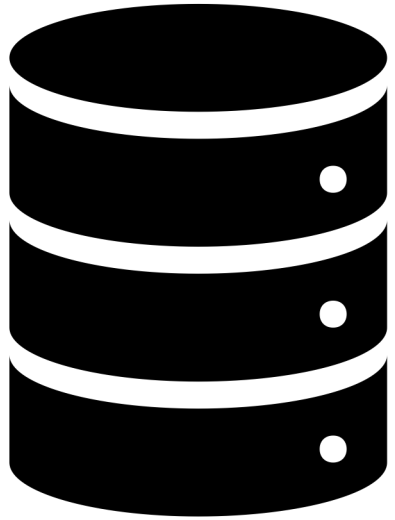
Базы данных



Лекция 8. Хранимые процедуры и функции.

Меркурьева Надежда
✉ merkurievanad@gmail.com
@merkurievanad

ФПМИ МФТИ, 2021

ХРАНИМЫЕ ПРОЦЕДУРЫ И ФУНКЦИИ

– объект базы данных, представляющий собой набор SQL-инструкций, который компилируется один раз и хранится на сервере

- Похожи на обыкновенные процедуры языков высокого уровня:
  - входные параметры
  - выходные параметры
  - локальные переменные
  - числовые вычисления и операции над символьными данными
- Могут выполняться стандартные операции с базами данных (как DDL, так и DML)
- Возможны циклы и ветвления

ХРАНИМЫЕ ФУНКЦИИ

- позволяют повысить производительность
- расширяют возможности программирования
- поддерживают функции безопасности данных

- Вместо хранения часто запроса, достаточно ссылаться на соответствующую хранимую процедуру

- Рассматриваем на примере PostgreSQL

ХРАНИМЫЕ ФУНКЦИИ

PostgreSQL

```
CREATE [ OR REPLACE ] FUNCTION
name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
[ RETURNS rettype |
  RETURNS TABLE ( column_name column_type [, ...] ) ]
{ LANGUAGE lang_name
  | TRANSFORM { FOR TYPE type_name } [, ... ]
  | WINDOW
  | IMMUTABLE
  | STABLE
  | VOLATILE
  | [ NOT ] LEAKPROOF
  | CALLED ON NULL INPUT
  | RETURNS NULL ON NULL INPUT
  | STRICT
  | [ EXTERNAL ] SECURITY INVOKER
  | [ EXTERNAL ] SECURITY DEFINER
  | COST execution_cost
  | ROWS result_rows
  | SET configuration_parameter { TO value | = value | FROM CURRENT }
  | AS 'definition'
 | AS 'obj_file', 'link_symbol'
} ...
[ WITH ( attribute [, ...] ) ]
```

ХРАНИМЫЕ ФУНКЦИИ

```
CREATE FUNCTION one()RETURNS integer AS $$
  SELECT 1 AS result;
$$ LANGUAGE SQL;

one
-----
1
```

ХРАНИМЫЕ ФУНКЦИИ

```
CREATE FUNCTION add_int(x integer, y integer)
RETURNS integer AS $$
  SELECT x + y;
$$ LANGUAGE SQL;


SELECT add_int(1, 2) AS answer;


answer
--------
3
```

ХРАНИМЫЕ ФУНКЦИИ

```
CREATE FUNCTION add_int(integer, integer)
RETURNS integer AS $$
  SELECT $1 + $2;
$$ LANGUAGE SQL;


SELECT add_int(1, 2) AS answer;


answer
--------
3
```

ХРАНИМЫЕ ФУНКЦИИ

```
CREATE FUNCTION add_int(integer, integer)
RETURNS integer
    AS 'select $1 + $2;’
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;

SELECT add_int(20, 22) AS answer;

answer
------
42
```

