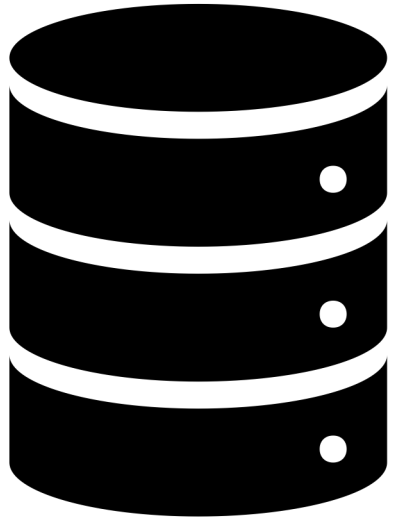
Базы данных



Лекция 7. Дополнительные возможности SQL.

Меркурьева Надежда
✉ merkurievanad@gmail.com
@merkurievanad

ФПМИ МФТИ, 2021

РОЛЕВАЯ МОДЕЛЬ

- В основе разграничений доступов PostgreSQL – ролевая модель
- Роль – множество пользователей БД
  - Может владеть объектами в базе
  - Может иметь определенный доступ к некоторым объектам базы, не являясь их владельцем
  - Может выдавать доступы на некоторые объекты в базе другим ролям
  - Может предоставлять членство в роли другой роли

До версии 8.1 были пользователи и группы, позднее – только роли

СОЗДАНИЕ ПОЛЬЗОВАТЕЛЯ

```
CREATE USER == CREATE ROLE
```

```
CREATE USER name [[WITH] option [...]]
```

where ***option*** can be:

```
    SUPERUSER | NOSUPERUSER
  | CREATEDB | NOCREATEDB
  | CREATEROLE | NOCREATEROLE
  | INHERIT | NOINHERIT
  | LOGIN | NOLOGIN
  | REPLICATION | NOREPLICATION
  | BYPASSRLS | NOBYPASSRLS
  | CONNECTION LIMIT connlimit
  | [ENCRYPTED] PASSWORD 'password' | PASSWORD NULL
  | VALID UNTIL 'timestamp'
  | IN ROLE role_name [, ...]
  | IN GROUP role_name [, ...]
  | ROLE role_name [, ...]
  | ADMIN role_name [, ...]
  | USER role_name [, ...]
  | SYSID uid
```

СОЗДАНИЕ ПОЛЬЗОВАТЕЛЯ

```
CREATE ROLE jonathan LOGIN;


CREATE USER davide
  WITH PASSWORD 'jw8s0F4’;


CREATE ROLE Miriam
  WITH LOGIN PASSWORD 'jw8s0F4’
  VALID UNTIL '2005-01-01';
```

DATA CONTROL LANGUAGE (DCL)

- Позволяет настраивать доступы к объектам
- Поддерживает 2 типа действий:
  - GRANT – выдача доступа к объекту
  - REVOKE – отмена доступа к объекту

DATA CONTROL LANGUAGE (DCL)

- Права, которые можно выдать на объект:
  - SELECT
  - INSERT
  - UPDATE
  - DELETE
  - TRUNCATE
  - REFERENCES
  - TRIGGER
  - CREATE
  - CONNECT
  - TEMPORARY
  - EXECUTE
  - USAGE

DCL: GRANT ON TABLE

```
GRANT {{SELECT | INSERT | UPDATE | DELETE | TRUNCATE |
        REFERENCES | TRIGGER}
    [, ...] | ALL [PRIVILEGES]}
    ON {[ TABLE] table_name [, ...]
        | ALL TABLES IN SCHEMA schema_name [, ...]}
    TO role_specification [, ...] [WITH GRANT OPTION]
```

DCL: REVOKE ON TABLE

```
REVOKE [GRANT OPTION FOR]
    {{SELECT | INSERT | UPDATE | DELETE | TRUNCATE |
      REFERENCES | TRIGGER}
    [, ...] | ALL [PRIVILEGES]}
    ON {[TABLE] table_name [, ...]
          | ALL TABLES IN SCHEMA schema_name [, ...]}
    FROM {[GROUP] role_name | PUBLIC} [, ...]
    [CASCADE | RESTRICT]
```

