMMUTABLE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Проверим этот вариант:

```
=> \timing on
```

```
Timing is on.
```

```
=> UPDATE data SET n = safe_to_integer_re(comment);
```

```
UPDATE 1000000
Time: 6548,002 ms (00:06,548)

=> \timing off

Timing is off.

=> SELECT count(*) FROM data WHERE n IS NOT NULL;

 count
--------
 989931
(1 row)
```

Получается заметно быстрее. В этом примере исключение срабатывало всего в 1% случаев. Чем чаще — тем больше будет дополнительных накладных расходов на откат к точке сохранения.

```
=> UPDATE data SET comment = 'не число'; -- 100%

UPDATE 1000000

=> \timing on

Timing is on.

=> UPDATE data SET n = safe_to_integer_ex(comment);

UPDATE 1000000
Time: 14898,737 ms (00:14,899)

=> \timing off

Timing is off.
```

Встречаются (и довольно часто) случаи, когда можно обойтись без обработки ошибок, если выбрать другие подходящие средства.

Задача: вернуть строку из справочника или NULL, если строки нет.

```
=> CREATE TABLE categories(code text UNIQUE, description text);

CREATE TABLE

=> INSERT INTO categories VALUES ('books','Книги'), ('discs','Диски');

INSERT 0 2
```

Функция с обработкой ошибки:

```
=> CREATE FUNCTION get_cat_desc(code text) RETURNS text
AS $$
DECLARE
    desc text;
BEGIN
    SELECT c.description INTO STRICT desc
    FROM categories c
    WHERE c.code = get_cat_desc.code;

    RETURN desc;
EXCEPTION
    WHEN no_data_found THEN
        RETURN NULL;
END
$$ STABLE LANGUAGE plpgsql;

CREATE FUNCTION
```

Проверим, что функция работает правильно:

```
=> SELECT get_cat_desc('books');

 get_cat_desc
--------------
 Книги
(1 row)


=> SELECT get_cat_desc('movies');

 get_cat_desc
--------------

(1 row)
```

Можно ли проще?

Да, надо просто убрать STRICT или использовать подзапрос:

```
=> CREATE OR REPLACE FUNCTION get_cat_desc(code text) RETURNS text
AS $$
BEGIN
    RETURN (SELECT c.description
            FROM categories c
            WHERE c.code = get_cat_desc.code);
END
$$ STABLE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

В таком варианте хорошо видно, что PL/pgSQL тут вообще не нужен — достаточно SQL.

Проверим:

```
=> SELECT get_cat_desc('books');
```

```
 get_cat_desc
--------------
 Книги
(1 row)


=> SELECT get_cat_desc('movies');
```

```
 get_cat_desc
--------------

(1 row)
```

Задача: обновить строку таблицы с определенным идентификатором, а если такой строки нет — вставить ее.

Первый подход. Что здесь плохо?

```
=> CREATE OR REPLACE FUNCTION change(code text, description text)
RETURNS void
AS $$
DECLARE
    cnt integer;
BEGIN
    SELECT count(*) INTO cnt
    FROM categories c WHERE c.code = change.code;

    IF cnt = 0 THEN
        INSERT INTO categories VALUES (code, description);
    ELSE
        UPDATE categories c
        SET description = change.description
        WHERE c.code = change.code;
    END IF;
END
$$ VOLATILE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Плохо практически все, начиная с того, что такая функция будет работать некорректно на уровне изоляции Read Committed при наличии нескольких параллельно выполняющихся сеансов. Причина в том, что после выполнения SELECT и перед следующей операцией данные в базе могут измениться.

