Работа с PostgreSQL. Часть 2

Занятие 1.6

Алексей Кузьмин
Директор разработки, Data Scientist ДомКлик.ру

Алексей Кузьмин

Директор разработки, Data Scientist
ДомКлик

Что сегодня изучим

1. Представления
2. Схемы запросов
3. Ускорение запросов. Индексы
4. Сложные типы данных
5. Массивы

Представления

1

Алексей Кузьмин
Директор разработки, Data Scientist ДомКлик.ру

View — это именованные запросы, которые помогают сделать представление (именно вид) данных, лежащий в таблицах PostgreSQL

- View основывается на одной или нескольких базовых таблицах. Удобны для часто используемых запросов

- View (кроме materialized view) не хранят данные

Создание

```
CREATE VIEW view_name AS query;
```

Пример с информацией для покупателя

```
SELECT cu.customer_id AS id,
    cu.first_name || ' ' || cu.last_name AS name,
    a.address,
    a.postal_code AS "zip code",
    a.phone,
    city.city,
    country.country,
        CASE
            WHEN cu.activebool THEN 'active'
            ELSE ''
        END AS notes,
    cu.store_id AS sid
   FROM customer cu
     INNER JOIN address a USING (address_id)
     INNER JOIN city USING (city_id)
     INNER JOIN country USING (country_id);
```

И теперь

Для получения данных о покупателях можно использовать простой Select

```
SELECT
 *
FROM
 customer_master;
```

Изменение

```
CREATE OR REPLACE view_name
AS
query
```

Удаление

```
DROP VIEW [ IF EXISTS ]
view_name;
```

Различия CTE и VIEW

1

Представления могут быть проиндексированы, но ОТВ не могут

2

ОТВ отлично работают с рекурсией

3

Представления — физические объекты БД, можно обращаться из нескольких запросов:

- гибкость
- централизованный подход

4

ОТВ — временные:

- создаются, когда будут использоваться
- удаляются после использования
- не хранится статистика на сервере

Время практики

Практика 1

Создайте view с колонками клиент (ФИО; email) и title фильма, который он брал в прокат последним

Практика 1. Решение

```
CREATE VIEW practice_1 AS
    WITH cte AS (
            SELECT r.*, ROW_NUMBER() OVER (PARTITION BY r.customer_id
ORDER BY r.rental_date DESC)
            FROM rental r
    )
    SELECT c.last_name, c.email, f.title
    FROM cte
    JOIN customer c ON c.customer_id = cte.customer_id
    JOIN inventory i ON i.inventory_id = cte.inventory_id
    JOIN film f ON f.film_id = i.film_id
    WHERE row_number = 1
```

Материализованное представление

- Хранит результат запроса. За счёт этого доступ к информации происходит быстрее, но материализованное представление надо периодически обновлять
- WITH DATA — загрузить данные сразу, WITH NO DATA — позже

```
CREATE MATERIALIZED VIEW
view_name
AS
query
WITH [NO] DATA;
```

Обновление

```
REFRESH MATERIALIZED VIEW
view_name;
```

Удаление

```
DROP MATERIALIZED VIEW
view_name;
```

Схема запроса

2

Алексей Кузьмин
Директор разработки, Data Scientist ДомКлик.ру

Оператор EXPLAIN демонстрирует этапы выполнения запроса и может быть использован для оптимизации

Давайте рассмотрим предыдущий запрос:

```
EXPLAIN ANALYZE
SELECT f.title, CONCAT(a.last_name, ' ', a.first_name)
FROM film f
JOIN film_actor fa ON fa.film_id = f.film_id
JOIN actor a ON fa.actor_id = a.actor_id
WHERE f.film_id < 100
```

Результат

```
Hash Left Join  (cost=23.58..125.42 rows=546 width=47)
  Hash Cond: (fa.actor_id = a.actor_id)
  ->  Hash Right Join  (cost=17.08..116.09 rows=546 width=17)
        Hash Cond: (fa.film_id = f.film_id)
        ->  Seq Scan on film_actor fa  (cost=0.00..84.62 rows=5462 width=4)
        ->  Hash  (cost=15.83..15.83 rows=100 width=19)
              ->  Index Scan using film_pkey on film f  (cost=0.28..15.83 rows=100 width=19)
                    Index Cond: (film_id < 100)
  ->  Hash  (cost=4.00..4.00 rows=200 width=17)
        ->  Seq Scan on actor a  (cost=0.00..4.00 rows=200 width=17)
```

План запроса можно посмотреть планируемый, фактический или в разных форматах

Синтаксис EXPLAIN приведён ниже:

```
EXPLAIN [ ( параметр [, ...] ) ] оператор
EXPLAIN [ ANALYZE ] [ VERBOSE ] оператор
Здесь допускается параметр:
    ANALYZE [ boolean ]
    VERBOSE [ boolean ]
    COSTS [ boolean ]
    BUFFERS [ boolean ]
    TIMING [ boolean ]
    FORMAT { TEXT | XML | JSON | YAML }
```