DCL: РАБОТА С ТАБЛИЦАМИ

```
GRANT ALL PRIVILEGES ON kinds TO manuel;
REVOKE ALL PRIVILEGES ON kinds FROM manuel;

GRANT SELECT ON kinds TO manuel WITH GRANT OPTION;
```

- Теперь manuel может выдавать права на **SELECT** на табличку `kinds`
- Если мы хотим забрать `grant option` у `manuel`, то нужно использовать **CASCADE**, чтобы забрать права у всех, кому он выдавал права. Иначе при попытке отозвать права у `manuel`, запрос упадет с ошибкой

DCL: GRANT ON SCHEMA

```
GRANT {{CREATE | USAGE} [, ...] | ALL [PRIVILEGES]}
    ON SCHEMA schema_name [, ...]
    TO role_specification [, ...] [WITH GRANT OPTION]
```

DCL: REVOKE ON SCHEMA

```
REVOKE [GRANT OPTION FOR]
    {{CREATE | USAGE} [, ...] | ALL [PRIVILEGES]}
    ON SCHEMA schema_name [, ...]
    FROM role_specification [, ...]
    [CASCADE | RESTRICT]
```

DCL: GRANT ON DATABASE

```
GRANT {{CREATE | CONNECT | TEMPORARY | TEMP} [, ...] |
ALL [PRIVILEGES]}
    ON DATABASE database_name [, ...]
    TO role_specification [, ...] [WITH GRANT OPTION]
```

DCL: REVOKE ON DATABASE

```
REVOKE [GRANT OPTION FOR]
    {{CREATE | CONNECT | TEMPORARY | TEMP} [, ...] | ALL
[PRIVILEGES]}
    ON DATABASE database_name [, ...]
    FROM role_specification [, ...]
    [CASCADE | RESTRICT]
```

DCL: ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ

```
CREATE USER davide WITH PASSWORD 'jw8s0F4';

GRANT CONNECT ON DATABASE db_prod TO davide;

GRANT USAGE ON SCHEMA my_schema TO davide;

GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA my_schema
TO davide;
```

DCL: ЧТО ДАЛЬШЕ?

- А дальше только https://www.google.com

CTE (COMMON TABLE EXPRESSION)

– именованный временный набор данных, используемый в запросе.

```
WITH cte_query_name
   AS (
        cte_query
   )
main_query;
```

CTE (COMMON TABLE EXPRESSION)

```
with regional_sales
  as (
      select region_id
           , sum(sales_amt) as total_sales_amt
        from orders
       group by region_id
  )
, top_region
  as (
      select region_id
        from regional_sales
       where total_sales_amt > (select sum(total_sales_amt) / 10
                                  from regional_sales)
  )
select region_id
     , product_id
     , sum(quantity_cnt) as product_unit_cnt
     , sum(sales_amt)    as product_sales_amt
  from orders  o
 inner join top_region tr
    on o.region_id = tr.region_id
 group by region_id
        , product_id;
```

CTE (COMMON TABLE EXPRESSION)

```
WITH RECURSIVE t(n) AS (
      VALUES (1)
   UNION ALL
      SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

Что делает этот запрос?

Необязательный модификатор RECURSIVE превращает WITH из просто удобной конструкции в уникальную функцию для получения сложных результатов. Используя RECURSIVE, запрос WITH может ссылаться на собственный вывод.

CTE (COMMON TABLE EXPRESSION)

- На самом деле RECURSIVE CTE является не рекурсией, а итерированием
- Как работает:
  - Выполняется «нерекурсивная часть». Результат сохраняется, а также записывается во временную ворковую таблицу.
  - Пока ворковая таблица не пуста:
    - Обрабатывается «рекурсивный» шаг с заменой ссылки CTE на себя ворковой таблицей.
    - Результат сохраняется, а также записывается во временную промежуточную таблицу.
    - Содержимое промежуточной таблицы заменяет содержимое ворковой таблицы. Промежуточная таблица очищается.

CTE (COMMON TABLE EXPRESSION)

```
WITH RECURSIVE t(n) AS (
      VALUES (1)
   UNION ALL
      SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

Сначала данные, для которых рекурсия не нужна

Через UNION (ALL) рекурсивная часть

Запрос на получение суммы целых чисел от 1 до 100

CTE (COMMON TABLE EXPRESSION)

```
WITH RECURSIVE t(n) AS (
      VALUES (1)
   UNION ALL
      SELECT n+1 FROM t
)
SELECT sum(n) FROM t LIMIT 100;
```

Сначала данные, для которых рекурсия не нужна

Через UNION (ALL) рекурсивная часть

Запрос на получение суммы целых чисел от 1 до 100

Такой вариант записи не используйте!

