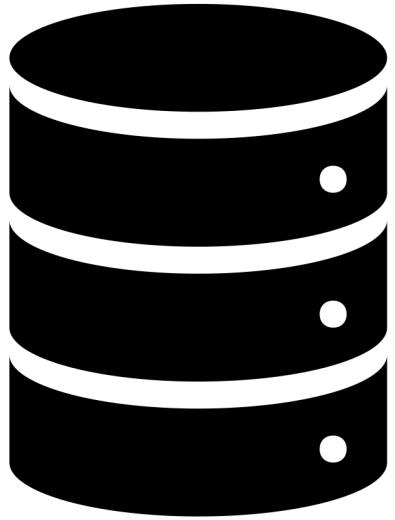

Базы данных

Лекция 11

Оптимизация запросов. Часть 2.

Индексы. Оптимизация.

Меркурьева Надежда
✉ merkurievanad@gmail.com
@merkurievanad

ФПМИ МФТИ, 2021

Планы на сегодня

- Индексы
- Оптимизация запросов

Всё на примере PostgeSQL.

Индексы

Индексы в PostgreSQL — специальные объекты базы данных, предназначенные в основном для ускорения доступа к данным. Это вспомогательные структуры: любой индекс можно удалить и восстановить заново по информации в таблице. Индексы служат также для поддержки некоторых ограничений целостности.

В настоящее время в PostgreSQL 9.6 встроены шесть разных видов индексов (но мы не будем рассматривать все).

Индексы

- Все индексы — вторичные, они отделены от таблицы. Вся информация о них содержится в системном каталоге.
- Индексы связывают ключи и TID
(tuple id - #page: offset) ← указатель на строку.
- Индексы могут быть многоколончатыми.
- Индексы не содержат информации о видимости.
- Любое обновление записи в таблице приводит к появлению новой записи в индексе.
- Обновление полей таблицы, по которым не создавались индексы, не приводит к перестроению индексов (Heap-Only Tuples).

Индексы: условия использования

- Совпадают оператор и типы аргументов.
- Индекс валиден.
- В многоколончатом индексе важен порядок.
- План с его использованием — оптимален (минимальная стоимость).
- Всю информацию Postgres берет из системного каталога.

CREATE INDEX

```
CREATE [UNIQUE] INDEX [CONCURRENTLY] [name] ON table_name [USING method]
       ({column_name|(expression)} [COLLATE collation] [opclass] [ASC|DESC]
       [NULLS {FIRST|LAST}] [, ...])
       [WITH (storage_parameter = value [, ...])]
       [TABLESPACE tablespace_name]
       [WHERE predicate]


ALTER INDEX [IF EXISTS] name RENAME TO new_name
ALTER INDEX [IF EXISTS] name SET TABLESPACE tablespace_name
ALTER INDEX [IF EXISTS] name SET (storage_parameter = value [, ... ])
ALTER INDEX [IF EXISTS] name RESET (storage_parameter [, ... ])

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

Индексы: способы сканирования

```
postgres=# create table t(a integer, b text, c boolean);
CREATE TABLE

postgres=# insert into t(a,b,c)
  select s.id, chr((32+random()*94)::integer), random() < 0.01
  from generate_series(1,100000) as s(id)
  order by random();
INSERT 0 100000

postgres=# create index on t(a);
CREATE INDEX

postgres=# analyze t;
ANALYZE
```

Мы создали таблицу с тремя полями.

Первое поле содержит числа от 1 до 100 000, и по нему создан индекс (пока нам не важно, какой именно).

Второе поле содержит различные ASCII-символы, кроме непечатных.

Наконец, третье поле содержит логическое значение, истинное примерно для 1% строк, и ложное для остальных. Строки вставлены в таблицу в случайном порядке.

Индексы: способы сканирования

```
postgres=# explain (costs off) select * from t
where a = 1;

