Оптимизация запросов

Иван Фролков
i.frolkov@postgrespro.ru
Postres Professional

Что мы будем рассматривать

- Чтобы что-то улучшить, это надо измерить.
- Чтобы что-то измерить, надо знать, что измерять
  - Методы доступа
  - Фильтрация
  - Способы соединения
- Измерили – что-то получили. А как оно вообще должно выглядеть? Насколько то, что есть, соответствует представлениям о том, что должно быть и почему так? Почему оптимизатор выбрал именно такие методы доступа и соединения? Что еще можно улучшить?
- Общие соображения

Выполнение запроса

- Parser – синтаксический разбор
- Transformation process – изменение запроса в соответствии с системным каталогом и начало транзакции, если транзакция еще не начата
- Planner/Optimizer – поиск и построение оптимального плана
- Executor – выполнение плана

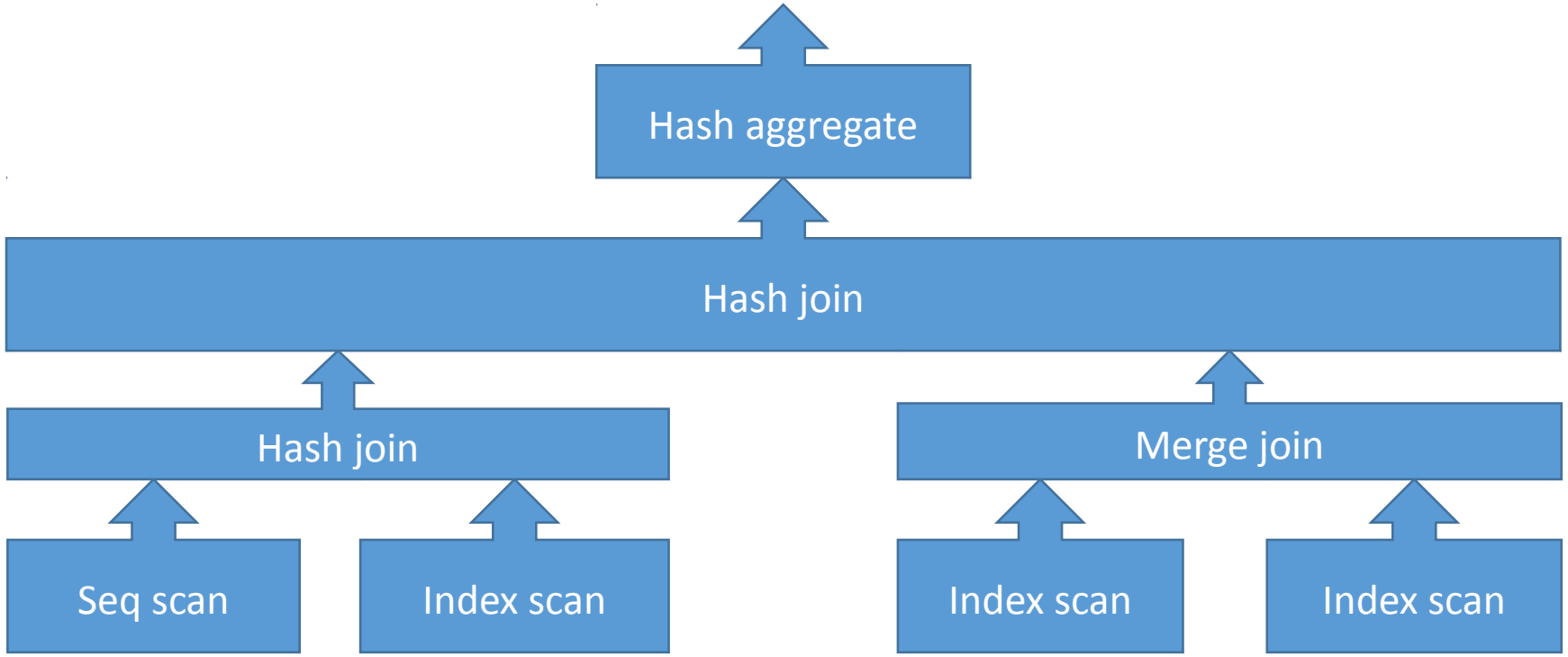
EXPLAIN

- EXPLAIN – показывает план выполнения.
- План – этот дерево, где строки поднимаются от листьев к корню.
- Субпланы. (InitPlan – независимый подзапрос, SubPlan – зависимый подзапрос с параметрами)
- ```
  explain
    select * from tbl
  explain analyze
    select  * from tbl
  explain(analyze, buffers,verbose)
    select * from tbl
  ```
  ```

Postgres PROFESSIONAL

EXPLAIN

Query plan



EXPLAIN - продолжение

```
• Sort  (cost=47.99..48.01 rows=10 width=73)
    Sort Key: m.name
    InitPlan 2 (returns $1)
      ->  Sort  (cost=5.57..5.84 rows=105 width=9)
            Sort Key: m_2.name
            ->  Seq Scan on producer_model m_2  (cost=0.00..2.05 rows=105 width=9)
  ->  Nested Loop  (cost=0.00..41.98 rows=10 width=73)
        Join Filter: (m.producer_id = p.id)
        ->  Seq Scan on producer p  (cost=0.00..1.01 rows=1 width=36)
        ->  Seq Scan on producer_model m  (cost=0.00..3.36 rows=10 width=45)
              Filter: (id = ANY ($1))
        SubPlan 1
          ->  Limit  (cost=0.00..3.75 rows=1 width=99)
                ->  Nested Loop  (cost=0.00..29.99 rows=8 width=99)
                      ->  Seq Scan on producer_model_color pc  (cost=0.00..15.09 rows=8 width=4)
                            Filter: (producer_model_id = m.id)
                      ->  Materialize  (cost=0.00..6.66 rows=1 width=103)
                            ->  Nested Loop  (cost=0.00..6.66 rows=1 width=103)
                                  ->  Nested Loop  (cost=0.00..3.34 rows=1 width=4)
                                        Join Filter: (p_1.id = m_1.producer_id)
                                        ->  Seq Scan on producer p_1  (cost=0.00..1.01 rows=1 width=4)
                                        ->  Seq Scan on producer_model m_1  (cost=0.00..2.31 rows=1 width=8)
                                              Filter: (id = m.id)
                                  ->  Seq Scan on producer_model_variant v  (cost=0.00..3.31 rows=1 width=103)
                                        Filter: (producer_model_id = m.id)
```

Методы доступа

- Seq scan
- Index scan
- Index scan backward
- Full index scan
- Index Only Scan

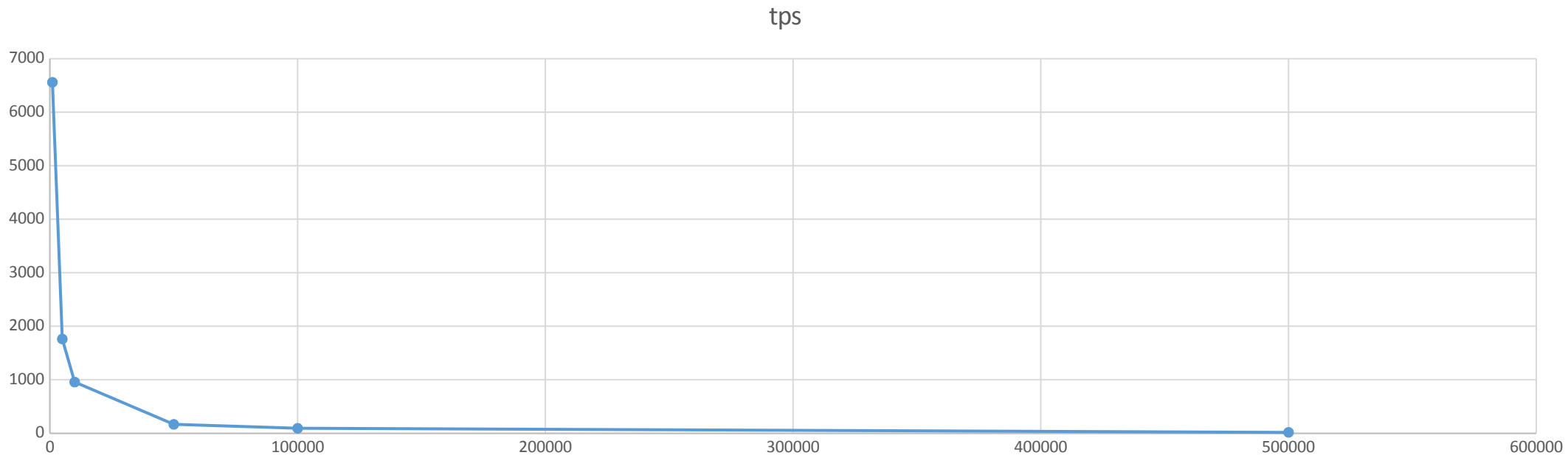
Последовательное сканирование/Seq scan

- Перебирается вся таблица
- В зависимости от замусоренности таблиц даже таблица с небольшим количеством строк может сканироваться заметное время.
  - Может ли FSM использоваться для оптимизации Seq scan? Увы, это дерево, не очень подходящее для таких целей.
  - ```
    create table filler(id int);
    insert into filler select n from generate_series(1,100000) as gs(n);
    explain(analyze, buffers); select * from filler –Buffers: shared hit=443
    vacuum filler;
    Buffers: shared hit=443
    ```
- Для совсем небольших таблиц – оптимальный метод доступа.
    ```