CTE (COMMON TABLE EXPRESSION)

```
WITH RECURSIVE t(n)
  AS (
       VALUES (1)
        UNION ALL
       SELECT n+1
         FROM t
        WHERE n < 100
)
SELECT sum(n)
  FROM t;
```

Объявляем CTE рекурсивной и задаем названия параметров

Задаем нерекурсивную часть – ровно 1 значение

Описываем логику рекурсивной части

На каждом шаге в ворковой таблице 1 строка
На каждом шаге в результат записывается 1 строка
Итог рекурсии – значения от 1 до 100 с шагом 1

Итог запроса – сумма целых чисел от 1 до 100

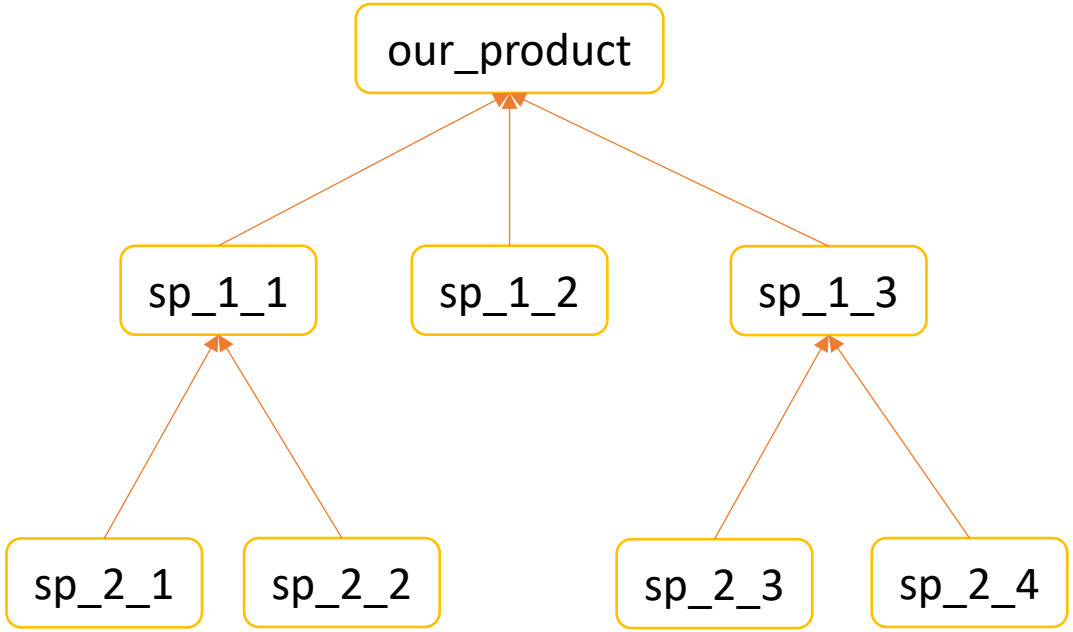
CTE (COMMON TABLE EXPRESSION)

sub_part	part	quantity
sub_part_1_1	our_product	2
sub_part_1_2	our_product	1
sub_part_1_3	our_product	2
sub_part_2_1	sub_part_1_1	2
sub_part_2_2	sub_part_1_1	3
sub_part_2_3	sub_part_1_3	1
sub_part_2_4	sub_part_1_3	1

Из какого общего числа составляющих складывается наш продукт?