           QUERY PLAN
-------------------------------
 Index Scan using t_a_idx on t
   Index Cond: (a = 1)
(2 rows)
```

Попробуем выбрать значение по условию «`a = 1`».

Заметим, что условие имеет вид «*индексированное-поле оператор выражение*», где в качестве *оператора* используется «равно», а *выражением* (ключом поиска) является «1».

В большинстве случаев условие должно иметь именно такой вид, чтобы индекс мог использоваться.

В данном случае оптимизатор принял решение использовать *индексное сканирование* (Index Scan). При индексном просмотре метод доступа возвращает значения TID по одному, до тех пор, пока подходящие строки не закончатся. Механизм индексирования по очереди обращается к тем страницам таблицы, на которые указывают TID, получает версию строки, проверяет ее видимость в соответствии с правилами многоверсионности, и возвращает полученные данные.

Индексы: способы сканирования

Индексное сканирование хорошо работает, когда речь идет всего о нескольких значениях. Однако при увеличении выборки возрастают шансы, что придется возвращаться к одной и той же табличной странице несколько раз. Поэтому в таком случае оптимизатор переключается на *сканирование по битовой карте* (bitmap scan):

```
postgres=# explain (costs off) select * from t
where a <= 100;
             QUERY PLAN
-----------------------------------
 Bitmap Heap Scan on t
   Recheck Cond: (a <= 100)
   ->  Bitmap Index Scan on t_a_idx
         Index Cond: (a <= 100)
(4 rows)
```

Сначала метод доступа возвращает все TID, соответствующие условию (узел Bitmap Index Scan), и по ним строится битовая карта версий строк.

Затем версии строк читаются из таблицы (Bitmap Heap Scan) — при этом каждая страница будет прочитана только один раз.

Обратите внимание, что на втором шаге условие может перепроверяться (Recheck Cond). Выборка может оказаться слишком велика, чтобы битовая карта версий строк могла целиком поместиться в оперативную память (ограниченную параметром work_mem). В этом случае строится только битовая карта *страниц,* содержащих хотя бы одну подходящую версию строки. Такая «грубая» карта занимает меньше места, но при чтении страницы приходится перепроверять условия для каждой хранящейся там строки.

Индексы: способы сканирования

Если условия наложены на несколько полей таблицы, и эти поля проиндексированы, сканирование битовой карты позволяет (если оптимизатор сочтет это выгодным) использовать несколько индексов одновременно. Для каждого индекса строятся битовые карты версий строк, которые затем побитово логически умножаются (если выражения соединены условием `AND`), либо логически складываются (если выражения соединены условием `OR`):

```
postgres=# create index on t(b);
CREATE INDEX

postgres=# analyze t;
ANALYZE

postgres=# explain (costs off) select * from t
where a <= 100 and b = 'a';
                    QUERY PLAN
-------------------------------------------------
 Bitmap Heap Scan on t
   Recheck Cond: ((a <= 100) AND (b = 'a'::text))
   ->  BitmapAnd
         ->  Bitmap Index Scan on t_a_idx
               Index Cond: (a <= 100)
         ->  Bitmap Index Scan on t_b_idx
               Index Cond: (b = 'a'::text)
(7 rows)
```

Здесь узел BitmapAnd объединяет две битовые карты с помощью битовой операции «и».

Индексы: способы сканирования

Что, если данные в страницах таблицы физически упорядочены точно так же, как и записи индекса? Нельзя полностью полагаться на физический порядок данных в страницах — если нужны отсортированные данные, в запросе необходимо явно указывать предложение `ORDER BY`. Но вполне возможны ситуации, в которых на самом деле «почти все» данные упорядочены: например, если строки добавляются в нужном порядке и не изменяются после этого, или после выполнения команды `CLUSTER`. Тогда построение битовой карты — лишний шаг, обычное индексное сканирование будет таким же. Поэтому при выборе метода доступа планировщик заглядывает в специальную статистику, которая показывает степень упорядоченности данных:

```
postgres=# select attname, correlation from pg_stats
where tablename = 't';

 attname | correlation
---------+-------------
 b       |    0.533512
 c       |    0.942365
 a       | -0.00768816
(3 rows)
```

Значения, близкие по модулю к единице, говорят о высокой упорядоченности (как для столбца c), а близкие к нулю — наоборот, о хаотичном распределении (столбец a).

Индексы: способы сканирования