Давайте попробуем

- ```
  create table t1(
   id serial primary key,
   val text)
  ```
- ```
  select * from t1 where id+1=100
  ```

Строк	1000	5000	10000	50000	100000	500000
tps	6557	1761	956	166	93	17



Выводы

- Оптимален для небольших таблиц
- Идеальный вариант в том случае, если нам требуются все данные (или любая строка)

Index scans

- Доступ по индексу: сначала проходится корень, потом до листа. Обычная глубина дерева – 3, у маленьких таблиц – 1 или 2, у очень больших – 4.
- Используется:
  - Быстрый поиск по ключу
  - Выдача отсортированного результата в том случае, когда необходимо быстро получить первую строку/строки.
  - Получение данных, которые присутствуют в индексе без обращения к таблице

Давайте попробуем

Выбираем ВСЕ колонки!!

- `select * from t1 where id=100`

Строк	1000	5000	10000	50000	100000	500000
tps	17449	16514	16717	17111	17134	17721

Index-only scan

- Когда можно получить данные прямо из индекса, без обращения к таблице
- Для прокешированных данных результаты практически не отличаются от index scan

Строк	1000	5000	10000	50000	100000	500000
tps	17562	17391	17262	16853	17124	17631

- Таким образом, эффективно для больших таблиц
- VACUUM!!!

Index & Index-only scans

- `insert into t1(val) select repeat('X',100) from generate_series(1,1000)`
- `vacuum t1; -- NB`
- ```
  explain(analyze, buffers)
  select id from t1 where id=100
  ```
- ```
  Index Only Scan using t1_pkey on t1  (cost=0.28..8.29 rows=1 width=4)
  (actual time=0.021..0.021 rows=0 loops=1)
    Index Cond: (id = 100)
    Heap Fetches: 0
    Buffers: shared hit=2
  Total runtime: 0.062 ms
  ```

А почему 2? 1000 значений integer – это же чуть меньше 4К? А страница 8К? Должно же хватать одной? Или нет? На самом деле нет – 4 байта на значение + 14 байт накладных расходов – 15-16К, т.е. как минимум три страницы. А почему три, умещается же в две?... Fillfactor для индекса? Нет…

Index & Index-only scans - продолжение

- \setrandom start 100 400000
  select id, substring('XXX' from 1 for 1) from t1 where id between :start and :start+5000
  - 424 rqs
- \setrandom start 100 400000
  select id, substring(val from 1 for 1) from t1 where id between :start and :start+5000
  - 239 rqs
- Почему? (:start = 100)
- Index Only Scan using t1_pkey on t1 (cost=0.42..151.48 rows=4753 width=4) (actual time=0.024..1.205 rows=5001 loops=1)
  Index Cond: ((id >= 100) AND (id <= 5100))
  Heap Fetches: 0
  Buffers: shared hit=17
  Total runtime: 1.449 ms
- Index Scan using t1_pkey on t1 (cost=0.42..248.37 rows=4753 width=105) (actual time=0.026..3.971 rows=5001 loops=1)
  Index Cond: ((id >= 100) AND (id <= 5100))
  Buffers: shared hit=103
  Total runtime: 4.229 ms

Index-only scan. Осторожно!

- Возможность определяется через Visibility Map (VM, не путать с FSM). (Имя файла - <filenode>_vm)
- Целиком для страницы
- Очищается только VACUUM (У только что созданной таблицы он пуст; VACUUM FULL также сбрасывает).
- Число реальных обращений к таблице можно увидеть в выдаче EXPLAIN как Heap Fetches

TID scan

- TID – Tuple ID
- Изрядное хакерство, но…
- CTID – внутренний идентификатор записи (страница, номер строки в странице)
- TID Scan
- Быстро? Быстро:
- ```
  explain(analyze, buffers)
  select * from t1 where ctid='(100,20)'::tid
  ```
- ```
  Tid Scan on t1  (cost=0.00..4.01 rows=1 width=105) (actual time=0.012..0.012 rows=1 loops=1)“
    TID Cond: (ctid = '(100,20)'::tid)“
    Buffers: shared hit=1“
  Total runtime: 0.030 ms"
  ```
- ```
  explain(analyze, buffers)
  select * from t1 where id=5820
  ```
- ```
  Index Scan using t1_pkey on t1  (cost=0.42..8.44 rows=1 width=105) (actual time=0.026..0.028 rows=1
  loops=1)
    Index Cond: (id = 5820)
    Buffers: shared hit=4
  Total runtime: 0.055 ms
  ```
- Вообще-то это внутренние дела и соваться туда не надо – никто не обещал, что это вообще будет работать
- Зачем это надо? Чтобы не было вопросов, если придется увидеть TID scan в плане

Postgres PROFESSIONAL

Выводы

- Table scan - если с умом – вполне уместная вещь
- Во многих случаях весьма важен
- Index scan обеспечивает практически константное время доступа
- Index-only scan – еще более быстро
  - Можно перегнуть палку, но
  - Пресловутые column-oriented databases
  - VACUUM!