Это легко продемонстрировать, если добавить задержку между командами. Для разнообразия возьмем немного другой (но тоже неправильный) вариант:

```
=> CREATE OR REPLACE FUNCTION change(code text, description text)
RETURNS void
AS $$
BEGIN
    UPDATE categories c
    SET description = change.description
    WHERE c.code = change.code;

    IF NOT FOUND THEN
        PERFORM pg_sleep(1); -- тут может произойти все, что угодно
        INSERT INTO categories VALUES (code, description);
    END IF;
END
$$ VOLATILE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Теперь выполним функцию в двух сеансах почти одновременно:

```
=> SELECT change('games', 'Игры');
```

```
  => SELECT change('games', 'Игры');
```

```
  ERROR:  duplicate key value violates unique constraint "categories_code_key"
  DETAIL:  Key (code)=(games) already exists.
  CONTEXT:  SQL statement "INSERT INTO categories VALUES (code, description)"
  PL/pgSQL function change(text,text) line 9 at SQL statement
```

```
 change
--------

(1 row)
```

Правильное решение можно построить с помощью обработки ошибки:

```
=> CREATE OR REPLACE FUNCTION change(code text, description text)
RETURNS void
AS $$
BEGIN
    LOOP
        UPDATE categories c
        SET description = change.description
        WHERE c.code = change.code;

        EXIT WHEN FOUND;
        PERFORM pg_sleep(1); -- для демонстрации

        BEGIN
            INSERT INTO categories VALUES (code, description);
            EXIT;
        EXCEPTION
            WHEN unique_violation THEN NULL;
        END;
    END LOOP;
END
$$ VOLATILE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Проверим.

```
=> SELECT change('vynil', 'Грампластинки');
```

```
  => SELECT change('vynil', 'Грампластинки');
```

```
   change
  --------

  (1 row)
```

```
 change
--------

(1 row)
```

Да, теперь все правильно.

Но можно решить задачу проще с помощью варианта команды INSERT, который пробует выполнить вставку, а при возникновении конфликта — обновление. И снова достаточно простого SQL.

```
=> CREATE OR REPLACE FUNCTION change(code text, description text)
RETURNS void
VOLATILE LANGUAGE sql
BEGIN ATOMIC
    INSERT INTO categories VALUES (code, description)
    ON CONFLICT(code)
        DO UPDATE SET description = change.description;
END;
```

```
CREATE FUNCTION
```

Задача: гарантировать, что данные обрабатываются одновременно только одним процессом (на уровне изоляции Read Committed).

Используя ту же таблицу, представим, что периодически категория требует специальной однопоточной обработки. Можно написать функцию следующим образом:

```
=> CREATE OR REPLACE FUNCTION process_cat(code text) RETURNS text
AS $$
BEGIN
    PERFORM c.code FROM categories c WHERE c.code = process_cat.code
        FOR UPDATE NOWAIT;  -- пробуем блокировать строку без ожидания
    PERFORM pg_sleep(1); -- собственно обработка
    RETURN 'Категория обработана';
EXCEPTION
    WHEN lock_not_available THEN
        RETURN 'Другой процесс уже обрабатывает эту категорию';
END
$$ VOLATILE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Проверим, что все правильно:

```
=> SELECT process_cat('books');
```

```
  => SELECT process_cat('books');

                   process_cat
  ----------------------------------------------
   Другой процесс уже обрабатывает эту категорию
  (1 row)


     process_cat
----------------------
 Категория обработана
(1 row)
```

Но и эту задачу можно решить без обработки ошибок, используя рекомендательные блокировки:

```
=> CREATE OR REPLACE FUNCTION process_cat(code text) RETURNS text
AS $$
BEGIN
    IF pg_try_advisory_xact_lock(hashtext(code)) THEN
        PERFORM pg_sleep(1); -- собственно обработка
        RETURN 'Категория обработана';
    ELSE
        RETURN 'Другой процесс уже обрабатывает эту категорию';
    END IF;
END
$$ VOLATILE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Проверим:

```
=> SELECT process_cat('books');
```

```
  => SELECT process_cat('books');

                   process_cat
  ----------------------------------------------
   Другой процесс уже обрабатывает эту категорию
  (1 row)


     process_cat
----------------------
 Категория обработана
(1 row)
```

Приведем и пример, когда без обработки ошибок не обойтись.