```
postgres=# explain (costs off) select * from t
where a <= 40000;
       QUERY PLAN
------------------------
 Seq Scan on t
   Filter: (a <= 40000)
(2 rows)
```

Для полноты картины следует сказать, что при неселективном условии оптимизатор предпочтет использованию индекса *последовательное сканирование* таблицы целиком:

Индексы работают тем лучше, чем выше селективность условия, то есть чем меньше строк ему удовлетворяет. При увеличении выборки возрастают и накладные расходы на чтение страниц индекса.

Ситуация усугубляется тем, что последовательное чтение выполняется быстрее, чем чтение страниц «вразнобой». Это особенно верно для жестких дисков, где механическая операция подведения головки к дорожке занимает существенно больше времени, чем само чтение данных; в случае дисков SSD этот эффект менее выражен.

Индексы: способы сканирования

```
postgres=# vacuum t;
VACUUM

postgres=# explain (costs off) select a from t
where a < 100;
             QUERY PLAN
------------------------------------
 Index Only Scan using t_a_idx on t
   Index Cond: (a < 100)
(2 rows)
```

Как правило, основная задача метода доступа — вернуть идентификаторы подходящих строк таблицы, чтобы механизм индексирования мог прочитать из них необходимые данные. Но что, если индекс уже содержит все необходимые для запроса данные?

Такой индекс называется *покрывающим* (covering), и в этом случае оптимизатор может применить *исключительно индексное сканирование* (Index Only Scan):

Название может навести на мысль, что механизм индексирования совсем не обращается к таблице, получая всю необходимую информацию исключительно от метода доступа. Но это не совсем так, потому что индексы в PostgreSQL не содержат информации, позволяющей судить о видимости строк. Поэтому метод доступа возвращает все версии строк, попадающие под условие поиска, независимо от того, видны они текущей транзакции или нет.

Индексы: способы сканирования

Неопределенные значения играют важную роль в реляционных базах данных как удобный способ представления того факта, что значение не существует или не известно.

Но особое значение требует и особого к себе отношения. Обычная булева логика превращается в трехзначную; непонятно, должно ли неопределенное значение быть меньше обычных значений или больше (отсюда специальные конструкции для сортировки `NULLS FIRST` и `NULLS LAST`); не очевидно, надо ли учитывать неопределенные значения в агрегатных функциях или нет; требуется специальная статистика для планировщика…

С точки зрения индексной поддержки с неопределенными значениями тоже имеется неясность: надо ли индексировать такие значения или нет? Если не индексировать `NULL`, то индекс может получиться компактнее. Зато если индексировать, то появляется возможность использовать индекс для условий вида «*индексированное-поле* `IS [NOT] NULL`», а также в качестве покрывающего индекса при полном отсутствии условий на таблицу (поскольку в этом случае индекс должен вернуть данные всех строк таблицы, в том числе и с неопределенными значениями).

Индексы: способы сканирования

```
postgres=# create index on t(a,b);
CREATE INDEX

postgres=# analyze t;
ANALYZE
```

Условия на несколько полей могут быть поддержаны с помощью *многоколоночных индексов*. Например, мы могли бы создать индекс по двум полям нашей таблицы:

```
postgres=# explain (costs off) select * from t where a
<= 100 and b = 'a';
                    QUERY PLAN
-----------------------------------------------
 Index Scan using t_a_b_idx on t
   Index Cond: ((a <= 100) AND (b = 'a'::text))
(2 rows)
```

Оптимизатор скорее всего предпочтет такой индекс объединению битовых карт, поскольку здесь мы сразу получаем нужные TID без каких-либо вспомогательных действий:

```
postgres=# explain (costs off) select * from t where a
<= 100;
               QUERY PLAN
-----------------------------------
 Bitmap Heap Scan on t
   Recheck Cond: (a <= 100)
   ->  Bitmap Index Scan on t_a_b_idx
         Index Cond: (a <= 100)
(4 rows)
```

Многоколоночный индекс может использоваться и для ускорения выборки по условию на часть полей — начиная с первого:

Индексы: почти конец

Что осталось за бортом:

- Функциональные индексы
- Частичные индексы
- Сортировки
- Параллельное построение
- Интерфейсы доступа

Где почитать/посмотреть:

- https://habrahabr.ru/company/postgrespro/blog/326096/

Оптимизация запросов

- Есть много малоизвестных специфичных для PostgreSQL приемов оптимизации запросов.
- В обычных презентациях и курсах посвященных оптимизации запросов они обычно не рассмотрены.

Оптимизация запросов: полезности

`VALUES` или генерация псевдо-таблицы из набора данных
(например для использования в `JOIN` вместо `IN`):