Фильтрация таблицы

- Сканирование
- Range query
- Bitmap index scan

Сканирование таблицы

- Не все так однозначно – нередко seq scan лучше
- `insert into t1(val) select repeat('X',100) from generate_series(1,10000)`

Сканирование таблицы - продолжение

```
 select id, substring(val from 1 for 1) from t1 where id between 100 and 100+8000
Seq Scan on t1  (cost=0.00..343.00 rows=8000 width=105) (actual time=0.028..6.707
rows=8001 loops=1)
  Filter: ((id >= 100) AND (id <= 8100))
  Rows Removed by Filter: 1999
  Buffers: shared hit=173
Total runtime: 7.090 ms
```

```
 set enable_seqscan=off;
 Bitmap Heap Scan on t1  (cost=178.29..491.28 rows=8000 width=105) (actual
 time=0.967..6.150 rows=8001 loops=1)
   Recheck Cond: ((id >= 100) AND (id <= 8100))
   Buffers: shared hit=163
   ->  Bitmap Index Scan on t1_pkey  (cost=0.00..176.29 rows=8000 width=0)
 (actual time=0.923..0.923 rows=8001 loops=1)
         Index Cond: ((id >= 100) AND (id <= 8100))
         Buffers: shared hit=24
 Total runtime: 6.551 ms
```

Индекс - range scan

- Поиск в диапазоне (column > 10 and column <200, column between *this* and *that, column like 'prefix%')*
  - Кстати – если индекс построен по колонке, то берем колонку. Если индекс построен по функции от колонки – берем эту функцию от колонки.
    - Index on table(col)
      - `where table.col=<something>` - хорошо!
      - `where func(table.col)=<something>` - плохо!
    - Index on table(func(col))
      - `where func(table.col)=<something>` - хорошо!
      - `where other_func(table.col)=<something>` - плохо!

- В планах можно видеть Bitmap Heap Scan – отличие от Index Scan в том, что если ожидается достаточно большое число строк, то доступ к ним сортируется таким образом, чтобы попытаться прочитать наибольшую часть страниц последовательно; просто Index Scan этим не озабочивается. (Обратите внимание, что Bitmap Scan сопровождается Recheck Condition. Почему?)

Индекс - point query

- Доступ по первичному или уникальному ключу
- В Postgres на самом деле это тоже range index scan
- Почему? Версии строк!
- Как обеспечивается уникальность?
  - Если нет такого ключа, то все хорошо
  - Если ключ есть, но удален завершенной транзакцией, то все хорошо
  - Если ключ есть, но удален собственной транзакцией, то все хорошо
  - Если ключ есть, но удален незавершенной транзакцией, то ждем того, что с ней случится, и либо все хорошо (транзакция откатилась), либо возбуждаем ошибку (другая транзакция завершилась успешно)
  - Если ключ есть, но добавлен незавершенной транзакцией, то ждем ее окончания и действуем по мотивам предыдущего пункта
  - Если ключ есть, но добавлен откаченной транзакцией, то все хорошо
  - VACUUM!!!

Кстати - COLLATION

- По умолчанию строки сравниваются в collation базы. Это не всегда нужно, но всегда медленнее:
- ```
  create table colltest(
   uuid text,
   uuid_nocoll text collate "C“
  )
  ```
- ```
  insert into colltest
  select uuid_generate_v4(), uuid_generate_v4() from generate_series(1,100000)
  ```
- Строим индекс:
- `create index colltest_uuid on colltest(uuid)` – 263 мс
- `create index colltest_uuid_nocoll on colltest(uuid_nocoll)` – 125 мс

COLLATION - продолжение

- ```
  explain(analyze, buffers)
  select * from colltest ct1, colltest ct2
  where ct1.uuid_nocoll between ct2.uuid_nocoll and ct2.uuid_nocoll || 'A‘
  ```
- ```
  Nested Loop  (cost=0.42..33385311.00 rows=1111111111 width=128) (actual time=0.070..310.218
  rows=100000 loops=1)
    Buffers: shared hit=401988 read=723
    ->  Seq Scan on colltest ct2  (cost=0.00..2283.00 rows=100000 width=64) (actual
  time=0.008..8.848 rows=100000 loops=1)
          Buffers: shared hit=1283
    ->  Index Scan using colltest_uuid_nocoll on colltest ct1  (cost=0.42..222.72 rows=11111
  width=64) (actual time=0.002..0.003 rows=1 loops=100000)
          Index Cond: ((uuid_nocoll >= ct2.uuid_nocoll) AND (uuid_nocoll <= (ct2.uuid_nocoll
  || 'A'::text)))
          Buffers: shared hit=400705 read=723
  Planning time: 0.433 ms
  Execution time: 316.013 ms
  ```

COLLATION - продолжение

- ```
  explain(analyze, buffers)
  select * from colltest ct1, colltest ct2
  where ct1.uuid between ct2.uuid and ct2.uuid || 'A'
  ```
- ```
  Nested Loop  (cost=0.42..33385311.00 rows=1111111111 width=148) (actual time=0.087..560.101
  rows=100000 loops=1)
    Buffers: shared hit=402711
    ->  Seq Scan on colltest ct2  (cost=0.00..2283.00 rows=100000 width=74) (actual
  time=0.012..8.673 rows=100000 loops=1)
          Buffers: shared hit=1283
    ->  Index Scan using colltest_uuid on colltest ct1  (cost=0.42..222.72 rows=11111
  width=74) (actual time=0.005..0.005 rows=1 loops=100000)
          Index Cond: ((uuid >= ct2.uuid) AND (uuid <= (ct2.uuid || 'A'::text)))
          Buffers: shared hit=401428
  Planning time: 0.229 ms
  Execution time: 565.587 ms
  ```

Что делать, если все-таки надо utf8 и like?

- CREATE INDEX name ON table (column opclass [sort options] [, ...]);
- text_pattern_ops, varchar_pattern_ops и bpchar_pattern_ops – text, varchar и char соответственно

Bitmap index scan

- Структура в памяти со ссылками на строку или страницу (exact/lossy)
- Зависит от work_mem

Bitmap index scan

- ```
  CREATE TABLE aa AS SELECT * FROM generate_series(1, 10000000) AS a ORDER BY random();
  CREATE INDEX aai ON aa(a);
  SET enable_indexscan=false;
  SET enable_seqscan=false;
  ```
- ```
  SET work_mem = '64kB';
  EXPLAIN ANALYZE SELECT * FROM aa WHERE a BETWEEN 100000 AND 200000;
  ```
- *http://michael.otacoo.com/postgresql-2/postgres-9-4-feature-highlight-lossyexact-pages-for-bitmap-heap-scan/*

Bitmap index scan

```
Bitmap Heap Scan on aa  (cost=952.93..43010.35 rows=50000
width=4) (actual time=33.593..1234.128 rows=100001 loops=1)
  Recheck Cond: ((a >= 100000) AND (a <= 200000))
  Rows Removed by Index Recheck: 9002327
  Heap Blocks: exact=546 lossy=35691
  ->  Bitmap Index Scan on aai  (cost=0.00..940.43 rows=50000
width=0) (actual time=33.355..33.355 rows=100001 loops=1)
        Index Cond: ((a >= 100000) AND (a <= 200000))
Planning time: 0.296 ms
Execution time: 1237.926 ms
```

Bitmap index scan

```
SET work_mem = '4MB';
Bitmap Heap Scan on aa  (cost=1907.19..42625.75 rows=100171
width=4) (actual time=19.695..132.142 rows=100001 loops=1)'
  Recheck Cond: ((a >= 100000) AND (a <= 200000))'
  Heap Blocks: exact=36237'
  ->  Bitmap Index Scan on aai  (cost=0.00..1882.14 rows=100171
width=0) (actual time=13.133..13.133 rows=100001 loops=1)'
        Index Cond: ((a >= 100000) AND (a <= 200000))'
Planning time: 0.102 ms'
Execution time: 135.866 ms'
```