Задача: организовать обработку пакета документов; ошибка при обработке одного документа не должна приводить к общему сбою.

```
=> CREATE TYPE doc_status AS ENUM -- тип перечисления
    ('READY', 'ERROR', 'PROCESSED');
```

```
CREATE TYPE
```

```
=> CREATE TABLE documents(
    id integer,
    version integer,
    status doc_status,
    message text
);
```

```
CREATE TABLE
```

```
=> INSERT INTO documents(id, version, status)
    SELECT id, 1, 'READY' FROM generate_series(1,100) id;
```

```
INSERT 0 100
```

Процедура, обрабатывающая один документ, иногда приводит к ошибке:

```
=> CREATE PROCEDURE process_one_doc(id integer)
AS $$
BEGIN
    UPDATE documents d
    SET version = version + 1
    WHERE d.id = process_one_doc.id;
    -- обработка может длиться долго
    IF random() < 0.05 THEN
        RAISE EXCEPTION 'Случилось страшное';
    END IF;
END
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

Теперь напишем процедуру, обрабатывающую все документы. Она вызывает в цикле обработку одного документа и при необходимости обрабатывает ошибку.

Обратите внимание, что фиксация транзакции выполняется вне блока с секцией EXCEPTION.

```
=> CREATE PROCEDURE process_docs()
AS $$
DECLARE
    doc record;
BEGIN
    FOR doc IN (SELECT id FROM documents WHERE status = 'READY')
    LOOP
        BEGIN
            CALL process_one_doc(doc.id);

            UPDATE documents d
            SET status = 'PROCESSED'
            WHERE d.id = doc.id;
        EXCEPTION
            WHEN others THEN
                UPDATE documents d
                SET status = 'ERROR', message = sqlerrm
                WHERE d.id = doc.id;
        END;
        COMMIT; -- каждый документ в своей транзакции
    END LOOP;
END
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

Такую же обработку можно организовать и при помощи функции, но тогда все документы будут обрабатываться в одной общей транзакции, что может приводить к проблемам, если обработка выполняется долго. Этот вопрос детально изучается в курсе DEV2.

Проверим результат:

```
=> CALL process_docs();
```

```
CALL
```

```
=> SELECT d.status, d.version, count(*)::integer
FROM documents d
GROUP BY d.status, d.version;
```

```
  status   | version | count
-----------+---------+-------
 PROCESSED |       2 |    96
 ERROR     |       1 |     4
(2 rows)
```

Как видим, часть документов не обработалась, но это не помешало обработке остальных.

Информация об ошибках удобно сохраняется в самой таблице:

```
=> SELECT * FROM documents d WHERE d.status = 'ERROR';
```

```
 id | version | status |      message
----+---------+--------+-------------------
 87 |       1 | ERROR  | Случилось страшное
 23 |       1 | ERROR  | Случилось страшное
 97 |       1 | ERROR  | Случилось страшное
 48 |       1 | ERROR  | Случилось страшное
(4 rows)
```

И еще раз обратим внимание, что при возникновении ошибки происходит откат к точке сохранения в начале блока: благодаря этому версии документов в статусе ERROR остались равными 1.

Итоги

Поиск обработчика ошибки происходит «изнутри наружу» в порядке вложенности блоков и вызова подпрограмм

В начале блока с EXCEPTION устанавливается неявная точка сохранения, при ошибке происходит откат к этой точке

Неперехваченная ошибка приводит к обрыву транзакции, сообщения отправляются клиенту и в журнал сервера

Обработка ошибок связана с накладными расходами

Практика

1. Если при добавлении новой книги указать одного и того же автора несколько раз, произойдет ошибка.
   Измените функцию add_book: перехватите ошибку нарушения уникальности и вместо нее вызовите ошибку с понятным текстовым сообщением.

   Проверьте изменения в приложении.