ХРАНИМЫЕ ФУНКЦИИ

```
CREATE FUNCTION dup(in int, out f1 int, out f2 text) AS $$
  SELECT $1, CAST($1 AS text) || ' is text'
$$ LANGUAGE SQL;

SELECT *
  FROM dup(42);

f1 | f2
---+-----------
42 |42 is text
```

ХРАНИМЫЕ ФУНКЦИИ

```
CREATE TYPE dup_result AS (f1 int, f2 text);

CREATE FUNCTION dup(int)
RETURNS dup_result AS $$
  SELECT $1, CAST($1 AS text) || ' is text'
$$ LANGUAGE SQL;

SELECT *
  FROM dup(42);

f1 | f2
---+-----------
42 |42 is text
```

ХРАНИМЫЕ ФУНКЦИИ

```
CREATE FUNCTION dup(int)
RETURNS TABLE(f1 int, f2 text) AS $$
  SELECT $1, CAST($1 AS text) || ' is text'
$$ LANGUAGE SQL;

SELECT *
  FROM dup(42);

f1 | f2
---+-----------
42 |42 is text
```

ХРАНИМЫЕ ФУНКЦИИ

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)
RETURNS int
LANGUAGE SQL AS $$
  SELECT $1 + $2 + $3;
$$;

SELECT foo(10, 20, 30);

foo
-----
60
```

ХРАНИМЫЕ ФУНКЦИИ

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)
RETURNS int
LANGUAGE SQL AS $$
  SELECT $1 + $2 + $3;
$$;

SELECT foo(10, 20);

foo
-----
33
```

ХРАНИМЫЕ ФУНКЦИИ

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)
RETURNS int
LANGUAGE SQL AS $$
  SELECT $1 + $2 + $3;
$$;

SELECT foo(10);

foo
-----
15
```

ХРАНИМЫЕ ФУНКЦИИ

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)
RETURNS int
LANGUAGE SQL AS $$
  SELECT $1 + $2 + $3;
$$;

SELECT foo();
```

Получаем ошибку, т.к. дефолтное значение для первого атрибута не задано

ХРАНИМЫЕ ФУНКЦИИ

```
CREATE FUNCTION tf1 (account_no integer, debit numeric)
RETURNS numeric AS $$
  UPDATE bank
     SET balance = balance - debit
   WHERE account_no = tf1.account_no;

  SELECT balance
    FROM bank
   WHERE account_no = tf1.account_no;
$$ LANGUAGE SQL;
```

PL/pgSQL

is easy to use. © postgresql.org/

ХРАНИМЫЕ ФУНКЦИИ

```
CREATE FUNCTION somefunc(integer, text)
RETURNS integer AS
  'function body text'
LANGUAGE plpgsql;
```

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
  statements
END
  [ label ];
```

ХРАНИМЫЕ ФУНКЦИИ

```
CREATE OR REPLACE FUNCTION increment(i integer)
RETURNS integer AS $$
  BEGIN
    RETURN i + 1;
  END;
$$ LANGUAGE plpgsql;

SELECT increment(41) AS answer;

answer
------
42
```

ХРАНИМЫЕ ФУНКЦИИ

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
  quantity integer := 30;
BEGIN
  RAISE NOTICE 'Quantity here is %', quantity; -- Prints 30
  quantity := 50;
  -- Create a subblock
  DECLARE
    quantity integer := 80;
  BEGIN
    RAISE NOTICE 'Quantity here is %', quantity; -- Prints 80
    RAISE NOTICE 'Outer quantity here is %', outerblock.quantity; -- Prints 50
  END;
  RAISE NOTICE 'Quantity here is %', quantity; -- Prints 50
  RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

ОСНОВЫ PL/pgSQL

1. Оператор присваивания:

```
var := 10
```

2. В блоке DECLARE можно переименовывать переменные:

```
new_name ALIAS FOR $1;
```

3. Динамическое выполнение запросов:

```
EXECUTE sql_query [INTO target] [USING expression]
```

Пример:

```
EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = $1 AND inserted <= $2'
    INTO c
   USING checked_user, checked_date;
```

- Параметры $1 и $2 в этом случае – это не входные данные функции, а параметры, заданные в блоке USING
- C (который идет после INTO) – таргет, в который будет записан результат выполнения запроса

4. Для имен таблиц и колонок можно (и даже правильнее) использовать функцию format():

```
EXECUTE format('SELECT count(*) FROM %I '
         'WHERE inserted_by = $1 AND inserted <= $2', tabname)
    INTO c
   USING checked_user, checked_date;
```

Для таких запросов необходимо помнить о NULL значениях и их обработке!

%I == quote_ident – оборачивание в кавычки, при условии их необходимости
%L == quote_nullable – корректная обработка NULL значений

ОСНОВЫ PL/pgSQL

5. Ветвление логики:

```
IF … THEN … [ELSEIF … THEN … ELSE …] END IF;
CASE выражения, по аналогии с обычным SQL
```

6. Использование циклов:

```
LOOP
  statements;
END LOOP;

LOOP
  statements;
  EXIT WHEN n > 100;     -- прерываем цикл, если выполнено условие
  CONTINUE WHEN n > 50; -- запускаем новую итерацию цикла, если выполнено условие
  more statements;
END LOOP;
```

6.1. Циклы WHILE:

```
WHILE boolean-expression
 LOOP
   statements;
 END LOOP;
```

ОСНОВЫ PL/pgSQL

6.2. Циклы FOR по целым числам:

```
-- цикл с заданными начальным и конечным значениями, шагом step в заданном порядке
FOR i IN [REVERSE] start_value .. end_value [BY step]
  LOOP
    statements;
  END LOOP;
```

6.2. Цикл FOR по результатам запроса:

```
FOR record_type_value IN query
  LOOP
    statements;
  END LOOP;
```

Для обращения к конкретному значению строки использовать «.»: `record_type_value.field_nm`

6.3. Цикл FOR по массиву:

```
FOREACH i IN ARRAY array_name
  LOOP
    statements;
  END LOOP;
```

ОСНОВЫ PL/pgSQL

И это все?

Конечно, нет
https://www.postgresql.org/docs/13/plpgsql.html

Не забываем про версию вашего postgres

```
select version();
```

ХРАНИМЫЕ ПРОЦЕДУРЫ

В PostgreSQL до 11 версии были только хранимые функции, которые все называли хранимыми функциями. В 11 появились хранимые процедуры.

Функция	Процедура
Возвращает 1 или несколько значений	Не возвращает никаких значений
1 функция – 1 транзакция, в рамках которой её запустили	В процедуре можно создавать транзакции, используя TCL
Запускается с использованием SELECT	Запускается с использованием CALL

ХРАНИМЫЕ ПРОЦЕДУРЫ

```
CREATE [ OR REPLACE ] PROCEDURE
  name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ]
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
} ...
```

ХРАНИМЫЕ ПРОЦЕДУРЫ

```
CREATE PROCEDURE insert_data(a integer, b integer)
LANGUAGE SQL
AS $$
  INSERT INTO tbl VALUES (a);
  INSERT INTO tbl VALUES (b);
$$;

CALL insert_data(1, 2);
```

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

ХРАНИМЫЕ ПРОЦЕДУРЫ

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

CALL transaction_test();
```

```
SELECT *
  FROM committed_table;

id
---
1
```

```
SELECT *
  FROM rollback_table;

---
ERROR: relation doesn’t
exist
```