Bitmap index scan

```
SET enable_indexscan=on; SET enable_seqscan=on;
vacuum analyze aa;
Index Only Scan using aai on aa  (cost=0.43..2894.82
rows=100519 width=4) (actual time=0.033..11.094 rows=100001
loops=1)
  Index Cond: ((a >= 100000) AND (a <= 200000))
  Heap Fetches: 0
Planning time: 0.156 ms
Execution time: 14.198 ms
```

Bitmap index scan

- Комбинация нескольких индексов.
- `create index t1_idfn on t1( (id%17) )`
-
```
select id, substring(val from 1 for 1) from t1
 where id between 100 and 2000
 and id%17=4
```
-
```
Bitmap Heap Scan on t1  (cost=48.20..78.25 rows=9 width=105) (actual time=0.341..0.455 rows=112 loops=1)
  Recheck Cond: (((id % 17) = 4) AND (id >= 100) AND (id <= 2000))
  Buffers: shared hit=45
  ->  BitmapAnd  (cost=48.20..48.20 rows=9 width=0) (actual time=0.326..0.326 rows=0 loops=1)
        Buffers: shared hit=11
        ->  Bitmap Index Scan on t1_idfn  (cost=0.00..4.66 rows=50 width=0) (actual time=0.083..0.083
rows=589 loops=1)
              Index Cond: ((id % 17) = 4)
              Buffers: shared hit=4
        ->  Bitmap Index Scan on t1_pkey  (cost=0.00..43.28 rows=1900 width=0) (actual time=0.218..0.218
rows=1901 loops=1)
              Index Cond: ((id >= 100) AND (id <= 2000))
              Buffers: shared hit=7
Total runtime: 0.497 ms
```

Bitmap index scan/комбинация индексов

- Когда разумно использовать – комбинация нескольких невысокоселективных индексов, например, пересечение среди сотрудников организации женщин и экскаваторщиков
- Требует память и некоторое время для начала работы (в плане указано время до выдачи первой строки `cost=48.20`)

Multicolumn indexes

- Состоят из нескольких колонок
- Могут использоваться для поиска, начиная с лидирующих колонок
- Могут использоваться в ряде случаев для доступа не к лидирующим колонкам
- ```
  create table t3(
   id serial primary key, id2 int, id3 int, id4 bigint, id5 bigint,
  val text);
  insert into t3(id2, id3, id4, id5, val)
   select n/100, n%100, n, random()*n, repeat('X',500)
  from generate_series(1,100000) as gs(n);
  create index t3_suppl on t3(id2, id3);
  ```
- vacuum t3; -- а почему сразу vacuum?
  ```

Multicolumn indexes - продолжение

- explain(analyze, verbose, buffers)
  select id3 from t3 where id3 in (10,93,1,2)
- Index Only Scan using t3_suppl on public.t3 (cost=0.29..4144.14 rows=3897 width=4) (actual time=0.056..18.738 rows=4000 loops=1)
  Output: id3
  Index Cond: (t3.id3 = ANY ('{10,93,1,2}'::integer[]))
  Heap Fetches: 0
  Buffers: shared hit=858 read=243
  Total runtime: 18.990 ms

Multicolumn indexes - продолжение

Колонки в индексе как фильтр

```
create table t2(id serial primary key, id2 int, id3 int, val text);

insert into t2(id2, id3, val)
 select n/100, n%100, n*random() from generate_series(1,100000) as gs(n);

create index t2_id2_val on t2(id2,val);

vacuum t2;

explain(analyze, buffers)

select id2, val from t2 where id2 between 30 and 200 and val like '%88%'
```

Multicolumn indexes - продолжение

```
Index Only Scan using t2_id2_val on t2  (cost=0.42..729.55 rows=1765
width=20) (actual time=0.020..5.200 rows=1955 loops=1)
  Index Cond: ((id2 >= 30) AND (id2 <= 200))
  Filter: (val ~~ '%88%'::text)
  Rows Removed by Filter: 15145
  Heap Fetches: 0
  Buffers: shared hit=89
Total runtime: 5.314 ms
```

- Вывод: колонки в индексе могут использоваться как фильтр.
- VACUUM!

Multicolumn indexes - продолжение

- Доступ по ключу + сортировка
- `explain(analyze, buffers)`
  `select id3 from t2 where id2=2 order by id3`
-
```
Index Only Scan using t2_suppl on t2  (cost=0.29..6.04 rows=100 width=4)
(actual time=0.023..0.039 rows=100 loops=1)
  Index Cond: (id2 = 2)
  Heap Fetches: 0
  Buffers: shared hit=3
Total runtime: 0.068 ms
```
- Использование:
  `colinindex1=:val1 and colinindex2=:val2 order by colinindex3, colinindex4…`

Параметры сервера, связанные с методами доступа

- enable_seqscan
- enable_indexscan
- enable_indexonlyscan
- enable_bitmapscan
- enable_tidscan

Параметры сервера - продолжение

- seq_page_cost (floating point) – стоимость получения страницы при последовательном чтении. По умолчанию 1.0
- random_page_cost – получение случайной страницы. По умолчанию 4.0
  - *Оба параметра выше могут быть изменены для tablespace (SSD) или когда данные целиком умещаются в памяти.*
  - *Оба параметра могут быть установлены для tablespace; таким образом, можно для активной части базы в памяти иметь random_page_cost близким к единице и бОльшим для архивных данных.*
- cpu_tuple_cost – стоимость получения страницы из памяти. По умолчанию 0.01
- cpu_index_tuple_cost – то же самое, только для индекса. 0.005 по умолчанию
- cpu_operator_cost – стоимость обработки оператора или функции. По умолчанию 0.0025

Параметры сервера - статистика

- track_counts – включена ли статистика
- track_functions – отслеживать ли вызов функций, и если да, то каких
- stats_temp_directory – где сборщику статистики хранить промежуточные данные

Статистика

- Для каждой таблицы и для каждой колонки собирается статистика. Команда ANALYZE
- View pg_stats. Что там? Некоторые колонки

null_frac	Часть не-null колонок
avg_width	Средняя ширина в байтах
n_distinct	>0 – примерное число уникальных значений; ожидается, что таблица расти не будет <0 – число уникальных значений/число строк * -1; ожидается, что таблица будет расти
most_common_vals	Список наиболее частых значений
most_common_freqs	Список частот таких значений (число значений/число строк)
histogram_bounds	Список значений, разделяющих знаения на примерно равные участки
correlation	Корреляция между физическим расположением значений на диске и логическим порядком значений. (При -1 или +1 ожидается, что сканирование по индексу будет дешевле, чем когда значение близко к нулю, т.е. значения более упорядочены (см. пример выше с выбором, что дешевле – сканирование или проход по индексу)

Статистика

- default_statistics_target – число элементов в массивах в таблице предыдущего слайда. Задается:
  - На уровне сервера
  - На уровне сессии
  - На уровне таблицы

Методы соединения

- Nested loops
- Hash join
- Hash anti-join
- Hash semi-join
- Merge join
- Иные способы
- Outer join
- Nested loop outer join
- Hash outer join

Nested loops

- ```
  foreach o in (outer){
      foreach i in (inner){
       if o.join_key=i.join_key {
          outstream.put(o,i);
       }
    }
  }
  ```
- Чего уж проще?
- Подходит для outer join, но задает порядок соединения – для left outer join сначала левая таблица, потом правая, для right outer join, соответственно, наоборот.
  ```

Nested loops - продолжение

При обычном сравнении для более-менее больших таблиц сервер использует nested loop неохотно, если только таблицы не совсем маленькие, но вот тогда – только его.

