День 4

Обработка ошибок

По умолчанию любая возникающая ошибка прерывает выполнение функции на PL/pgSQL и транзакцию, в которая она выполняется.

Использование в блоке секции EXCEPTION позволяет перехватывать и обрабатывать ошибки. Синтаксис секции EXCEPTION расширяет синтаксис обычного блока:

```
[ <<метка>> ]
[ DECLARE
   объявления ]
BEGIN
   операторы
EXCEPTION
   WHEN условие [ OR условие ... ] THEN
      операторы_обработчика
   [ WHEN условие [ OR условие ... ] THEN
        операторы_обработчика
     ... ]
END;
```

Обработка ошибок

Если ошибок не было, то выполняются все операторы блока и управление переходит к следующему оператору после END.

Но если при выполнении оператора происходит ошибка, то дальнейшая обработка прекращается и управление переходит к списку исключений в секции EXCEPTION.

В этом списке ищется первое исключение, условие которого соответствует ошибке. Если исключение найдено, то выполняются соответствующие операторы_обработчика и управление переходит к следующему оператору после END.

Если исключение не найдено, то ошибка передаётся наружу, как будто секции EXCEPTION не было. При этом ошибку можно перехватить в секции EXCEPTION внешнего блока. Если ошибка так и не была перехвачена, то обработка функции прекращается.

В качестве условия может задаваться одно из имён, перечисленных в Приложении A документации.

Если задаётся имя категории, ему соответствуют все ошибки в данной категории.

Специальному имени условия OTHERS (другие) соответствуют все типы ошибок, кроме QUERY_CANCELED и ASSERT_FAILURE.

Имена условий воспринимаются без учёта регистра. Условие ошибки также можно задать кодом SQLSTATE; например, эти два варианта равнозначны:

```
WHEN division_by_zero THEN ...
WHEN SQLSTATE '22012' THEN ...
```

Обработка ошибок

Если при выполнении операторов_обработчика возникнет новая ошибка, то она не может быть перехвачена в этой секции EXCEPTION.

Ошибка передаётся наружу и её можно перехватить в секции EXCEPTION внешнего блока.

При выполнении команд в секции EXCEPTION локальные переменные функции на PL/pgSQL сохраняют те значения, которые были на момент возникновения ошибки.

Однако все изменения в базе данных, выполненные в блоке, будут отменены.

EXCEPTION

- Соответствует блоку try catch в других языках
- Неявно создает точку сохранения в начале каждого блока
- Перед обработкой ошибки происходит откат к этой точке сохранения

Имена и коды ошибок

Каждой ошибке соответствует имя и код

Код - строка из пяти символов

В условии WHEN можно указывать имя или код.

Ошибки организованы в двухуровневую иерархию.

Согласно стандарту, первые два символа кода ошибки обозначают класс ошибок, а последние три символа обозначают определённое условие в этом классе. Таким образом, приложение, не знающее значение определённого кода ошибки, всё же может понять, что делать, по классу ошибки.

https://postgrespro.ru/docs/postgresql/15/errcodes-appendix

Имена и коды ошибок - пример

Класс 23 — Нарушение ограничения целостности	
23000	integrity_constraint_violation
23001	restrict_violation
23502	not_null_violation

Влияние на производительность

- Наличие секции EXCEPTION значительно увеличивает накладные расходы на вход/выход из блока, поэтому не используйте EXCEPTION без надобности
  - Установка точки сохранения
  - Откат к точке сохранения

Получение информации об ошибках

- Специальные переменные
  - Внутри секции EXCEPTION специальная переменная SQLSTATE содержит код ошибки, для которой было вызвано исключение
  - Специальная переменная SQLERRM содержит сообщение об ошибке, связанное с исключением. Эти переменные являются неопределёнными вне секции EXCEPTION.
- GET STACKED DIAGNOSTICS

GET STACKED DIAGNOSTICS

Имя	Тип	Описание
RETURNED_SQLSTATE	text	код исключения, возвращаемый SQLSTATE
COLUMN_NAME	text	имя столбца, относящегося к исключению
CONSTRAINT_NAME	text	имя ограничения целостности, относящегося к исключению
PG_DATATYPE_NAME	text	имя типа данных, относящегося к исключению
MESSAGE_TEXT	text	текст основного сообщения исключения
TABLE_NAME	text	имя таблицы, относящейся к исключению
SCHEMA_NAME	text	имя схемы, относящейся к исключению
PG_EXCEPTION_DETAIL	text	текст детального сообщения исключения (если есть)
PG_EXCEPTION_HINT	text	текст подсказки к исключению (если есть)
PG_EXCEPTION_CONTEXT	text	строки текста, описывающие стек