ИЗМЕНЕНИЕ ПРОЦЕДУРЫ

```
ALTER FUNCTION name ([[argmode][argname] argtype [, ...]]) action [ ... ] [RESTRICT]
ALTER FUNCTION name ([[argmode][argname] argtype [, ...]]) RENAME TO new_name
ALTER FUNCTION name ([[argmode][argname] argtype [, ...]]) OWNER TO new_owner
ALTER FUNCTION name ([[argmode][argname] argtype [, ...]]) SET SCHEMA new_schema

action in:
    CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    IMMUTABLE | STABLE | VOLATILE | [NOT] LEAKPROOF
    [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER
    COST execution_cost
    ROWS result_rows
    SET configuration_parameter { TO | = } { value | DEFAULT }
    SET configuration_parameter FROM CURRENT
    RESET configuration_parameter
    RESET ALL

DROP FUNCTION [IF EXISTS] name ([[argmode][argname] argtype [, ...]])[CASCADE | RESTRICT]
```

ПРЕИМУЩЕСТВА ХРАНИМЫХ ПРОЦЕДУР

- Скорость
- Сокрытие структуры данных
- Гибкое управление правами доступа
- Меньшая вероятность SQL injection
- Повторное использование SQL
- Простая отладка SQL

НЕДОСТАТКИ ХРАНИМЫХ ПРОЦЕДУР

- Размазывание бизнес-логики
- Скудность языка СУБД
- Непереносимость хранимых функций
- Отсутствие необходимых навыков у команды и высокая «стоимость» соответствующих специалистов

ТРИГГЕР

— хранимая процедура особого типа, которую пользователь не вызывает непосредственно, а исполнение которой обусловлено действием по модификации данных: добавлением (INSERT), удалением (DELETE) строки в заданной таблице, или изменением (UPDATE) данных в определённом столбце заданной таблицы реляционной базы данных.

ТРИГГЕР

- применяется для обеспечения целостности данных и реализации сложной бизнес-логики
- запускается сервером автоматически при попытке изменения данных в таблице, с которой он связан
- в случае обнаружения ошибки или нарушения целостности данных может произойти откат транзакции

ТИПЫ ТРИГГЕРОВ

- Уровень срабатывания
  - ROW LEVEL – для каждой отдельной строки в таблице
  - STATEMENT LEVEL – для всех строк одной инструкции
- Событие срабатывания
  - UPDATE
  - DELETE
  - INSERT
  - TRUNCATE
- Время срабатывания
  - BEFORE
  - AFTER
  - INSTEAD OF

НАЗНАЧЕНИЕ ТРИГГЕРОВ

- Реализация обновляемых представлений
- Реализация бизнес логики
- Вспомогательные расчеты
- Системные процессы
  - Репликация, например
    - Для тех СУБД, которые не умеют
- Всё, что угодно
  - Накопление истории
  - Логирование

ТРИГГЕР

When	Event	Row-level	Statement-level
BEFORE	INSERT/UPDATE/DELETE	Tables	Tables and views
	TRUNCATE	—	Tables
AFTER	INSERT/UPDATE/DELETE	Tables	Tables and views
	TRUNCATE	—	Tables
INSTEAD OF	INSERT/UPDATE/DELETE	Views	—
	TRUNCATE	—	—

ТРИГГЕР

```
CREATE [CONSTRAINT] TRIGGER name {BEFORE | AFTER
| INSTEAD OF} { event [OR ...] }
  ON table
  [FROM referenced_table_name]
  [NOT DEFERRABLE | [DEFERRABLE] {INITIALLY
IMMEDIATE | INITIALLY DEFERRED}]
  [FOR [EACH] {ROW | STATEMENT} ]
  [WHEN (condition) ]
  EXECUTE PROCEDURE function_name (arguments)
```

ПЕРЕМЕННЫЕ ТРИГГЕРОВ

- OLD – старая строка для операций UPDATE/DELETE row-level триггеров
- NEW – новая строка для операций UPDATE/INERT row-level триггеров
- TG_NAME – имя сработавшего триггера
- TG_WHEN – тип события срабатывания триггера
- TG_LEVEL – уровень срабатывания триггера
- TG_OP – событие, вызвавшее триггер
- TG_TABLE_NAME – таблица, на которую сработал триггер
- TG_TABLE_SCHEMA – схема таблицы, на которую сработал триггер