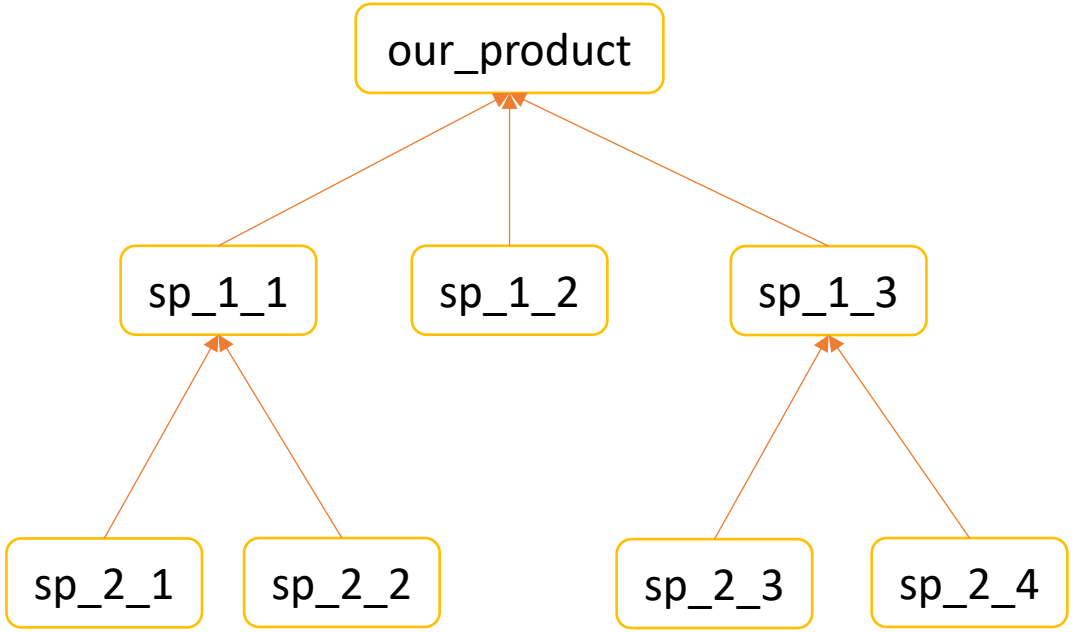
CTE (COMMON TABLE EXPRESSION)

sub_part	part	quantity
sub_part_1_1	our_product	2
sub_part_1_2	our_product	1
sub_part_1_3	our_product	2
sub_part_2_1	sub_part_1_1	2
sub_part_2_2	sub_part_1_1	3
sub_part_2_3	sub_part_1_3	1
sub_part_2_4	sub_part_1_3	1



CTE (COMMON TABLE EXPRESSION)

sub_part	part	quantity
sub_part_1_1	our_product	2
sub_part_1_2	our_product	1
sub_part_1_3	our_product	2
sub_part_2_1	sub_part_1_1	2
sub_part_2_2	sub_part_1_1	3
sub_part_2_3	sub_part_1_3	1
sub_part_2_4	sub_part_1_3	1



quantity(sp_2_1)*quantity(sp_1_1) + quantity(sp_2_2)*quantity(sp_1_1) +
+ quantity(sp_2_3)*quantity(sp_1_3) + quantity(sp_2_4)*quantity(sp_1_3) +
+ quantity(sp_1_1) + quantity(sp_1_2) + quantity(sp_1_3)

CTE (COMMON TABLE EXPRESSION)

```
WITH RECURSIVE included_parts(sub_part, part, quantity)
  AS (
       SELECT sub_part, part, quantity
         FROM parts
        WHERE part = 'our_product'
        UNION ALL
       SELECT p.sub_part, p.part, p.quantity * ip.quantity
         FROM included_parts ip
        INNER JOIN  parts p
           ON p.part = ip.sub_part
  )
SELECT sub_part, sum(quantity) as total_quantity
  FROM included_parts
 GROUP BY sub_part
```

CTE (COMMON TABLE EXPRESSION)

```
WITH moved_rows AS (
         DELETE FROM products
          WHERE "date" >= '2010-10-01'
            AND "date" < '2010-11-01'
      RETURNING * )
INSERT INTO products_log
SELECT *
  FROM moved_rows;
```

Удаляем данные из таблички products

Те же самые данные, что удалили ранее, записываем в products_log

```
WITH t AS (
         UPDATE products
            SET price = price * 1.05
      RETURNING * )
SELECT *
  FROM t;
```

Проиндексировали все цены в табличке products

Сразу вывели обновленные данные

ПРЕДСТАВЛЕНИЕ (VIEW)

– это виртуальная (логическая) таблица, представляющая собой поименованный запрос (синоним к запросу), который будет подставлен как подзапрос при использовании представления.