GET STACKED DIAGNOSTICS

```
DECLARE
  text_var1 text;
  text_var2 text;
  text_var3 text;
BEGIN
  -- здесь происходит обработка, которая может вызвать исключение
  ...
EXCEPTION WHEN OTHERS THEN
  GET STACKED DIAGNOSTICS text_var1 = MESSAGE_TEXT,
                          text_var2 = PG_EXCEPTION_DETAIL,
                          text_var3 = PG_EXCEPTION_HINT;
END;
```

GET DIAGNOSTICS

Команда GET DIAGNOSTICSполучает информацию о текущем состоянии выполнения кода

Её элемент состояния PG_CONTEXT позволяет определить текущее место выполнения кода.

PG_CONTEXT возвращает текст с несколькими строками, описывающий стек вызова.

В первой строке отмечается текущая функция и выполняемая в данный момент команда GET DIAGNOSTICS, а во второй и последующих строках отмечаются функции выше по стеку вызовов.

GET STACKED DIAGNOSTICS ... PG_EXCEPTION_CONTEXT возвращает похожий стек вызовов, но описывает не текущее место, а место, в котором произошла ошибка.

GET DIAGNOSTICS

```
BEGIN
  GET DIAGNOSTICS stack = PG_CONTEXT;
  RAISE NOTICE E'--- Стек вызова ---\n%', stack;
  RETURN 1;
END;
```

```
NOTICE:  --- Стек вызова ---
PL/pgSQL function inner_func() line 5 at GET DIAGNOSTICS
PL/pgSQL function outer_func() line 3 at RETURN
CONTEXT:  PL/pgSQL function outer_func() line 3 at RETURN
```

Генерация ошибок и вывод сообщений

- RAISE
- ASSERT

RAISE

- Команда RAISE предназначена для вывода сообщений и вызова ошибок
- Варианты вызова

RAISE [уровень] 'формат' [, выражение [, ...]] [USING параметр = значение [, ...]];

RAISE [уровень] имя_условия [USING параметр = выражение [, ...]];

RAISE [уровень] SQLSTATE 'sqlstate' [USING параметр = выражение [, ...]];

RAISE [уровень] USING параметр = выражение [, ...];

RAISE ;

Уровень

уровень задаёт уровень важности ошибки.

Возможные значения: DEBUG, LOG, INFO, NOTICE, WARNING и EXCEPTION.

По умолчанию используется EXCEPTION.

EXCEPTION вызывает ошибку (что обычно прерывает текущую транзакцию), остальные значения уровня только генерируют сообщения с различными уровнями приоритета. Будут ли сообщения конкретного приоритета переданы клиенту или записаны в журнал сервера, или и то, и другое, зависит от конфигурационных переменных log_min_messages и client_min_messages. Это настраивает администратор в конфигурации системы.

Формат

После указания уровня, если оно есть, можно задать строку формата (это должна быть простая строковая константа, не выражение).

Строка формата определяет вид текста об ошибке, который будет выдан.

За строкой формата могут следовать необязательные выражения аргументов, которые будут вставлены в сообщение.

Внутри строки формата знак % заменяется строковым представлением значения очередного аргумента. Чтобы выдать символ % буквально, продублируйте его (как %%). Число аргументов должно совпадать с числом местозаполнителей % в строке формата, иначе при компиляции функции возникнет ошибка.

```
RAISE NOTICE 'Вызов функции cs_create_job(%)', v_job_id;
```

USING

При помощи USING и последующих элементов параметр = выражение можно добавить дополнительную информацию к отчёту об ошибке. Все выражения представляют собой строковые выражения.

```
RAISE EXCEPTION 'Несуществующий ID --> %', user_id
    USING HINT = 'Проверьте ваш пользовательский ID’;
```

```
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = 'unique_violation';
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';
```

SQLSTATE

RAISE division_by_zero;

RAISE SQLSTATE '22012’;

Предложение USING в этом синтаксисе можно использовать для того, чтобы переопределить стандартное сообщение об ошибке, детальное сообщение, подсказку

RAISE unique_violation USING MESSAGE = 'ID пользователя уже существует: ' || user_id;

Воспроизведение ошибки

И заключительный вариант, в котором RAISE не имеет параметров вообще.