Единственный способ для соединений не по =

```
explain(analyze, buffers)
select * from t1, t1 t
where t.id between t1.id and t1.id+10
limit 100
Limit  (cost=0.43..7.90 rows=100 width=210) (actual time=0.062..0.145 rows=100 loops=1)
  Buffers: shared hit=41
  ->  Nested Loop  (cost=0.43..8307625710.00 rows=111111111111 width=210) (actual time=0.060..0.135 rows=100 loops=1)
        Buffers: shared hit=41
        ->  Seq Scan on t1  (cost=0.00..27242.00 rows=1000000 width=105) (actual time=0.020..0.021 rows=10 loops=1)
              Buffers: shared hit=1
        ->  Index Scan using t1_pkey on t1 t  (cost=0.43..7196.49 rows=111111 width=105) (actual time=0.005..0.007
rows=10 loops=10)
              Index Cond: ((id >= t1.id) AND (id <= (t1.id + 10)))
              Buffers: shared hit=40
Total runtime: 0.232 ms
```

Hash join – как работает

- hash(table1.val)=hash(table2.val)
- Обычный подход: строим хеш по одной таблице, прогоняем вторую и смотрим
- А если не умещается в памяти?
  - Вариант 1
    - Хешируем из меньшей таблицы сколько можем.
    - Прогоняем вторую
    - И так далее, пока данные из первой не кончатся
  - GRACE Hash join
    - Хешируем обе таблицы некоторой функцией H_1
    - Получаем набор групп для каждой из таблиц. Если T_1.key=T_2.key, то и $H_1(T_1.key)= H_1(T_2.key)$
    - Соединяем каждую пару групп в памяти

Hash join

- explain(analyze, buffers)
  select * from t1, t1 t
  where t1.id=10*t.id/10

```
Hash Join  (cost=56344.00..160892.00 rows=1000000 width=210) (actual time=19083.867..25519.331 rows=1000000 loops=1)
  Hash Cond: (((10 * t.id) / 10) = t1.id)
  Buffers: shared hit=3548 read=30936, temp read=29658 written=29596
  ->  Seq Scan on t1 t  (cost=0.00..27242.00 rows=1000000 width=105) (actual time=0.012..467.418 rows=1000000 loops=1)
        Buffers: shared hit=1790 read=15452
  ->  Hash  (cost=27242.00..27242.00 rows=1000000 width=105) (actual time=19079.580..19079.580 rows=1000000 loops=1)
        Buffers: 8192  Batches: 32  Memory Usage: 3980kB
        Buffers: shared hit=1758 read=15484, temp written=14767«
        ->  Seq Scan on t1  (cost=0.00..27242.00 rows=1000000 width=105) (actual time=0.002..18387.413 rows=1000000
loops=1)
              Buffers: shared hit=1758 read=15484
Total runtime: 25582.276 ms
```

Проверка соответствия

Построение хеш-таблицы

Размер таблицы и число проходов

Hash join - продолжение

- set work_mem='256MB‘
- Hash Join (cost=39742.00..109484.00 rows=1000000 width=210) (actual time=850.529..3575.678 rows=1000000 loops=1)
  Hash Cond: (((10 * t.id) / 10) = t1.id)
  Buffers: shared hit=3804 read=30680
  -> Seq Scan on t1 t (cost=0.00..27242.00 rows=1000000 width=105) (actual time=0.011..504.974 rows=1000000 loops=1)
        Buffers: shared hit=1918 read=15324
  -> Hash (cost=27242.00..27242.00 rows=1000000 width=105) (actual time=850.303..850.303 rows=1000000 loops=1)
        Buckets: 131072 Batches: 1 Memory Usage: 125977kB
        Buffers: shared hit=1886 read=15356
        -> Seq Scan on t1 (cost=0.00..27242.00 rows=1000000 width=105 (actual time=0.004..460.874 rows=1000000 loops=1)
              Buffers: shared hit=1886 read=15356
  Total runtime: 3657.914 ms

Hash semi-join

- EXISTS, IN
- set work_mem='32MB';
  explain(analyze, buffers)
  select * from t1
  where exists(select * from t1 t where t.id=t1.id+1)
- Hash Semi Join (cost=39742.00..81734.00 rows=500000 width=105) (actual time=713.527..2230.119 rows=999999 loops=1)
  Hash Cond: ((t1.id + 1) = t.id)
  Buffers: shared hit=14020 read=20464
  -> Seq Scan on t1 (cost=0.00..27242.00 rows=1000000 width=105) (actual time=0.011..351.397 rows=1000000 loops=1)
  Buffers: shared hit=7026 read=10216
  -> Hash (cost=27242.00..27242.00 rows=1000000 width=4) (actual time=713.317..713.317 rows=1000000 loops=1)
  Buckets: 131072 Batches: 1 Memory Usage: 27344kB
  Buffers: shared hit=6994 read=10248
  -> Seq Scan on t1 t (cost=0.00..27242.00 rows=1000000 width=4) (actual time=0.005..432.381 rows=1000000 loops=1)
  Buffers: shared hit=6994 read=10248
  Total runtime: 2290.837 ms

Hash anti-join

- NOT EXISTS, NOT IN
- explain(analyze, buffers)
  select * from t1
  where not exists(select * from t1 t where t.id=t1.id+1)
- Hash Anti Join (cost=39742.00..81734.00 rows=500000 width=105) (actual time=2090.394..2090.395 rows=1 loops=1)
  Hash Cond: ((t1.id + 1) = t.id)
  Buffers: shared hit=14148 read=20336
  -> Seq Scan on t1 (cost=0.00..27242.00 rows=1000000 width=105) (actual time=0.013..336.467 rows=1000000 loops=1)
  Buffers: shared hit=7090 read=10152
  -> Hash (cost=27242.00..27242.00 rows=1000000 width=4) (actual time=728.698..728.698 rows=1000000 loops=1)
  Buckets: 131072 Batches: 1 Memory Usage: 27344kB
  Buffers: shared hit=7058 read=10184
  -> Seq Scan on t1 t (cost=0.00..27242.00 rows=1000000 width=4) (actual time=0.005..438.681 rows=1000000 loops=1)
  Buffers: shared hit=7058 read=10184
  Total runtime: 2094.008 ms

Hashed subplan

- NOT EXISTS, NOT IN
- explain(analyze, buffers)
- select * from t1
  where t1.id not in(select t.id from t1 t)
- Seq Scan on t1 (cost=292.00..584.00 rows=5000 width=105) (actual time=11.784..11.784 rows=0 loops=1)
  Filter: (NOT (hashed SubPlan 1))
  Rows Removed by Filter: 10000
  Buffers: shared hit=334
  SubPlan 1
  -> Seq Scan on t1 t (cost=0.00..267.00 rows=10000 width=4) (actual time=0.005..2.905 rows=10000 loops=1)
  Buffers: shared hit=167
  Planning time: 0.134 ms
  Execution time: 11.835 ms

Merge join

- Сливаются два отсортированных списка строк
- Главное отличие – нет правой и левой стороны
- Подходит для outer join-ов
  - Более того, **не определяет жестко порядок соединения таблиц**
- Чуть ли не единственный способ реализовать full outer join