- Не является самостоятельной частью набора данных
- Вычисляется динамически на основании данных, хранящихся в реальных таблицах
- Изменение данных в таблицах немедленно отражается в содержимом представлений

ПРЕДСТАВЛЕНИЕ (VIEW)

```
CREATE [OR REPLACE][TEMP | TEMPORARY] [RECURSIVE]
VIEW view_name [(column_name [,…])]
  AS query;
```

ПРЕДСТАВЛЕНИЕ (VIEW)

- **`CREATE VIEW`** – создание нового представления
- **`CREATE OR REPLACE VIEW`** – создание или замена уже существующего представления
  - В случае замены в новом представлении должны присутствовать все поля старого представления (имена, порядок, тип данных). Допустимо только добавление новых полей
- **`TEMPORARY | TEMP`** – временное представление, будет существовать до конца сессии
- *`view_name`* – название представления
- *`column_name`* – *список полей представления. Если не указан, используются поля запроса*
- *`query`* – **`SELECT`** или **`VALUES`** команды

ПРЕДСТАВЛЕНИЕ (VIEW)

```
CREATE VIEW v_test AS
SELECT 'Hello World';
```

Как делать не надо

Как делать правильно

```
CREATE VIEW v_test AS
SELECT 'Hello World'::text AS hello;
```

Зафиксировали тип

Зафиксировали название поля

ПРЕДСТАВЛЕНИЕ (VIEW)

```
CREATE VIEW v_comedy AS
SELECT *
  FROM film
 WHERE genre_nm = 'Comedy';
```

ПРЕДСТАВЛЕНИЕ (VIEW)

```
CREATE RECURSIVE VIEW public.nums_1_100(n) AS
  VALUES (1)
   UNION ALL
  SELECT n+1
    FROM nums_1_100
   WHERE n < 100;
```

ПРЕДСТАВЛЕНИЕ (VIEW)

- Горизонтальное представление:
  - Ограничение данных по строкам

```
CREATE VIEW V_IT_EMPLOYEE AS
SELECT *
  FROM EPLOYEE
 WHERE DEPARTMENT_NM = 'IT';
```

- Вертикальное представление
  - Ограничение данных по столбцам

```
CREATE VIEW V_EMP AS
SELECT EMP_NM, DEPARTMENT_NM
  FROM EPLOYEE;
```

EMP_ID	EMP_NM	DEPARTMENT_NM	SALARY_AMT
1	Иванов И.И.	IT	100000
135	Николаев С.Т.	IT	123000
16	Терентьев А.П.	IT	56000

EMP_NM	DEPARTMENT_NM
Иванов И.И.	IT
Степанов Р.В.	R&D
Николаев С.Т.	IT
Медведев И.А.	SALES
Терентьев А.П.	IT

ПРЕДСТАВЛЕНИЕ (VIEW)

```
CREATE VIEW name [ ( column_name [, ...] ) ]
    AS query
[WITH [CASCADED | LOCAL] CHECK OPTION]
```

- CHECK OPTION
  - LOCAL
  - CASCADED

ОБНОВЛЯЕМОЕ ПРЕДСТАВЛЕНИЕ

- Представление называется *обновляемым*, если к нему применимы операции UPDATE и DELETE для изменения данных в таблицах, на которых построено это представление.

- Для того, чтобы представление было обновляемым, должно быть выполнено 2 условия:
  - Соответствие 1-1 между строками представления и таблиц, на которых основано представление
  - Поля представления должны быть простым перечислением полей таблиц

ОБНОВЛЯЕМОЕ ПРЕДСТАВЛЕНИЕ

- Обновляемое представление, основанное на нескольких таблицах, может обновлять только одну таблицу за запрос.