```
SELECT * FROM (VALUES (29),(68),(45),(47),(50)) AS t(author);

author
--------
29
68
45
47
50
```

Оптимизация запросов: полезности

Многоколоночный `VALUES`:

```
SELECT * FROM (VALUES (29, 'a’), (68, 'b’), (45, 'c’), (47, 'd’))
AS t(f1, f2);

f1 | f2
---+----
29 | a
68 | b
45 | c
47 | d
```

Оптимизация запросов: полезности

Получение строки из таблицы в виде ОДНОГО поля `ROW`.
Полезно для использования в подзапросах.

```
Table "public.t"
Column        | Type
-------+------------------
id     | integer
val    | double precision

SELECT t AS t_row
FROM t LIMIT 2;

t_row
-----------------------
(1, 0.1937)
(2, 0.4503)
```

Оптимизация запросов: полезности

Разворот `ROW` назад в набор колонок производится через запись вида `(ROW).*`
(внимание: скобки вокруг `ROW` тут обязательны).

```
SELECT (t_row).* FROM
(SELECT t AS t_row FROM t LIMIT 2) AS somealias;

id     | val
-------+--------
1      | 0.1937
2      | 0.4503
```

Оптимизация запросов: полезности

Свертка `N` значений в одно поле `array[]`.
Опять же, полезно для использования в подзапросах.

```
SELECT ARRAY(SELECT id FROM t LIMIT 5);

array
-------------
{1, 2, 3, 4, 5}
```

Оптимизация запросов: полезности

`UNNEST` Или обратный разворот `array[]` в строки.

```
SELECT UNNEST(ARRAY(SELECT t AS t_row FROM t LIMIT 5));

unnest
----------------------
(1, 0.1937)
(2, 0.4503)
(3, 0.9300)
(4, 0.7175)
(5, 0.1310)
```

Оптимизация запросов: полезности

Генерация серий (опять же полезно в `JOIN` и в сложных запросах).

```
SELECT * FROM generate_series(1,5) AS g(id);

id
----
1
2
3
4
5
```

Оптимизация запросов: полезности

- `UNNEST` обратная операция к `ARRAY`
- `(ROW).*` обратная операция к `SELECT t FROM t`

В итоге можно всю таблицу загнать в одну строку и развернуть назад:

```
SELECT (UNNEST(ARRAY(SELECT t AS t_row FROM t))).* ;
```

то же самое что

```
SELECT * FROM t;
```

Оптимизация запросов: пример 1

Задача:
Выбрать по некоему условию N строк из таблицы 1 и если требуемых N строк не нашлось – добрать недостающее из таблицы 2.

- Легко решается через 2 запроса кодом.
- Кодом получается в 2 раза больше сетевых задержек.

Попробуем применить вышеописанные техники:

```
WITH RECURSIVE
        t_res AS (SELECT ARRAY(SELECT test FROM test WHERE val < 3 ORDER
BY val) AS arr),
        t1_lim AS (SELECT GREATEST(0, 50 - (SELECT ARRAY_UPPER(arr, 1) FROM
t_res)) AS lim),
        t1_res AS (SELECT * FROM test1 WHERE val < 3 ORDER BY val LIMIT
(SELECT lim FROM t1_lim))

SELECT (UNNEST(arr)).* FROM t_res
UNION ALL
SELECT * FROM t1_res;
```

Оптимизация запросов: пример 2

- Известно что большие `OFFSET` работают медленно.
- Если надо ускорить работу больших `OFFSET` то в некоторых

случаях это возможно на `9.2+` (где поддерживается `index only scan`).

Проблемный запрос:

```
explain analyze select * from test order by val limit 10
offset 1000000;

 QUERY PLAN
-----------------------------------------------------------------------
Limit (cost=3752587.05..3752624.58 rows=10 width=144) (actual
time=3528.158..3528.192 rows=10 loops=1)
    -> Index Scan using test_val_key on test (cost=0.43..37525866.58 rows=
    10000000 width=144) (actual time=0.025..3255.450 rows=1000010 loops=1)
Total runtime: 3528.219 ms
```

Оптимизация запросов: пример 2

- У нас есть такая штука как `INDEX ONLY SCAN`
- Не будет ли он быстрее?