Эта форма может использоваться только в секции EXCEPTION блока и предназначена для того, чтобы повторно вызвать ошибку, которая сейчас перехвачена и обрабатывается.

ASSERT

ASSERT условие [, сообщение];

Оператор ASSERT представляет удобное средство вставлять отладочные проверки в функции PL/pgSQL.

ASSERT

условие — это логическое выражение, которое, как ожидается, должно быть всегда истинным; если это так, оператор ASSERT больше ничего не делает.

Если же оно возвращает ложь или NULL, этот оператор выдаёт исключение ASSERT_FAILURE. (Если ошибка происходит при вычислении условия, она выдаётся как обычная ошибка.)

Если в нём задаётся необязательное сообщение, результат этого выражения (если он не NULL) заменяет сообщение об ошибке по умолчанию «assertion failed» (нарушение истинности), в случае, если условие не выполняется. В обычном случае, когда условие утверждения выполняется, выражение сообщения не вычисляется.

Проверку утверждений можно включить или отключить с помощью конфигурационного параметра plpgsql.check_asserts, принимающего логическое значение; по умолчанию она включена (on). Если этот параметр отключён (off), операторы ASSERT ничего не делают.

Учтите, что оператор ASSERT предназначен для выявления программных дефектов, а не для вывода обычных ошибок (для этого используется оператор RAISE, описанный выше).

Функции в PL/pgSQL

- Функции могут писаться на разных языках, в том числе на PL/pgSQL
- Основные принципы написания схожи с функциями на языке SQL
- В теле функции используется блок PL/pgSQL
- Язык задается language plpgsql

Create Function

```
create [or replace] function function_name(param_list)
returns return_type
language plpgsql
As
$$
declare -- variable declaration
begin – logic
end;
$$
```

Параметры функций

- IN -- входной, по умолчанию
- OUT -- выходной
- INOUT – принимает входное и отдает обновленное значение

Перегрузка функций

- Аналогично SQL функциям
- Уникальная сигнатура – одинаковое имя и набор разных входных параметров
- Не стоит задавать дефолтовые значения параметрам – может вызвать ошибку из-за неопределенности сигнатуры

Drop function

drop function [if exists] function_name(argument_list)

[cascade | restrict]

При удалении перегруженной функции обязательно указать нужную комбинацию параметров

Триггеры

- DML и DDL триггеры
- DML триггер
- функция, срабатывающая автоматически при заданном событии (INSERT, UPDATE, DELETE или TRUNCATE)
- Состоит из определения объекта триггер и привязанной к нему пользовательской функции, заранее для этого созданной
- Row и statement level
- Before и After

Стандарт и особенности PostgreSQL

- Поддержка триггеров для TRUNCATE
- Statement-level триггер на представления
- Обязательно использовать UDF

Триггер и триггерная функция

Триггер

- объект базы данных — задает список обрабатываемых событий
- По событию вызывается триггерная функция, ей передается контекст

Триггерная функция

- объект базы данных — обработчик события
- Любой язык, кроме чистого SQL
- выполняется в той же транзакции, что и основная операция
- функция не принимает параметры,
- возвращает псевдотип trigger (это по сути record, повторяющий структуру таблицы)
- может использоваться в нескольких триггерах

Create trigger function

```
CREATE FUNCTION trigger_function()
RETURNS TRIGGER
LANGUAGE PLPGSQL
AS
$$BEGIN
-- trigger logic
END;
$$
```

Особенности

- Не принимает аргументы
- Возращает псевдотип trigger
- Структура TriggerData
- OLD – состояние записи в таблице до операции
- NEW – после операции
- TG_ … - локальные переменные

CREATE TRIGGER statement

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} { event }
 ON table_name
[FOR [EACH] { ROW | STATEMENT }]
EXECUTE PROCEDURE trigger_function
```

event может быть - INSERT , DELETE, UPDATE или TRUNCATE

DROP TRIGGER

DROP TRIGGER [IF EXISTS] trigger_name

ON table_name [CASCADE | RESTRICT];

RESTRICT – по умолчанию, не дает удалить триггер при наличии зависимых объектов

ALTER TRIGGER

