Неклассические приемы оптимизации запросов

- Есть много малоизвестных **специфичных для PostgreSQL** приемов оптимизации запросов.

- В обычных презентациях и курсах посвященных оптимизации запросов они обычно не рассмотрены.

Введение: полезные приемы

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
```

Введение: полезные приемы

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
```

Введение: полезные приемы

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)
```

Введение: полезные приемы

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
```

Введение: полезные приемы

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}
```

Введение: полезные приемы

Тоже самое можно сделать с ROW типом:

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

{"(1,0.1937)","(2,0.4503)","(3,0.9300)","(4,0.7175)","(5,0.1310)
"}
```

Введение: полезные приемы

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)
```

Введение: полезные приемы

А теперь еще развернем ROW назад в строки:

```
SELECT (UNNEST(ARRAY(SELECT t AS t_row FROM t LIMIT 5))).*;
 id |   val
----+--------
  1 | 0.1937
  2 | 0.4503
  3 | 0.9300
  4 | 0.7175
  5 | 0.1310
```

Введение: полезные приемы

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
```

Введение: полезные приемы

- Замечания о обратимости операций.

- 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))).*

тоже самое что

SELECT * FROM t;

Выборка N строк из 2х таблиц с приоритетом первой таблицы.

- 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 или хранимкой.

Кодом получается в 2 раза больше сетевых задержек а хранимки использовать не хочется.

- Попробуем применить вышеописанные техники:

Выборка N строк из 2х таблиц с приоритетом одной из.

```
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 работы с большими OFFSET

- 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 работы с большими OFFSET

- А вот 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
```

Вот стало в 3 раза быстрее.

Оптимизация работы большими OFFSET

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)
   InitPlan 1 (returns $0)
->  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 работы с длинными IN

- 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 работы с длинными IN

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 работы с длинными IN

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 работы с длинными IN

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 работы с длинными IN

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)

Вынос JOIN за пределы LIMIT/OFFSET

Это даже скорее не оптимизация а исправление распространенной ошибки в написании запросов (или недоработки оптимизатора).

Рассмотрим ситуацию вида:

```
select …
from table1
join table2 using (table2id)
join table3 using (table3id)
where
набор условий ТОЛЬЕО по table1
Order by (набор полей table1) LIMIT SOMETHING OFFSET SOMETHING
```

при соблюдении важного условия – вы знаете (по бизнес логике или по факту наличия FK) то что JOIN’s с таблицами table2 и table3 на результат запрос не влияет (т.е. это просто подтягивание дополнительных данных к результату запроса по table1)

Вынос JOIN за пределы LIMIT/OFFSET

Вот пример плохого плана:

```
explain analyze select * from blogs join users on
blogs.owner_user_id=users.id where  sport_id=1 order by blogs.ctime desc
limit 10;
                    QUERY PLAN
-----------------------------------------------------------------------------------
 Limit  (cost=4039.83..4039.85 rows=10 width=1710) (actual time=116.676..116.680 rows=10
loops=1)
   ->  Sort  (cost=4039.83..4055.55 rows=6290 width=1710) (actual time=116.674..116.675
rows=10 loops=1)
         Sort Key: blogs.ctime
         Sort Method: top-N heapsort  Memory: 57kB
         ->  Nested Loop  (cost=0.00..3903.90 rows=6290 width=1710) (actual
time=0.079..91.362 rows=6043 loops=1)
               ->  Index Scan using blogs_sport_id_id_key on blogs  (cost=0.00..567.50
rows=6290 width=634) (actual time=0.051..14.149 rows=6043 loops=1)
                     Index Cond: (sport_id = 1)
               ->  Index Scan using users_pkey on users  (cost=0.00..0.48 rows=1 width=1076)
(actual time=0.009..0.010 rows=1 loops=6043)
                     Index Cond: (id = blogs.owner_user_id)
 Total runtime: 117.092 ms
```