Merge join - продолжение

- explain(analyze, buffers)
  select * from t1 t, t1 where t1.id=t.id
- Merge Join (cost=0.85..101450.85 rows=1000000 width=210) (actual time=0.021..1982.601 rows=1000000 loops=1)
  Merge Cond: (t.id = t1.id)
  Buffers: shared hit=20956 read=18998
  -> Index Scan using t1_pkey on t1 t (cost=0.42..43225.43 rows=1000000 width=105) (actual time=0.009..280.390 rows=1000000 loops=1)
  Buffers: shared hit=19977
  -> Index Scan using t1_pkey on t1 (cost=0.42..43225.43 rows=1000000 width=105) (actual time=0.006..961.402 rows=1000000 loops=1)
  Buffers: shared hit=979 read=18998
  Total runtime: 2044.910 ms

Merge left join

- explain(analyze, buffers)
  select * from t1 t left outer join t1 on t1.id=t.id
- Merge Left Join (cost=0.85..101450.85 rows=1000000 width=210) (actual time=0.021..2135.078 rows=1000000 loops=1)
  Merge Cond: (t.id = t1.id)
  Buffers: shared hit=19983 read=19971
  -> Index Scan using t1_pkey on t1 t (cost=0.42..43225.43 rows=1000000 width=105) (actual time=0.009..272.459 rows=1000000 loops=1)
  Buffers: shared hit=19977
  -> Index Scan using t1_pkey on t1 (cost=0.42..43225.43 rows=1000000 width=105) (actual time=0.006..1153.697 rows=1000000 loops=1)
  Buffers: shared hit=6 read=19971
  Total runtime: 2195.236 ms

Outer joins

- Могут быть реализованы с помощью nested loop или merge join

Иное

- Subqueries
- Scalar subqueries
  - Dependent subqueries
- Subqueries во FROM

Postgres PROFESSIONAL

Subquery

- select * from table t where exists(select ….)
- select (select col from table2 where …) from table1
  - К сожалению, не кеширует
- select * from (select * from …)
  - lateral

Subqueries во FROM

- Могут подниматься вверх

WITH

- Всегда создается временная таблица
- work_mem

Остальные параметры оптимизатора

- geqo_* - generic query optimizer
- geqo_threshold – с какого количества таблиц включать GEQO
- constraint_exclusion – включать ли проверку check для запросов. По умолчанию стоит только partition – только для inheritance & union all – подзапросов. Интересный момент:
- ```
  create view tv as
    select 'T1' as src, id from t1
    union all
    select 'T2', id from t2
  ```
  ```

constraint_exclusion

- explain(analyze, buffers)
  select * from
   (values('T1',1,100000),('T4',10,100000),('T3', 20, 30)) as v(src, s,e), tv
   where tv.src=v.src and tv.id between v.s and v.e

- Nested Loop (cost=0.42..31438.53 rows=1833 width=76) (actual time=0.056..**171.934** rows=100000 loops=1)
  Buffers: shared hit=4559
  -> Values Scan on "*VALUES*" (cost=0.00..0.04 rows=3 width=40) (actual time=0.002..0.011 rows=3 loops=1)
  -> Append (cost=0.42..10473.38 rows=612 width=36) (actual time=21.767..51.698 rows=33333 loops=3)
        Buffers: shared hit=4559
        -> Index Only Scan using t1_pkey on t1 (cost=0.42..10116.42 rows=556 width=36) (actual time=12.673..31.229 rows=33333 loops=3)
              Index Cond: ((id >= "*VALUES*".column2) AND (id <= "*VALUES*".column3))
              Filter: ("*VALUES*".column1 = 'T1'::text)
              Rows Removed by Filter: 33334
              Heap Fetches: 200002
              Buffers: shared hit=4006
        -> Index Only Scan using t2_pkey on t2 (cost=0.29..356.96 rows=56 width=36) (actual time=17.197..17.197 rows=0 loops=3)
              Index Cond: ((id >= "*VALUES*".column2) AND (id <= "*VALUES*".column3))
              Filter: ("*VALUES*".column1 = 'T2'::text)
              Rows Removed by Filter: 66667
              Heap Fetches: 0
              Buffers: shared hit=553
  Total runtime: 176.929 ms

constraint_exclusion - продолжение

```
create or replace function tvf(p_src text, s int, e int)
returns table(src text, id int) as
$code$
begin
 case p_src
  when 'T1' then
    return query select p_src, t1.id from t1 where t1.id between s and e;
  when 'T2' then
    return query select p_src, id from t2 where t2.id between s and e;
  else
    null;
  end case;
end;
$code$
stable
language plpgsql
```

constraint exclusion - продолжение

- ```
  explain(analyze, buffers)
  select * from
   (values('T1',1,100000),('T4',10,100000),('T3', 20, 30)) as v(src, s,e), lateral tvf(v.src, v.s,v.e)
  ```
- ```
  Nested Loop  (cost=0.25..60.29 rows=3000 width=76) (actual time=68.347..99.090
  rows=100000 loops=1)
    Buffers: shared hit=2005
    ->  Values Scan on "*VALUES*"  (cost=0.00..0.04 rows=3 width=40) (actual
  time=0.002..0.008 rows=3 loops=1)
    ->  Function Scan on tvf  (cost=0.25..10.25 rows=1000 width=36) (actual
  time=22.798..25.733 rows=33333 loops=3)
          Buffers: shared hit=2005
  Total runtime: 104.443 ms
  ```

Параметры - продолжение

- cursor_tuple_fraction (floating point) – какая часть курсора ожидается быть полученной. По умолчанию 0.1 – влияет на выбор плана для курсора. Если поставить в 1.0, то планы для запросов курсоров будут планироваться так же, как и планы для обычных запросов.
- В чем разница? В том, что для обычных запросов планы строятся с целью выполнить результат максимально быстро, а не максимально быстро получить первую строку; кстати, подобное же действие оказывает LIMIT.

Параметры - продолжение

- from_collapse_limit – оптимизатор будет пытаться поднять вложенные подзапросы во from, если только получившийся список таблиц будет не больше этого параметра. По умолчанию 8; без особой нужды и твердого понимания лучше не трогать.
- join_collapse_limit – аналогично с join. Если выставить в 1, то можно сохранить порядок соеднения, указанный в запросе. По умолчанию равен from_collapse_limit. Если оптимизатор выбирает неверный порядок соединения, то можно воспользоваться.

Функции

- Скалярые
- Табличные

Скалярные

- IMMUTABLE – для одних и тех же данных всегда возвращают одно и то же
- STABLE – всегда возвращают одно и то же значение во время выполнения запроса;
- VOLATILE – необходимо вызывать каждый раз. По умолчанию.
- COST – стоимость в cpu_operator_cost.
- RETURN NULL ON NULL INPUT/STRICT – и так все понятно

Порядок вычисления условий

- В каком порядке будет вычислено выражение f1() and f2()?
- И будет ли оно short-circuit или нет?

Порядок вычисления - продолжение