1. Чтобы определить название ошибки, которую необходимо перехватить, перехватите все ошибки (WHEN OTHERS) и выведите необходимую информацию (вызвав новую ошибку с соответствующим текстом).

После этого не забудьте заменить WHEN OTHERS на конкретную ошибку — пусть остальные типы ошибок обрабатываются на более высоком уровне, раз в этом месте кода нет возможности сделать что-то конструктивное.

(В реальной жизни не стоило бы обрабатывать и нарушение уникальности — лучше было бы изменить приложение так, чтобы оно не позволяло указывать двух одинаковых авторов.)

1. Обработка повторяющихся авторов при добавлении книги

```
=> CREATE OR REPLACE FUNCTION add_book(title text, authors integer[])
RETURNS integer
AS $$
DECLARE
    book_id integer;
    id integer;
    seq_num integer := 1;
BEGIN
    INSERT INTO books(title)
        VALUES(title)
        RETURNING books.book_id INTO book_id;
    FOREACH id IN ARRAY authors LOOP
        INSERT INTO authorship(book_id, author_id, seq_num)
            VALUES (book_id, id, seq_num);
        seq_num := seq_num + 1;
    END LOOP;
    RETURN book_id;
EXCEPTION
    WHEN unique_violation THEN
        RAISE EXCEPTION 'Один и тот же автор не может быть указан дважды';
END
$$ VOLATILE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Практика+

1. Ряд языков имеет конструкцию try … catch … finally …, в которой try соответствует BEGIN, catch — EXCEPTION, а операторы из блока finally срабатывают всегда, независимо от того, возникло ли исключение и было ли оно обработано блоком catch. Предложите способ добиться подобного эффекта в PL/pgSQL.
2. Сравните стеки вызовов, получаемые конструкциями GET STACKED DIAGNOSTICS с элементом pg_exception_context и GET [CURRENT] DIAGNOSTICS с элементом pg_context.
3. Напишите функцию getstack, возвращающую текущий стек вызовов в виде массива строк. Сама функция getstack не должна фигурировать в стеке.

1. Самый простой способ — просто повторить операторы finally в нескольких местах. Однако попробуйте построить такую конструкцию, чтобы эти операторы можно было написать ровно один раз.

2. Для начала обратитесь к документации:

https://postgrespro.ru/docs/postgresql/16/plpgsql-statements#PLPGSQL-STATEMENTS-DIAGNOSTICS

https://postgrespro.ru/docs/postgresql/16/plpgsql-control-structures#PLPGSQL-EXCEPTION-DIAGNOSTICS

1. Try-catch-finally

```
=> CREATE DATABASE plpgsql_exceptions;
```

```
CREATE DATABASE
```

```
=> \c plpgsql_exceptions
```

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

Сложность состоит в том, что операторы finally должны выполняться всегда, даже в случае возникновения ошибки в операторах catch (блок EXCEPTION).

Решение может использовать два вложенных блока и фиктивное исключение, которое вызывается при нормальном завершении внутреннего блока. Это дает возможность поместить операторы finally в одно место — обработчик ошибок внешнего блока.

```
=> DO $$
BEGIN
    BEGIN
        RAISE NOTICE 'Операторы try';
        --
        RAISE NOTICE '...нет исключения';
    EXCEPTION
        WHEN no_data_found THEN
            RAISE NOTICE 'Операторы catch';
    END;
    RAISE SQLSTATE 'ALLOK';
EXCEPTION
    WHEN others THEN
        RAISE NOTICE 'Операторы finally';
        IF SQLSTATE != 'ALLOK' THEN
            RAISE;
        END IF;
END
$$;
```

```
NOTICE:  Операторы try
NOTICE:  ...нет исключения
NOTICE:  Операторы finally
DO
```