ТРИГГЕР

```
CREATE TRIGGER check_update
   BEFORE UPDATE ON accounts
   FOR EACH ROW
   EXECUTE PROCEDURE check_account_update();
```

ТРИГГЕР

```
CREATE TRIGGER check_update
   BEFORE UPDATE OF balance ON accounts
   FOR EACH ROW
   EXECUTE PROCEDURE check_account_update();
```

ТРИГГЕР

```
CREATE TRIGGER check_update
   BEFORE UPDATE ON accounts
   FOR EACH ROW
   WHEN (OLD.balance IS DISTINCT FROM
NEW.balance)
   EXECUTE PROCEDURE check_account_update();
```

ТРИГГЕР

```
CREATE TRIGGER log_update
   AFTER UPDATE ON accounts
   FOR EACH ROW
   WHEN (OLD.* IS DISTINCT FROM NEW.*)
   EXECUTE PROCEDURE log_account_update();
```

ТРИГГЕР

```
CREATE TRIGGER view_insert
   INSTEAD OF INSERT ON my_view
   FOR EACH ROW
   EXECUTE PROCEDURE view_insert_row();
```

```
CREATE TABLE emp (
  empname text PRIMARY KEY,
  salary integer
);

CREATE TABLE emp_audit(
  operation char(1) NOT NULL,
  userid text NOT NULL,
  empname text NOT NULL,
  salary integer,
  stamp timestamp NOT NULL
);

CREATE VIEW emp_view AS
SELECT e.empname
     , e.salary
     , max(ea.stamp) AS last_updated
  FROM emp e
  LEFT JOIN emp_audit ea
    ON ea.empname = e.empname
 GROUP BY e.empname
        , e.salary;
```

```
CREATE OR REPLACE FUNCTION update_emp_view() RETURNS TRIGGER AS $$
 BEGIN
    -- Perform the required operation on emp, and create a row in emp_audit
    -- to reflect the change made to emp
    IF (TG_OP = 'DELETE') THEN
        DELETE FROM emp WHERE empname = OLD.empname;
        IF NOT FOUND THEN RETURN NULL; END IF;

        OLD.last_updated = now();
        INSERT INTO emp_audit VALUES('D', user, OLD.*);
        RETURN OLD;
    ELSIF (TG_OP = 'UPDATE') THEN
        UPDATE emp SET salary = NEW.salary WHERE empname = OLD.empname;
        IF NOT FOUND THEN RETURN NULL; END IF;

        NEW.last_updated = now();
        INSERT INTO emp_audit VALUES('U', user, NEW.*);
        RETURN NEW;
    ELSIF (TG_OP = 'INSERT') THEN
        INSERT INTO emp VALUES(NEW.empname, NEW.salary);

        NEW.last_updated = now();
        INSERT INTO emp_audit VALUES('I', user, NEW.*);
        RETURN NEW;
    END IF;
  END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_view
FOR EACH ROW
EXECUTE PROCEDURE update_emp_view();
```

ИЗМЕНЕНИЕ ТРИГГЕРА

```
ALTER TRIGGER name ON table RENAME TO new_name
```

```
ALTER TRIGGER emp_stamp ON emp RENAME TO
emp_track_chgs;
```

```
DROP TRIGGER [IF EXISTS] name ON table [CASCADE
| RESTRICT]
```

```
DROP TRIGGER if_dist_exists ON films;
```

ТРИГГЕР

- Достоинства:
  - Реализация сложной, событийно-ориентированной логики
  - Сокрытие алгоритмов обработки
  - Возможность вносить корректировки в работы системы не затрагивая основные механизмы

- Недостатки:
  - При сложной схеме данных логика растягивается на множество триггеров
  - Увеличение числа зависимостей между объектами
  - Усложнение отладки