```
create or replace function f1() returns boolean as $code$
begin
  raise notice 'f1';
  return true;
end; $code$
cost 300 language plpgsql;

create or replace function f2() returns boolean as $code$
begin
  raise notice 'f2';
  return true;
end;
$code$
cost 200 language plpgsql;
```

cost – стоимость в cpu_operator_cost, 0.0025

Порядок вычисления - продолжение

- `select 1 where f1() and f2()`
- `ЗАМЕЧАНИЕ:  f2`
  `ЗАМЕЧАНИЕ:  f1`
- `Вычисляются в порядке стоимости, а не в порядке записи`
- `Но short-circuit`

А что насчет coalesce?

- select coalesce(f2(),f1())
  - ЗАМЕЧАНИЕ: f2
- coalesce – short-circuit

Табличные

- returns setof record/type; returns table(colum definitions...)
- ROWS – ожидаемое число строк. По умолчанию 1000.

Встраивание функций (inlining)

- Функции могут быть развернуты прямо в тело запроса
- `LANGUAGE SQL, STABLE/IMMUTABLE, не SECURITY DEFINER, нет SET,`
- Скалярные
  - Не RETURNS RECORD
  - Просто select (нет подзапросов, CTE, GROUP, агрегатов)
- Табличные
  - Просто select
  - Как view с параметрами
- Не всегда это может быть хорошо

Встраивание функций (inlining)

```
create function t1func(sv int, ev int, str text) returns table(id int, descr text) as
 $code$
  select t1.id, t1.val|| str from t1 where id between sv and ev;
 $code$
 language sql
 stable

explain
 select * from t1, t1func(t1.id,t1.id+10,'->text') where t1.id=10


Nested Loop  (cost=0.57..78.70 rows=1111 width=210) (actual time=0.025..0.036 rows=11 loops=1)
  ->  Index Scan using t1_pkey on t1  (cost=0.29..8.30 rows=1 width=105) (actual time=0.010..0.011
rows=1 loops=1)
        Index Cond: (id = 10)
  ->  Index Scan using t1_pkey on t1 t1_1  (cost=0.29..56.51 rows=1111 width=105) (actual
time=0.005..0.010 rows=11 loops=1)
        Index Cond: ((id >= t1.id) AND (id <= (t1.id + 10)))
Planning time: 0.280 ms
Execution time: 0.074 ms
```

Транзитивность

- ```
  explain analyze
  select * from generate_series(1,1000) as g1(n),
  generate_series(1,1000) as g2(n) where g1.n=g2.n and g2.n=100
  ```
- ```
  Nested Loop  (cost=0.01..75.25 rows=25 width=8) (actual time=0.563..0.938
  rows=1 loops=1)
    ->  Function Scan on generate_series g1  (cost=0.00..12.50 rows=5
  width=4) (actual time=0.281..0.469 rows=1 loops=1)
          Filter: (n = 100)
          Rows Removed by Filter: 999
    ->  Function Scan on generate_series g2  (cost=0.00..12.50 rows=5
  width=4) (actual time=0.280..0.467 rows=1 loops=1)
          Filter: (n = 100)
          Rows Removed by Filter: 999
  Planning time: 0.114 ms
  Execution time: 1.015 ms
  ```
- В запросах это не очень надо; а вот во view надо
- Не работает с < и >

Сортировка

- Зачем надо
  - ORDER BY
  - GROUP BY – выгодна при большом числе групп
  - Оконные и агрегирующие функции (string_agg, array_agg и т.п.)
- Как?
  - Быстрая сортировка (quicksort)
  - TOP-N quicksort
  - Merge sort
- Недостатки: для сортировки большого количества требуются значительные ресурсы памяти и процессора; кроме того, узел сначала должен прочитать весь вход прежде чем он сможет что-то отдать на выход.

Агрегация

- Требует отдельного узла в дереве плана
  - Хеширование – оптимально для не очень большого числа групп
  - Сортировка – оптимальная для большого числа групп
- min/max могут быть получены через индекс (в том числе многостолбцовый); запрос
  select * from t where t.date=(select max(t1.date) from t t1 where t1.date<t.date)) – вполне ок.

LIMIT, OFFSET

- Может заставить оптимизатор выбрать план, который быстрее возвращает первую строку, а не весь результат – склонность к nested loop и сортировке через индекс.
- Если хочется сказать limit 10, offset 100000, то для выполнения всего этого сначала надо получить 100000 строк и никуда от этого В ПРИНЦИПЕ не денешься, т.е. быстрым это не будет никогда.

Postgres PROFESSIONAL

PREPARED STATEMENTS

```
PREPARE name [ ( data_type [, ...] ) ] AS statement

DEALLOCATE name

EXECUTE name
```

PREPARED STATEMENTS - продолжение

- Зачем это надо?
  - Производительность
  - Безопасность

PREPARED STATEMENTS - продолжение

Планы в зависимости от параметров

```
prepare sth(int,int) as
 select id from t1 where id between $1 and $2

explain(analyze, buffers)
  execute sth(1,1000000)
Seq Scan on t1  (cost=0.00..32242.00 rows=1000000 width=4) (actual
time=0.009..5823.402 rows=1000000 loops=1)
  Filter: ((id >= 1) AND (id <= 1000000))
  Buffers: shared hit=1726 read=15516
Total runtime: 5872.688 ms
```

PREPARED STATEMENTS - продолжение

```
explain(analyze, buffers)
  execute sth(1,100)


Index Only Scan using t1_pkey on t1  (cost=0.42..11.32 rows=95 width=4) (actual
time=0.019..0.053 rows=100 loops=1)
  Index Cond: ((id >= 1) AND (id <= 100))
  Heap Fetches: 100
  Buffers: shared hit=5
Total runtime: 0.088 ms
```

Postgres PROFESSIONAL

ANALYZE

- Собирает статистику по
- ANALYZE - базе,
- ANALYZE thetable - таблице,
- ANALYZE thetable(thecolumnone) - или колонкам

Общие принципы

- Надо знать, какие бывают методы (доступа, соединения, агрегации…) и в каких случаях какие необходимо применить
- Посмотреть, какие реально используются
- Если используются не те, что требуются, выяснить, кто ошибается – разработчик или оптимизатор (оптимизатор ошибается реже)
- Устранить проблему
- Чудес не бывает :-(

Кардинальность и селективность

- Кардинальность – *сколько* строк?
- Селективность – *какая часть* строк? (т.е. <=1), чем значение меньше, тем она выше

Общий подход

- Главный принцип: **чем меньше данных, тем быстрее**
- **Следовательно, надо стремиться к тому, чтобы наибольшая часть ненужных строк отсекалась на самых ранних стадиях выполнения запроса.**
- Соединение таблиц по возможности (это доступно не всегда) должно начинаться с таблицы с наибольшей *селективностью*, с той, в которой отношение числа отфильтрованных строк к общему числу строк минимально; каждая следующая таблица также должна, в свою очередь, выбираться из кандидатов с наибольшей селективностью.
- Принцип выше *не обязательно даст оптимальный план*, но по крайней мере почти наверняка не даст безумный. Например: таблица «пользователи» (10 млн строк), таблица «состояние» (два значения – «активен/заблокирован»), таблица «Страна».
  - Запрос «Все активные пользователи из Андорры» - первая пара «страны», «пользователи», потом «состояние», Index scan стран, Index scan/Bitmap heap scan пользователей, table scan «состояние».
  - Запрос «Все заблокированные пользователи из Китая с именем Евлогий» - index scan «пользователи», index scan «страна», table scan «состояние».

Особый случай - большие слабо прокешированные таблицы

- Вне зависимости от селективности следует стремиться к тому, чтобы такие таблицы соединялись в последнюю очередь.
- LATERAL и секционирование

Postgres PROFESSIONAL

Общие принципы - продолжение

Знайте ваши данные!

Экскаваторщики почти никогда не бывают кормящими женщинами, но сервер об этом не знает

Что делать, если план не соответствует ожиданиям

- Здраво оценить, может ли запрос с имеющимися данными, таблицами, индексами вообще выполняться с желаемой скоростью
- Кто неправ
  - Разработчик
  - Оптимизатор
- Кто виноват
  - Выяснить, что не так
  - Выяснить, почему не так
- Что делать
  - Переписать запрос
  - Создать/изменить индекс
  - Разбить запрос на части
  - Предпоследний шанс - поставить параметры оптимизатора (и не забыть их вернуть в прежнее состояние)
  - Действительно последний шанс – переписать навигационно.

Неверный метод доступа

- Основная причина – неверная/старая статистика, неактуальный VACUUM, отсутствие подходящих альтернатив
- Что делать
  - Собрать статистику
  - Настроить VACUUM
  - Создать подходящий индекс (в т.ч. условный, функциональный, попытаться добиться Index-Only Scan)
  - CLUSTER, pg_reorg/pg_repack
  - Разбить таблицу на части

Неверный метод соединения

- И опять собрать статистику, возможно, повысив точность (default_statistics_target)
- Построить подходящий индекс (Index-Only scans? – и снова vacuum!)
- Разбить запрос на части

Неверный порядок соединения

- Выяснить, почему оптимизатор выбрал именно такой способ
- Задать порядок скобками - ((t1 join t2) join (t3 join t4)) и параметр join_collapse_limit=1

Параметры оптимизатора

- random_page_cost
- enable_hashjoin etc

Для **стремительных** запросов надо

- Минимальный объем данных
- Nested loop
- Сортировка через индекс

Materialized view как своими руками, так и не своими

- Группировка нередко становится значительной проблемой
- Что делать?
- Materialized view!
- Обновляемые периодически (встроенные) и немедленно (в транзакции); можно придумать комбинации
- В силу транзакционной природы СУБД при завершении транзакции одновременно становятся видны как данные, так и агрегаты по ним.
- Одной группировкой дело не ограничивается
- view как основа для matview. При обновлении удаляем в matview затронутые строки и вставляем выборку из view

Меньше данных (в байтах) - быстрее работает. Пакуем таблицы

- Друзья (friends)
- Множество пар integer, integer
- Решение в лоб:
- ```
  create table friend(
    id int,
    friend_id int,
    constraint friend_pk primary key (id, friend_id)
  )
  ```
  ```