```
ALTER TRIGGER trigger_name
ON table_name
RENAME TO new_trigger_name;
```

Enable и disable триггеров

```
ALTER TABLE table_name
DISABLE TRIGGER trigger_name | ALL
```

```
ALTER TABLE table_name
ENABLE TRIGGER trigger_name |  ALL;
```

Порядок срабатывания триггеров

1. BEFORE STATEMENT
2. BEFORE ROW
3. INSTEAD OF ROW
4. AFTER ROW
5. AFTER STATEMENT

Before statement

- Срабатывает до операции
- Возвращаемое значение игнорируется
- Контекст TG-переменные
  - TG_WHEN = «BEFORE»,
  - TG_LEVEL = «STATEMENT»,
  - TG_OP = «INSERT»/«UPDATE»/«DELETE»/TRUNCATE
- Возвращаемое значение триггерной функции игнорируется, можно вернуть NULL. Если в триггере происходит ошибка, то операция отменяется

Before row

- Срабатывает перед действием со строкой в процессе выполнения операции
- Возвращаемое значение
  - строка (возможно, измененная)
- null отменяет действие
- Контекст
  - OLD update, delete
  - NEW insert, update
  - TG-переменные
    - TG_WHEN = «BEFORE»,
    - TG_LEVEL = «ROW»,
    - TG_OP = «INSERT»/«UPDATE»/«DELETE»

Instead of row

- Срабатывает вместо действия со строкой только для представлений
- Возвращаемое значение
  - строка (возможно, измененная) — видна в RETURNING
  - null отменяет действие
- Контекст
  - OLD update, delete
  - NEW insert, update
  - TG-переменные

After row

- Срабатывает после выполнения операции
- Обработка в очереди из прошедших условие WHEN
- Возвращаемое значение
  - игнорируется
- Контекст
  - OLD, OLD TABLE update, delete
  - NEW, NEW TABLE insert, update
  - TG-переменные
    - TG_WHEN = «AFTER»,
    - TG_LEVEL = «ROW»,
    - TG_OP = «INSERT»/«UPDATE»/«DELETE».

After statement

- Срабатывает после операции (даже если не затронута ни одна строка)
- Возвращаемое значение
  - игнорируется
- Контекст
  - OLD TABLE update, delete
  - NEW TABLE insert, update
  - TG-переменные
    - TG_WHEN = «AFTER»,
    - TG_LEVEL = «STATEMENT»,
    - TG_OP = «INSERT»/«UPDATE»/«DELETE»/«TRUNCATE»

DDL триггеры

DDL триггер похож на обычный, но это другой объект, срабатывающий на DDL-операции (CREATE, ALTER, DROP, COMMENT, GRANT, REVOKE)

- Триггерная функция
  - не принимает параметры,
  - возвращает значение псевдотипа event_trigger
- Контекст
  - Применяются специальные функции
- События
  - DDL_COMMAND_START перед выполнением команды
  - DDL_COMMAND_END после выполнения команды
  - TABLE_REWRITE перед перезаписью таблицы
  - SQL_DROP после удаления объектов

Пользовательские агрегаты

- Построчная обработка агрегируемой выборки
  - состояние
  - функции перехода и финализации

Принцип работы

- Имеется некоторое начальное состояние, представленное значением типа данных, которое инициализируется стартовым значением (например, тип numeric и стартовое значение 0).
- Для каждой строки агрегируемой выборки вызывается функция перехода, которой передается значение из текущей строки, при этом функция должна обновить состояние (например, операция сложения).
- В конце вызывается функция финализации, которая преобразует полученное состояние в результат (например, достаточно просто вернуть число — в итоге получается аналог функции sum)

CREATE AGGREGATE