```
explain analyze select val from test order by val limit 10
offset 1000000;

 QUERY PLAN
-----------------------------------------------------------------
Limit (cost=25969.24..25969.49 rows=10 width=8) (actual
time=670.881..670.890 rows=10 loops=1)
    -> Index Only Scan using test_val_key on test (cost=0.43..259688.43
    rows= 10000000 width=8) (actual time=0.062..413.584 rows=1000010 loops=1)
Total runtime: 670.906 ms
```

Стало в 3 раза быстрее!

Оптимизация запросов: пример 2

- Но нам же нужны полные данные из `test` а не только `val`?!

```
explain analyze select * from test where val >= (select val from
test order by val limit 1 offset 1000000) order by val limit 10;

 QUERY PLAN
-----------------------------------------------------------------------
Limit (cost=25969.70..26007.25 rows=10 width=144) (actual time = 675.492
..675.542 rows=10 loops=1)
    -> Limit (cost=25969.24..25969.26 rows=1 width=8) (actual time = 675.459
..675.460 rows=1 loops=1)
        -> Index Only Scan using test_val_key on test test_1(cost=0.43..259688.43
rows=10000000 width=8) (actual time=0.042..417.548 rows=1000001 loops=1)
    -> Index Scan using test_val_key on test (cost=0.43..12518323.54 rows=
3333333 width=144) (actual time=675.491..675.539 rows=10 loops=1)
    Index Cond: (val >= $0)
Total runtime: 675.577 ms (было Total runtime: 3528.219 ms)
```

Оптимизация запросов: пример 3

- Зачастую возникает ситуация когда надо на некоторый достаточно быстрый запрос наложить дополнительный фильтр вида `IN` (простынка на 100 - 1000+ значений).
- Как правило это приводит к резкому замедлению запроса так как каждую строку ответа приходится проверять (линейным поиском) на вхождение в этот `IN` список.

Оптимизация запросов: пример 3

Пример:

1. `SELECT * FROM test WHERE id<10000`
   `1.2ms`
2. `SELECT * FROM test WHERE id<10000 AND val IN (список от 1 до 10)`
   `2.1ms`
3. `SELECT * FROM test WHERE id<10000 AND val IN (список от 1 до 100)`
   `6ms`
4. `SELECT * FROM test WHERE id<10000 AND val IN (список от 1 до 1000)`
   `38ms`
5. `SELECT * FROM test WHERE id<10000 AND val IN (список от 1 до 10000)`
   `380ms` (и далее линейно от длинны массива)

Оптимизация запросов: пример 3

План запроса для 100 `IN`:

```
explain analyze select * from test where id<10000 and val IN (1,...,100);

 QUERY PLAN
-------------------------------------------------------------------------
Index Scan using test_pkey on test (cost=0.43..1666.85 rows=10 width=140)
(actual time=0.448..5.602 rows=16 loops=1)
Index Cond: (id < 10000)
Filter: (val = ANY ('{1,...,100}'::integer[]))
Rows Removed by Filter: 9984
```

Оптимизация запросов: пример 3

Вопрос – а почему бы не использовать `hash` с линейным поиском по хешу `IN` списка. Не умеет пока PostgreSQL так. Зато он умеет `hash join`. Попробуем переделать запрос на `join` с `VALUES()`:

```
explain select count(*) from test JOIN (VALUES (1),...,(10)) AS v(val) USING
(val) where id<10000;