```
=> DO $$
BEGIN
    BEGIN
        RAISE NOTICE 'Операторы try';
        --
        RAISE NOTICE '...исключение, которое обрабатывается';
        RAISE no_data_found;
    EXCEPTION
        WHEN no_data_found THEN
            RAISE NOTICE 'Операторы catch';
    END;
    RAISE SQLSTATE 'ALLOK';
EXCEPTION
    WHEN others THEN
        RAISE NOTICE 'Операторы finally';
        IF SQLSTATE != 'ALLOK' THEN
            RAISE;
        END IF;
END
$$;
```

```
NOTICE:  Операторы try
NOTICE:  ...исключение, которое обрабатывается
NOTICE:  Операторы catch
NOTICE:  Операторы finally
DO
```

```
=> DO $$
BEGIN
    BEGIN
        RAISE NOTICE 'Операторы try';
        --
        RAISE NOTICE '...исключение, которое не обрабатывается';
        RAISE division_by_zero;
    EXCEPTION
        WHEN no_data_found THEN
            RAISE NOTICE 'Операторы catch';
    END;
    RAISE SQLSTATE 'ALLOK';
EXCEPTION
    WHEN others THEN
        RAISE NOTICE 'Операторы finally';
        IF SQLSTATE != 'ALLOK' THEN
            RAISE;
        END IF;
END
$$;
```

```
NOTICE:  Операторы try
NOTICE:  ...исключение, которое не обрабатывается
NOTICE:  Операторы finally
ERROR:  division_by_zero
CONTEXT:  PL/pgSQL function inline_code_block line 7 at RAISE
```

Но в предложенном решении всегда происходит откат всех изменений, выполненных в блоке, поэтому оно не годится для команд, изменяющих состояние базы данных. Также не стоит забывать о накладных расходах на обработку исключений: это задание — не более, чем просто упражнение.

2. GET DIAGNOSTICS

```
=> DO $$
DECLARE
    ctx text;
BEGIN
    RAISE division_by_zero;                      -- line 5
EXCEPTION
    WHEN others THEN
        GET STACKED DIAGNOSTICS ctx := PG_EXCEPTION_CONTEXT;
        RAISE NOTICE E'stacked =\n%', ctx;
        GET CURRENT DIAGNOSTICS ctx := PG_CONTEXT; -- line 10
        RAISE NOTICE E'current =\n%', ctx;
END
$$;
```

```
NOTICE:  stacked =
PL/pgSQL function inline_code_block line 5 at RAISE
NOTICE:  current =
PL/pgSQL function inline_code_block line 10 at GET DIAGNOSTICS
DO
```

GET STACKED DIAGNOSTICS дает стек вызовов, приведший к ошибке.

GET [CURRENT] DIAGNOSTICS дает текущий стек вызовов.

3. Стек вызовов как массив

Собственно функция:

```
=> CREATE FUNCTION getstack() RETURNS text[]
AS $$
DECLARE
    ctx text;
BEGIN
    GET DIAGNOSTICS ctx := PG_CONTEXT;
    RETURN (regexp_split_to_array(ctx, E'\n'))[2:];
END
$$ VOLATILE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Чтобы проверить ее работу, создадим несколько функций, которые вызывают друг друга:

```
=> CREATE FUNCTION foo() RETURNS integer
AS $$
BEGIN
    RETURN bar();
END

$$ VOLATILE LANGUAGE plpgsql;

CREATE FUNCTION

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

CREATE FUNCTION

=> CREATE FUNCTION baz() RETURNS integer
AS $$
BEGIN
    RAISE NOTICE '%', getstack();
    RETURN 0;
END
$$ VOLATILE LANGUAGE plpgsql;

CREATE FUNCTION

=> SELECT foo();

NOTICE:  {"PL/pgSQL function baz() line 3 at RAISE","PL/pgSQL function bar() line 3 at
RETURN","PL/pgSQL function foo() line 3 at RETURN"}
 foo
-----
   0
(1 row)


=> \c postgres

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

=> DROP DATABASE plpgsql_exceptions;

DROP DATABASE
```