```
CREATE AGGREGATE avg (float8)
(
    sfunc = float8_accum,
    stype = float8[],
    finalfunc = float8_avg,
    initcond = '{0,0,0}'
);
```

Пользовательские операторы

- Любой оператор представляет собой «синтаксический сахар» для вызова нижележащей функции, выполняющей реальную работу; поэтому прежде чем вы сможете создать оператор, необходимо создать нижележащую функцию.
- Однако оператор — *не исключительно* синтаксический сахар, так как он несёт и дополнительную информацию, помогающую планировщику запросов оптимизировать запросы с этим оператором.
- PostgreSQL поддерживает префиксные и инфиксные операторы.
- Операторы могут быть перегружены; то есть одно имя оператора могут иметь различные операторы с разным количеством и типами операндов.
- Когда выполняется запрос, система определяет, какой оператор вызвать, по количеству и типам предоставленных операндов.

CREATE OPERATOR

CREATE OPERATOR имя (

{FUNCTION|PROCEDURE} = имя_функции

[, LEFTARG = тип_слева] [, RIGHTARG = тип_справа]

[, COMMUTATOR = коммут_оператор] [, NEGATOR = обратный_оператор]

[, RESTRICT = процедура_ограничения] [, JOIN = процедура_соединения]

[, HASHES] [, MERGES]

)

Пользовательский оператор

В следующем примере создаётся оператор сложения двух комплексных чисел.

Сначала нам нужна функция, собственно выполняющая операцию, а затем мы сможем определить оператор:

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS 'имя_файла', 'complex_add'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    function = complex_add,
    commutator = +
);
-- применяем
SELECT (a + b) AS c FROM test_complex;
```

Преобразователь

Трансформация определяет, как преобразовать тип данных для процедурного языка. Например, если написать на языке PL/Python функцию, использующую тип hstore, PL/Python заведомо не знает, как должны представляться значения hstore в среде Python. Обычно реализации языка нисходят к текстовому представлению, но это может быть неудобно, когда более уместен был бы, например, ассоциативный массив или список.

Трансформация определяет две функции:

- Функция «из SQL» преобразует тип из среды SQL в среду языка. Эта функция будет вызываться для аргументов функции, написанной на этом языке.
- Функция «в SQL» преобразует тип из среды языка в среду SQL. Эта функция будет вызываться для значения, возвращаемого из функции на этом языке.

Предоставлять обе эти функции не требуется, можно ограничиться одной. Если одна из них не указана, при необходимости выбирается поведение, принятое для языка по умолчанию.

Шаги создания трансформации

```
CREATE TYPE hstore ...; -- создаем тип
CREATE EXTENSION plpython3u; --создаем язык
-- создаем функции
CREATE FUNCTION hstore_to_plpython(val internal) RETURNS internal
LANGUAGE C STRICT IMMUTABLE
AS ...;

CREATE FUNCTION plpython_to_hstore(val internal) RETURNS hstore
LANGUAGE C STRICT IMMUTABLE
AS ...;

-- создаем трансформацию
CREATE TRANSFORM FOR hstore LANGUAGE plpython3u (
   FROM SQL WITH FUNCTION hstore_to_plpython(internal),
   TO SQL WITH FUNCTION plpython_to_hstore(internal)
);
```

Домены

Домен основывается на определённом нижележащем типе и во многих аспектах взаимозаменяем с ним.

Однако домен может иметь ограничения, уменьшающие множество допустимых для него значений относительно нижележащего типа.

Домены создаются SQL-командой CREATE DOMAIN

CREATE DOMAIN

CREATE DOMAIN имя [AS] тип_данных
    [COLLATE правило_сортировки]
    [DEFAULT выражение]
    [ограничение [...]]

Здесь ограничение:

[CONSTRAINT имя_ограничения]
{ NOT NULL | NULL | CHECK (выражение) }

CREATE DOMAIN

```
CREATE DOMAIN us_postal_code AS TEXT
CHECK(
   VALUE ~ '^\d{5}$'
OR VALUE ~ '^\d{5}-\d{4}$'
);

CREATE TABLE us_snail_addy (
  address_id SERIAL PRIMARY KEY,
  street1 TEXT NOT NULL,
  street2 TEXT,
  street3 TEXT,
  city TEXT NOT NULL,
  postal us_postal_code NOT NULL
);
```

Rule - правила

Система правил PostgreSQL позволяет определить альтернативное действие, заменяющее операции добавления, изменения или удаления данных в таблицах базы данных.

Грубо говоря, правило описывает дополнительные команды, которые будут выполняться при вызове определённой команды для определённой таблицы.

Кроме того, правило INSTEAD может заменить заданную команду другой, либо сделать, чтобы она не выполнялась вовсе.