 QUERY PLAN
-------------------------------------------------------------------------
Aggregate (cost=497.65..497.66 rows=1 width=0)
     -> Hash Join (cost=0.69..497.65 rows=1 width=0)
     Hash Cond: (test.val = "*VALUES*".column1)
          -> Index Scan using test_pkey on test (cost=0.43..461.22 rows=9645
          width=4)
          Index Cond: (id < 10000)
     -> Hash (cost=0.12..0.12 rows=10 width=4)
          -> Values Scan on "*VALUES*" (cost=0.00..0.12 rows=10 width=4)
```

Оптимизация запросов: пример 3

Ура! `HASH JOIN`. А как с производительностью?

1. `SELECT * FROM test WHERE id<10000`
   `1.2ms`
2. `JOIN (VALUES (1),...,(10))`
   `1.6ms` (**было** **`2.1ms`**)
3. `JOIN (VALUES (1),...,(100))`
   `2ms` (**было** **`6ms`**)
4. `JOIN (VALUES (1),...,(1000))`
   `3.9ms` (**было** **`38ms`**)
5. `JOIN (VALUES (1),...,(10000))`
   `10ms` (**было** **`380ms`**)

Оптимизация запросов: пример 4

Иногда возникает задача подсчитать количество (или вывести всех) уникальных авторов в библиотеке или подобные ей которые обычно решаются через:

```
SELECT DISTINCT val FROM test;
```

И как только проект запускается и размер таблицы `test` вырастает – вышеприведенный запрос начинает тормозить (так как он требует перебора всех строк таблицы `test`).

Оптимизация запросов: пример 4

Плохой случай:

```
explain analyze select distinct val from test;

 QUERY PLAN
-------------------------------------------------------------------------
HashAggregate (cost=333333.67..333334.68 rows=101 width=4) (actual time=
6910.518..6910.547 rows=101 loops=1)
    -> Seq Scan on test (cost=0.00..308333.74 rows=9999974 width=4)
    (actual time=0.034..3513.216 rows=10000001 loops=1)
Total runtime: 6910.612 ms
```

6!!!! секунд чтобы вывести 100 уникальных `val` из таблицы.

Оптимизация запросов: пример 4

Что можно сделать? Добавить индекс по `val` на таблицу и применить технологию называемую `loose index scan`:

```
WITH RECURSIVE t AS (
 (SELECT min(val) AS val FROM test)
 UNION ALL
 SELECT (SELECT min(val) FROM test WHERE val > t.val)
AS val FROM t WHERE t.val IS NOT NULL
 )

SELECT val FROM t WHERE val IS NOT NULL;
```

Оптимизация запросов: пример 4

```
CTE Scan on t (cost=52.92..54.94 rows=100 width=4) (actual time=0.034..2.127 rows=101 loops=1)
 Filter: (val IS NOT NULL)
 Rows Removed by Filter: 1
 CTE t
   -> Recursive Union (cost=0.47..52.92 rows=101 width=4) (actual time=0.032..2.058 rows=102
loops=1)
     -> Result (cost=0.47..0.48 rows=1 width=0) (actual time=0.032..0.033 rows=1 loops=1)
       -> Limit (cost=0.43..0.47 rows=1 width=4) (actual time=0.029..0.029 rows=1 loops=1)
         -> Index Only Scan using test_val_key on test (cost=0.43..366924.94 rows=9999974 width=4)
(actual time=0.029..0.029 rows=1 loops=1)
         Index Cond: (val IS NOT NULL)
     -> WorkTable Scan on t t_1 (cost=0.00..5.04 rows=10 width=4) (actual time=0.019..0.019 rows=1
loops=102)
     Filter: (val IS NOT NULL)
     Rows Removed by Filter: 0
       -> Result (cost=0.47..0.48 rows=1 width=0) (actual time=0.017..0.018 rows=1 loops=101)
         -> Limit (cost=0.43..0.47 rows=1 width=4) (actual time=0.016..0.016 rows=1 loops=101)
           -> Index Only Scan using test_val_key on test test_1 (cost=0.43..130649.92 rows=3333325
width=4) (actual time=0.015..0.015 rows=1
loops=101)
           Index Cond: ((val IS NOT NULL) AND (val > t_1.val))
           Heap Fetches: 100
```

Total runtime: **2.178 ms** (**было 6900ms**).

Оптимизация запросов: конец

Что осталось за бортом:

- Куча всего
- Ещё немножко

Где почитать?

- http://www.google.com

Вопросы?

Задавайте.