Вынос JOIN за пределы LIMIT/OFFSET

А вот как можно сделать в 10 раз быстрее:

```
explain analyze select * from (select * from blogs where  sport_id=1 order by blogs.ctime desc limit 10) as blogs join sport_users on blogs.owner_user_id=sport_users.id order by blogs.ctime desc limit 10;
                                                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=703.43..709.76 rows=10 width=1710) (actual time=11.566..11.697 rows=10 loops=1)
   ->  Nested Loop  (cost=703.43..709.76 rows=10 width=1710) (actual time=11.565..11.692 rows=10 loops=1)
         ->  Limit  (cost=703.43..703.45 rows=10 width=634) (actual time=11.540..11.542 rows=10 loops=1)
               ->  Sort  (cost=703.43..719.15 rows=6290 width=634) (actual time=11.540..11.542 rows=10 loops=1)
                     Sort Key: public.blogs.ctime
                     Sort Method: top-N heapsort  Memory: 30kB
                     ->  Index Scan using blogs_sport_id_id_key on blogs  (cost=0.00..567.50 rows=6290 width=634) (actual time=0.034..7.927 rows=6043 loops=1)
                           Index Cond: (sport_id = 1)
         ->  Index Scan using sport_users_pkey on sport_users  (cost=0.00..0.53 rows=1 width=1076) (actual time=0.011..0.011 rows=1 loops=10)
               Index Cond: (id = public.blogs.owner_user_id)
 Total runtime: 11.926 ms (было  Total runtime: 117.092 ms)
```

Оптимизация distinct

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 distinct

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 distinct

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 distinct

```
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)
                InitPlan 1 (returns $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)
                              Heap Fetches: 1
          ->  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
                SubPlan 3
                  ->  Result  (cost=0.47..0.48 rows=1 width=0) (actual time=0.017..0.018 rows=1 loops=101)
                        InitPlan 2 (returns $3)
                          ->  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 top1 по списку id

Классическая задача: по заданному списку ID’s получить например по самому последнему события связанному с этими id’s.

Обычно это решается через конструкцию DISTINCT ON что опять же обычно приводит к ужасным планам если количество событий в каждом из заданных id большое.

Оптимизация top1 по списку id

Классический очень плохой случай (привет PgAdmin и его запросам при работе с PgAgent):

```
explain analyze select distinct ON (author) * FROM test WHERE author IN (29,68,45,47,50,41,11,4,83,60) ORDER BY author, rating desc;
```

```
                                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Unique  (cost=444136.34..449376.34 rows=11 width=148) (actual time=2760.087..3676.198 rows=10 loops=1)
   ->  Sort  (cost=444136.34..446756.34 rows=1048001 width=148) (actual time=2760.085..3395.677 rows=1000004 loops=1)
         Sort Key: author, rating
         Sort Method: external merge  Disk: 154344kB
         ->  Bitmap Heap Scan on test  (cost=21354.36..262326.38 rows=1048001 width=148) (actual time=236.000..1162.755 rows=1000004 loops=1)
               Recheck Cond: (author = ANY ('{29,68,45,47,50,41,11,4,83,60}'::integer[]))
               ->  Bitmap Index Scan on test_val_key  (cost=0.00..21092.36 rows=1048001 width=0) (actual time=177.704..177.704 rows=1000004 loops=1)
                     Index Cond: (author = ANY ('{29,68,45,47,50,41,11,4,83,60}'::integer[]))
 Total runtime: 3701.711 ms
```

Оптимизация top1 по списку id

А вот как это надо делать по хорошему в версиях 8.4+:

```
EXPLAIN ANALYZE SELECT (_row).* FROM (
  SELECT (
    SELECT test FROM test WHERE test.author=t.author ORDER BY rating DESC LIMIT 1
  ) AS _row
  FROM (
    VALUES (29),(68),(45),(47),(50),(41),(11),(4),(83),(60)
  ) AS t(author) OFFSET 0
) as _t;
                                                                          QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------
 Subquery Scan on _t  (cost=0.00..42.46 rows=10 width=32) (actual time=0.022..0.109 rows=10 loops=1)
   ->  Values Scan on "*VALUES*"  (cost=0.00..42.36 rows=10 width=4) (actual time=0.021..0.100 rows=10 loops=1)
         SubPlan 1
           ->  Limit  (cost=0.43..4.22 rows=1 width=180) (actual time=0.008..0.009 rows=1 loops=10)
                 ->  Index Scan using test_val_key on test  (cost=0.43..375069.31 rows=99010 width=180) (actual
time=0.008..0.008 rows=1 loops=10)
                       Index Cond: (author = "*VALUES*".column1)
 Total runtime: 0.131 ms (было 3701.711 ms)
```

Оптимизация top1 по списку

На версиях 9.3 и выше с введением LATERAL этот запрос становится проще и короче (и кстати работает раза в 2 быстрее... все эти array/unnest они не бесплатные):