Правила также применяются для реализации SQL-запросов. Важно понимать, что правило это фактически механизм преобразования команд (макрос).

Заданное преобразование имеет место до начала выполнения команды.

Когда требуется выполнить некоторую операцию независимо для каждой физической строки, скорее всего, для этого нужно применять триггер, а не правило.

CREATE RULE

CREATE [OR REPLACE] RULE имя AS ON событие

TO имя_таблицы [WHERE условие]

DO [ALSO | INSTEAD] { NOTHING | команда | (команда ; команда ...) }

Здесь допускается событие:

SELECT | INSERT | UPDATE | DELETE

Правила

INSTEAD

INSTEAD указывает, что заданные команды должны выполняться вместо исходной команды.

ALSO

ALSO указывает, что заданные команды должны выполняться в дополнение к исходной команде.

Если ни ALSO, ни INSTEAD не указано, по умолчанию подразумевается ALSO.

безусловное правило DO INSTEAD NOTHING применяется, чтобы система понимала, что ей никогда не придётся изменять нижележащую таблицу

Правила

```
CREATE RULE "_RETURN" AS
   ON SELECT TO t1
   DO INSTEAD
      SELECT * FROM t2;

CREATE RULE "_RETURN" AS
   ON SELECT TO t2
   DO INSTEAD
      SELECT * FROM t1;

SELECT * FROM t1;
```

Хранимые процедуры

- Введены в версии 11
- Разрешают управление транзакциями в теле процедуры

Процедуры

Процедура — объект базы данных, подобный функции, но имеющий следующие отличия:

- Процедуры определяются командой CREATE PROCEDURE, а не CREATE FUNCTION.
- Процедуры, в отличие от функций, не возвращают значение; поэтому в CREATE PROCEDURE отсутствует предложение RETURNS. Однако процедуры могут выдавать данные в вызывающий код через выходные параметры.
- Функции вызываются как часть запроса или команды DML, а процедуры вызываются отдельно командой CALL.
- Процедура, в отличие от функции, может фиксировать или откатывать транзакции во время её выполнения (а затем автоматически начинать новую транзакцию), если вызывающая команда CALL находится не в явном блоке транзакции.

Создание и вызов процедуры

```
CREATE PROCEDURE procedure1(INOUT p1 TEXT)
AS $$
BEGIN
   RAISE NOTICE 'Procedure Parameter: %', p1 ;
END ;
$$
LANGUAGE plpgsql ;

CALL procedure1 (‘ Test procedure’);
CALL procedure1 (p1=>’Test stored proc');
```

Управление транзакцией в процедуре

```
CREATE OR REPLACE PROCEDURE transaction_test()
LANGUAGE plpgsql
AS $$
DECLARE
BEGIN
  CREATE TABLE committed_table (id int);
  INSERT INTO committed_table VALUES (1);
  COMMIT;
  CREATE TABLE rollback_table (id int);
  INSERT INTO rollback_table VALUES (1);
  ROLLBACK;
END $$;
```

Temporary таблицы

- Только на время сессии или транзакции (как задано)

CREATE [GLOBAL | LOCAL] TEMPORARY | TEMP TABLE [IF NOT EXISTS] name

(column definitions and constraints)

[ON COMMIT PRESERVE ROWS | DELETE ROWS | DROP]

Расширения - extension

- Похожи на package в Oracle
- Механизм добавления новой функциональности в базу данных
- Сервер поставляется с набором готовых к использованию расширений
- Их надо подключить к базе данных через create extension
- Можно создавать свои расширения, устанавливать их на сервер и затем также подключать к базе данных

Управление расширениями

- CREATE extension hstore;
- \dx
- CREATE extension bloom WITH schema public;
- DROP extension hstore;

- требуются права суперюзера!

Разработка расширения

- control file формата extension_name.control
  - Описывает метаданные расширения
- SQL script file формата extension--version.sql
  - Содержит объекты расширения
- Эти два файла можно скопировать в SHAREDIR/extension, после чего можно загружать расширение в базу через create extension
- Билд PGXS
  - Makefile
  - Make install

Identity

- Новый вариант счетчика на замену serial (с версии 10)
- Управляется через alter table
- Возможно несколько счетчиков identity в таблице
- Как и serial использует в реализации sequence