Ускорение запросов. Индексы

3

Алексей Кузьмин
Директор разработки, Data Scientist ДомКлик.ру

Представим, что в таблице film отсутствует индекс по столбцу film_id:

```
EXPLAIN ANALYZE
SELECT f.title, CONCAT(a.last_name, ' ', a.first_name)
FROM film f
JOIN film_actor fa ON fa.film_id = f.film_id
JOIN actor a ON fa.actor_id = a.actor_id
WHERE f.film_id < 100
```

Результат

```
->  Seq Scan on film f  (cost=0.00...66.50 rows=100 width=19) (actual
time=0.007..0.134 rows=99 loops=1)
        Filter: (film_id < 100)
        Rows Removed by Filter: 901
```

Ускорить запрос можно с помощью создания индексов. Индексы можно создавать на лету

```
CREATE INDEX ON film(film_id);
```

После того как индекс создан — запросы начинают выполняться бодрее, время сокращается в разы

Результат

```
->  Index Scan using film_pkey on film f  (cost=0.28..15.83 rows=100 width=19)
(actual time=0.005..0.028 rows=99 loops=1)
      Index Cond: (film_id < 100)
```

Сложные типы данных

4

Алексей Кузьмин
Директор разработки, Data Scientist ДомКлик.ру

JSON

JSON — JavaScript Object Notation. JSON — де-факто стандарт для хранения данных в виде key-value пар

Рассмотрим простой пример:

```
CREATE TABLE orders (
ID serial NOT NULL PRIMARY KEY,
 info json NOT NULL
);
```

```
INSERT INTO orders (info)
VALUES
 (
 '{ "customer": "John Doe", "items": {"product": "Beer","qty": 6}}'
 ),
 (
 '{ "customer": "Lily Bush", "items": {"product": "Diaper","qty": 24}}'
 ),
 (
 '{ "customer": "Josh William", "items": {"product": "Toy Car","qty": 1}}'
 ),
 (
 '{ "customer": "Mary Clark", "items": {"product": "Toy Train","qty": 2}}'
 );
```

Запрос данных:

```
SELECT
 info
FROM
orders;
```

Postgres возвращает ответ в виде типа JSON. Для работы с ним есть 2 специальных оператора -> и ->>
-> возвращает результат в виде JSON-объекта ->> возвращает результат в виде текста

Получить имена всех покупателей:

```
SELECT
info ->> 'customer' AS customer
FROM
 orders;
```

т. к. оператор -> возвращает JSON-объект, то к его результату можно снова применять оператор -> и ->>.

Пример:

```
SELECT
 info -> 'items' ->> 'product' as product
FROM
 orders
ORDER BY
 product;
```

Ещё пример. Поиск людей, которые купили 2 продукта:

```
SELECT
 info ->> 'customer' AS customer,
 info -> 'items' ->> 'product' AS product
FROM
 orders
WHERE
 CAST (
 info -> 'items' ->> 'qty' AS INTEGER
 ) = 2
```

Время практики

Практика 2

- Создайте таблицу orders — скрипт выше в лекции
- Выведите общее количество заказов

Практика 2. Решение

```
SELECT SUM((info->'items'->>'qty')::int)
FROM orders
```

Массивы

5

Алексей Кузьмин
Директор разработки, Data Scientist ДомКлик.ру

Массив — это коллекция элементов. В одной колонке вы можете хранить несколько атрибутов одного типа

Пример:

```
CREATE TABLE contacts (
id SERIAL PRIMARY KEY,
 name VARCHAR (100),
 phones TEXT []
);
```

Вставка данных

```
INSERT INTO contacts (name, phones)
VALUES
(
'John Doe',
ARRAY [ '(408)-589-5846',
'(408)-589-5555' ]
);
```

Или

```
INSERT INTO contacts (name, phones)
VALUES
(
'John Doe',
'{ "(408)-589-5846", "(408)-589-5555" }'
);
```

Выборка данных:

```
SELECT
 name,
 phones
FROM
 contacts;
```

Запрос конкретного элемента массива:

```
SELECT
 name,
 phones [ 1 ]
FROM
 contacts;
```

Индексы начинаются с 1.
Но есть возможность указать и отрицательные числа в индексах.

Работа с массивами. Функции

Возвращает значение индекса первого вхождения:

```
SELECT ARRAY_POSITION(ARRAY['a','b','c','d'], 'b') -- 2
```

Возвращает длину указанной размерности массива:

```
SELECT ARRAY_LENGTH(ARRAY['a','b','c','d'], 1) -- 4
```

Содержит:

```
SELECT ARRAY[1,4,3] @> ARRAY[3,1,3] -- true
```

Содержится в:

```
SELECT ARRAY[2,2,7] <@ ARRAY[1,7,4,2,6] -- true
```

Пересечение (есть общие элементы):

```
SELECT ARRAY[1,4,3] && ARRAY[2,1] -- true
```

Полезные материалы

Алексей Кузьмин
Директор разработки, Data Scientist ДомКлик.ру

Полезные материалы

- https://habr.com/ru/post/269497/
- https://medium.com/@hakibenita/be-careful-with-cte-in-postgresql-fca5e24d2119 актуально для PostgreSQL версии 10 и ниже
- https://postgrespro.ru/docs/postgrespro/9.5/using-explain
- https://habr.com/ru/post/203320/

Спасибо за внимание!

Алексей Кузьмин
Директор разработки, Data Scientist ДомКлик.ру