```
SELECT test.*
FROM (
  VALUES (29),(68),(45),(47),(50),(41),(11),(4),(83),(60)
) AS t(author),
LATERAL (
  SELECT * FROM test WHERE test.author=t.author ORDER BY rating
DESC LIMIT 1
) AS _t;
```

Оптимизация distinct ON

Продолжение предыдущей задачи, теперь надо выбрать самые свежие события не по списку ID а для всех объектов в таблице событий.

Обычно это решается через опять же DISTINCT ON но практически это работает очень медленно.

Оптимизация distinct ON

Опять же классический плохой случай:

```
explain (analyze, costs, buffers, timing) select distinct ON (author) * from test
ORDER BY author, rating desc;
                                QUERY PLAN
-------------------------------------------------------------------------------------
 Unique  (cost=2214935.03..2264935.08 rows=101 width=148) (actual
time=18661.433..30088.668 rows=101 loops=1)
   Buffers: shared hit=2240 read=215152, temp read=193029 written=193029
   ->  Sort  (cost=2214935.03..2239935.06 rows=10000009 width=148) (actual
time=18661.430..27188.731 rows=10000001 loops=1)
         Sort Key: author, rating
         Sort Method: external merge  Disk: 1543504kB
         Buffers: shared hit=2240 read=215152, temp read=193029 written=193029
         ->  Seq Scan on test  (cost=0.00..317392.09 rows=10000009 width=148)
(actual time=0.016..3392.857 rows=10000001 loops=1)
               Buffers: shared hit=2240 read=215152
 Total runtime: 30295.630 ms
```

Оптимизация distinct ON

А что можно сделать?

Скрестить ужа с слайда 27 (loose index scan) для получения всех уникальных значений в таблице и ежа с слайда 31-32, на выходе получив запрос который будет работать сильно быстрее.

Оптимизация distinct ON

Вот что получается в версии для 8.4 и выше:

```
WITH RECURSIVE t AS (
   (SELECT min(author) AS author FROM test)
   UNION ALL
   SELECT (SELECT min(author) FROM test WHERE author > t.author) AS author FROM t WHERE
t.author IS NOT NULL
)
SELECT (_row).* FROM (
  SELECT (
    SELECT test FROM test WHERE test.author=t.author ORDER BY rating DESC LIMIT 1
  ) AS  _row
  FROM t WHERE author IS NOT NULL
  OFFSET 0
) as _t;
```

Время работы 2.7ms вместо 30000ms.

Оптимизация distinct ON

В версии для 9.3 и выше можно воспользоваться новым функционалом LATERAL и использовать сильно более простого ужа:

```
WITH RECURSIVE t AS (
   (SELECT min(author) AS author FROM test)
   UNION ALL
   SELECT (SELECT min(author) FROM test WHERE author > t.author) AS author FROM
t WHERE t.author IS NOT NULL
 )
SELECT test.* FROM t, LATERAL (
  SELECT * FROM test WHERE test.author=t.author ORDER BY rating DESC LIMIT 1
) AS test
WHERE t.author IS NOT NULL;
```

Время работы 2.3ms вместо 30000ms.

Оптимизация topN по списку

А что если нам надо вывести не по 1 самому свежему событию по списку ID а по N событий (N>1).

Через DISTINCT ON задача просто не решается.

А вот с использованием вышеприведенных техник решается достаточно просто.

Оптимизация topN по списку

Версия для 8.4 и выше:

```
WITH t(author) AS (
  VALUES (29),(68),(45),(47),(50),(41),(11),(4),(83),(60)
)
SELECT (_row).* FROM (
  SELECT unnest(array(SELECT test FROM test WHERE
test.author=t.author ORDER BY rating DESC LIMIT 5)) AS _row
  FROM t WHERE author IS NOT NULL
  OFFSET 0
) AS _t;
```

Используются приемы с слайдов 2,9,31.

Оптимизация topN по списку

На версиях 9.3 и выше с введением LATERAL этот запрос становится проще и короче (и кстати работает раза в 2 быстрее... все эти array/unnest они не бесплатные):

```
SELECT test.*
FROM (
  VALUES (29),(68),(45),(47),(50),(41),(11),(4),(83),(60)
) AS t(author),
LATERAL (
  SELECT * FROM test WHERE test.author=t.author ORDER BY rating
DESC LIMIT 5
) AS _t;
```

Оптимизация topN по всей таблице

А теперь финальная задача. Надо вывести topN по всей таблице а не по заданному списку ID.

Все тоже самое только еж другой (вместо VALUES уже несколько раз упомянутый Loose Index Scan).

Оптимизация topN по всей таблице

Версия для 8.4+:

```
WITH RECURSIVE t AS (
   (SELECT min(author) AS author FROM test)
   UNION ALL
   SELECT (SELECT min(author) FROM test WHERE author > t.author)
AS author FROM t WHERE t.author IS NOT NULL
)
SELECT (_row).* FROM (
  SELECT unnest(array(SELECT test FROM test WHERE
test.author=t.author ORDER BY rating DESC LIMIT 5)) AS _row
  FROM t WHERE author IS NOT NULL
  OFFSET 0
) AS _t;
```

Оптимизация topN по всей таблице

Версия для 9.3+ (работает раза в 2 быстрее):

```
WITH RECURSIVE t AS (
   (SELECT min(author) AS author FROM test)
   UNION ALL
   SELECT (SELECT min(author) FROM test WHERE author > t.author)
AS author FROM t WHERE t.author IS NOT NULL
)
SELECT test.* FROM t, LATERAL (
   SELECT * FROM test WHERE test.author=t.author ORDER BY rating
DESC LIMIT 5
) AS test
WHERE t.author IS NOT NULL;
```

Оптимизация topN по списку ID

А бывает альтернативная задача – по списку ID (авторы/теги) эффективно выбрать N самых свежих событий (с поддержкой OFFSET).

Классическая лента постов в соц сетях например.

Можно, но я просто приведу финальный запрос без разбора (так как его разбор это еще на час). Эта штука работает в production и более быструю версию пока еще никто не придумал.

Исходный запрос звучал как (но работал недопустимо медленно):

SELECT * FROM comments WHERE tag_id IN (LIST) ORDER BY ctime DESC OFFSET N LIMIT M;

PS: разбор запроса есть у меня в ЖЖ по адресу

http://astarsan.livejournal.com/6895.html

Оптимизация topN по списку

```
WITH RECURSIVE
tags AS (SELECT '{1900,1752,4391,...tag list...}'::bigint[] AS _tags),  --tag list
gs AS (SELECT _pos FROM generate_subscripts((SELECT _tags FROM tags),1) gs(_pos)), --pregenerated iterator array
r AS(
    SELECT
        NULL::comments as _result,
        0::integer as _rows_found,
        ARRAY(SELECT (SELECT ctime FROM comments WHERE tag_id=_tag ORDER BY ctime DESC LIMIT 1) FROM unnest(_tags) u(_tag)) as _tags_ts
--INITIAL PER tag ARRAY of latest commentss
        FROM tags
   UNION ALL
    SELECT
        CASE WHEN _rows_found>=1980 THEN (SELECT comments FROM comments WHERE tag_id=(SELECT _tags[_pos] FROM tags) AND ctime=_tags_ts[_pos])
--return row to the result set if we already go through _offset or more entries
        ELSE NULL END,
        _rows_found+1, --increase found row count
        _tags_ts[1:_pos-1]||(
            SELECT ctime FROM comments
           WHERE tag_id=(SELECT _tags[_pos] FROM tags) AND ctime<_tags_ts[_pos]
            ORDER BY ctime DESC LIMIT 1
        )||_tags_ts[_pos+1:array_length(_tags_ts,1)] --replace current entry of latest message by tag with previous by date for the same tag
        FROM (
            SELECT *,
            (SELECT _pos FROM gs ORDER BY _tags_ts[_pos] DESC NULLS LAST LIMIT 1) AS _pos --find position of the latest entry in the tag list
            FROM r OFFSET 0
        ) as t2
        WHERE _rows_found<20+1980 --we had found the required amount of rows (offset+limit done)
)
SELECT (_result).* FROM r WHERE NOT _result IS NULL ORDER BY _rows_found;
```

Заключение

Для чего это все можно еще применять.

Например для создания крайне легких и быстрых сайтов через связку:

client-side (JS/AJAX) → JSON запрос → маппинг JSON запроса на хранимку или запрос в базе → база (sql или pl/pgsql) → JSON ответ от базы → client-side (JSON/AJAX)-> отображение.

В таком варианте программирование как таковое присутствует только на клиенте и минимально в базе.

Заключение

При этом уходит фактически все server-side программирование (php/perl/java/etc).

При желании http json<->postgresql общение можно сделать через простой модуль в nginx (libpq поддерживает асинхронный неблокирующий режим работы с базой).

Производительность таких решений легко может на порядок превышать производительность классического веб сайта (при одинаковом железе).

Заключение

Чем сейчас занимается код веб-сайта 90% времени:

1)генерацией запросов в базу (ORM)

2)превращением ответов базы в объекты (опять ORM)

3)преобразованием объектов в HTML

Вопрос: зачем городить такие сложности если можно получить от базы готовый JSON и отдать его клиенту как есть для отрисовки?

Вопросы?