- Как же быть?
  - Явно указывать, значения в каких столбцах вы хотите обновить
  - За одну `UPDATE` операцию указывать только те столбцы, которые принадлежат одной таблице-источнику
  - `DELETE` для таких представлений не поддерживается
  - `INSERT` работает, только если вставка происходит в единственную реальную таблицу

ОБНОВЛЯЕМОЕ ПРЕДСТАВЛЕНИЕ

По версии PostgreSQL:

- VIEW является обновляемым, если
  - Ровно 1 источник в блоке FROM
  - Отсутствуют выражения WITH, DISTINCT, GROUP BY, HAVING, LIMIT
  - Отсутствуют теоретико-множественные операции
  - В SELECT нет агрегатов, оконных или возвращающих несколько значений функций
- Колонка VIEW является обновляемой, если:
  - Она ссылается на колонку таблицы без преобразований

ПРЕДСТАВЛЕНИЯ (VIEW)

Для обновляемых представлений

```
CREATE VIEW name [ ( column_name [, ...] ) ]
    AS query
[ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

- `CHECK OPTION` – дополнительная проверка всех `UPDATE` и `INSERT` операций
  - `CASCADED` – проверка целостности этого представления и зависимых представлений
  - `LOCAL` – проверка целостности только этого представления

VIEW: CASCADED CHECK OPTION

```
CREATE VIEW v_city_a AS
SELECT city_id,
       city,
       country_id
  FROM city
 WHERE city LIKE 'A%';
```

```
CREATE VIEW v_city_a_usa AS
SELECT city_id,
       city,
       country_id
  FROM v_city_a
 WHERE country_id = 103
WITH CASCADED CHECK OPTION;
```

При вставке в `v_city_a_usa` будет проверяться только условие из WHERE , т.е. `country_id = 103`

`INSERT INTO` `city_a_usa (city, country_id)` **`VALUES`**`('Houston', 103);` упадет с ошибкой именно на этапе вставки в `v_city_a`, т.к. не выполняется `city` **`LIKE`** `'A%';`

Если заменить **`CASCADED CHECK OPTION`** на **`LOCAL CHECK OPTION`**, то вставка отработает!

ИЗМЕНЕНИЕ ПРЕДСТАВЛЕНИЙ

```
ALTER VIEW [IF EXISTS] name ALTER [COLUMN] column_name SET DEFAULT expression
ALTER VIEW [IF EXISTS] name ALTER [COLUMN] column_name DROP DEFAULT
ALTER VIEW [IF EXISTS] name OWNER TO new_owner
ALTER VIEW [IF EXISTS] name RENAME TO new_name
ALTER VIEW [IF EXISTS] name SET SCHEMA new_schema
ALTER VIEW [IF EXISTS] name SET ( view_option_name [= view_option_value] [, ... ] )
ALTER VIEW [IF EXISTS] name RESET ( view_option_name [, ... ] )

DROP VIEW [IF EXISTS] name [, ...] [ CASCADE | RESTRICT ]
```

ПРЕДСТАВЛЕНИЕ (VIEW)

Что может содержать в себе представление?

- Подмножество записей из таблицы БД, отвечающее определённым условиям
- Подмножество столбцов таблицы БД, требуемое программой
- Результат обработки данных таблицы определёнными операциями
- Результат соединения (join) нескольких таблиц
- Результат слияния нескольких таблиц с одинаковыми именами и типами полей, когда в представлении попадают все записи каждой из сливаемых таблиц
- Результат группировки записей в таблице
- Практически любую комбинацию вышеперечисленных возможностей

ПРЕДСТАВЛЕНИЕ (VIEW)

Зачем это вообще кому-то нужно?

- Представления скрывают от прикладной программы сложность запросов и саму структуру таблиц БД
- Использование представлений позволяет отделить прикладную схему представления данных от схемы хранения
- С помощью представлений обеспечивается ещё один уровень защиты данных
- Выигрыш во времени за счет оптимизации

ПРЕИМУЩЕСТВА ПРЕДСТАВЛЕНИЙ

- **Безопасность:** можно искусственно ограничивать информацию, к которой у пользователя есть доступ
- **Простота запросов:** при написании запросов обращаемся к вью, как и к обычной таблице
- **Защита от изменений:** пользователю не обязательно знать, что структуры / имена таблиц поменялись. Достаточно обновить представление

НЕДОСТАТКИ ПРЕДСТАВЛЕНИЙ

- **Производительность:** кажущийся простым запрос с использованием вью на деле может оказаться очень сложным из-за логики, зашитой во вью
- **Управляемость:** вью может быть основана на вью, которая в свою очередь тоже основана на другой вью и т.д.
- **Ограничение на обновление:** не любую вью можно обновить, что не всегда очевидно пользователю