Меньше данных - быстрее

- ```
  insert into friend(id, friend_id)
  select n, fn
  from generate_series(1,100000) as gs(n),
  (select (100000*random())::integer from generate_series(1,50)) as
  gs1(fn)
  order by random()
  ```
- 173 МБ данных, 139 МБ индексов
- Все правильно: 5000000*(8+26) + еще заголовки страниц – что-то так и получается; 5000000*(8+14) + заголовки страниц + неполные страницы – все ожидаемо
  ```

Меньше данных - быстрее

- explain(analyze, buffers)
  select * from friend where id between 300 and 3000
- Bitmap Heap Scan on friend (cost=3450.76..27685.49 rows=140715 width=8)
  (actual time=52.618..460.941 rows=135050 loops=1)
    Recheck Cond: ((id >= 300) AND (id <= 3000))
    Buffers: shared hit=3 read=22584
    -> Bitmap Index Scan on friend_pk (cost=0.00..3415.58 rows=140715
  width=0) (actual time=45.578..45.578 rows=135050 loops=1)
          Index Cond: ((id >= 300) AND (id <= 3000))
          Buffers: shared hit=2 read=500
  Total runtime: 471.025 ms

Меньше данных - быстрее

- Можно ли что-то сделать? Да!
- ```
  create table friend2(
   id int,
   friend_id int[],
   constraint friend2_pk primary key (id)
  )
  ```
- ```
  insert into friend2(id, friend_id)
  select n, array(select (100000*random())::integer from
  generate_series(1,50))
  from generate_series(1,100000) as gs(n)
  order by random()
  ```
- Таблица – 25 МБ, индекс – 2784 КБ, итого 27-28 МБ. Более чем в 10 раз меньше.

Меньше данных - быстрее

- Почему?
- 26 байт на строку, 14 байт на индекс
- Индексируется в 50 раз меньше элементов
- Но несколько неудобно. Делаем view:
- ```
  create or replace view v_friend2 as
  select f.id, unnest(f.friend_id) as friend_id from friend2 f
  ```
  ```

Меньше данных - быстрее

```
explain(analyze, buffers)
select * from v_friend2 where id between 300 and 3000
```

```
Bitmap Heap Scan on friend2 f  (cost=66.96..4600.56 rows=260200 width=228) (actual
time=0.812..28.007 rows=135050 loops=1)
  Recheck Cond: ((id >= 300) AND (id <= 3000))
  Buffers: shared hit=1878
  ->  Bitmap Index Scan on friend2_pk  (cost=0.00..66.31 rows=2602 width=0) (actual
time=0.502..0.502 rows=2701 loops=1)
        Index Cond: ((id >= 300) AND (id <= 3000))
        Buffers: shared hit=12
Total runtime: 35.110 ms
```

Даром ничего не бывает —

1) Надо следить, чтобы при обновлениях таблица не замусоривалась.
2) Может пострадать параллелизм.

Типичные проблемный запрос

- DISTINCT — стало быть, выбираем ЛИШНИЕ данные
- LEFT OUTER JOIN — строго задает порядок соединения и получает ВСЕ данные
- OFFSET на весь результат — всегда нужно получить как минимум OFFSET строк
- И еще все делаем два раза (первый раз — общее количество, второй — вырезка страницы)
- Например:

```
select distinct u.user_id, goods.name, goods.price,
….
addr.street, addr.house
from user u
 left outer join user_x_reseller uxr on u.id=uxr.user_id
 left outer join reseller_x_stock rxs on uxr.reseller_id=rxs.stock_id
 left outer join stock_x_goods sxg on rxs.stock_id=sxd.stock_id
 left outer join goods g on sxd.goods_id=g.id
 …
 left outer join address_x_user axu on u.id=axu.user_id
 left outer join address addr on addr.id=axu.address_id
where uxr.reseller_id=$1 and goods.type='TABLET'
order by goods.added
limit 25 offset 20000
```

Типичные ошибки

- ```
  select array_agg(uxr.user_id) into uids from
   user_x_reseller uxr on u.id=uxr.user_id
      join reseller_x_stock rxs on uxr.reseller_id=rxs.stock_id
      join stock_x_goods sxg on rxs.stock_id=sxd.stock_id
      join goods g on sxd.goods_id=g.id
   where uxr.reseller_id=$1 and goods.type='TABLET'
   order by g.added;
  ```

- Сколько – array_length(uids,1)

- Результат -
  ```
  select u.id, addr.street,…
    from unnest(uids(1:25)) as u(id)
      [left outer] join address_x_user axu on aux.user_id=u.id
  …
  ```
- Если все время получаются сложные запросы, то
  - Либо есть какие-то серьезные проблемы с дизайном схемы
  - Либо они как-то неаккуратно формируются
  ```

Получение данных на клиенте

- По умолчанию клиент получает сразу всю выборку
- Для интерактивных приложений редко надо больше 10-100 строк за один раз
- Пожалуйста, внимательно следите за этим ИЛИ
- Используйте курсоры.

Postgres PROFESSIONAL

Литература

- http://www.postgresql.org/docs/current/static/index.html
- Dan Tow, SQL Tuning