SQL: Query optimization in practice

@jsuchal (@rubyslava #21)

Query optimization

"The fastest query is the one you never make"

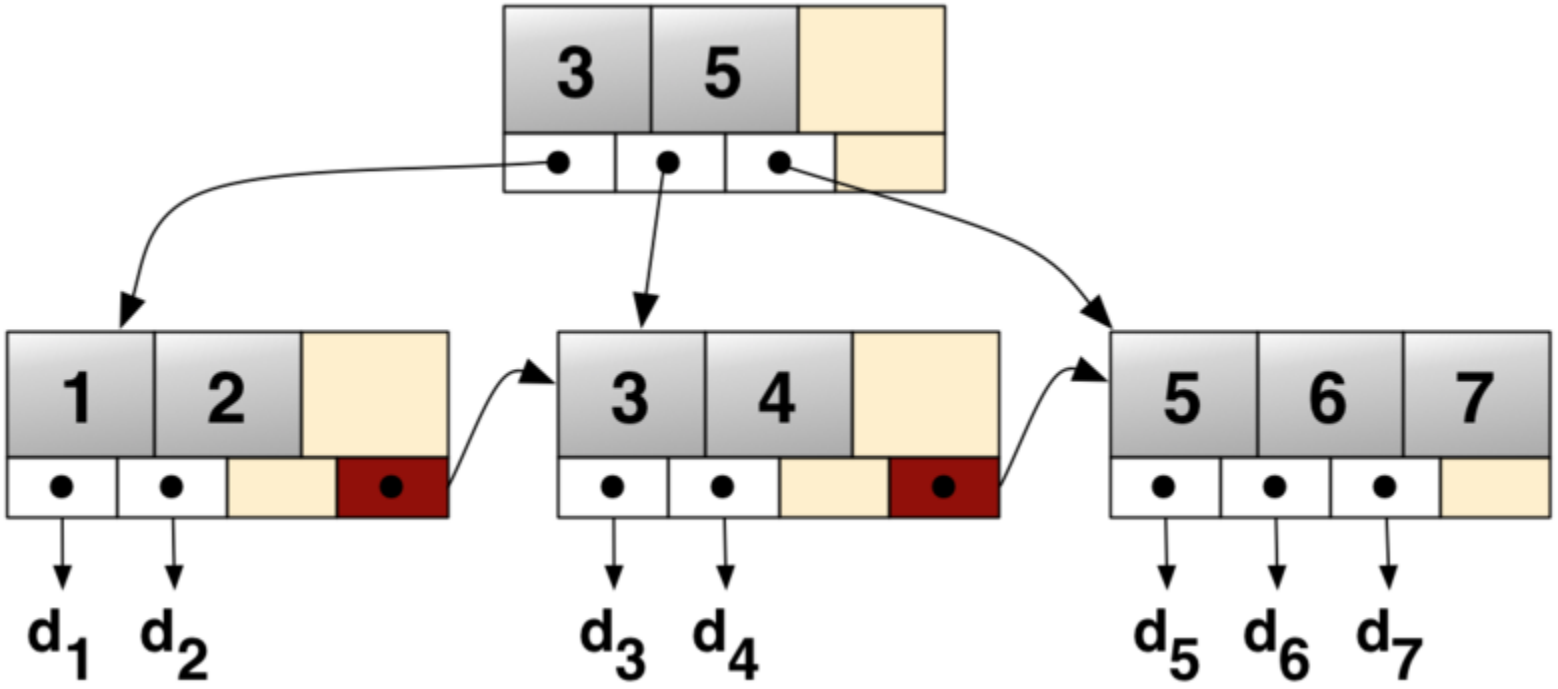
Query planner

- **Problem:** Find the most efficient way to execute this query

```
SELECT s.* FROM students s
    JOIN enrolments e ON s.id = e.student_id
    JOIN courses c ON e.course_id = c.id
    WHERE s.gender = 'F' AND e.year = '2012'
        AND c.name = 'Programming in Ruby'
```

- **Use**
  - Indexes
  - Statistics about data
  - Access methods
  - Join types
  - Aggregation / Sort / Pipelining

B+ tree index



- leaf nodes
  - pointers to table data (heap)
  - doubly linked list (fast in-order traversal)

Access methods

- Sequential scan (a.k.a. full table scan)
  - fetch data directly from table (heap)
- Index scan
  - fetch data from table in index order
- Bitmap index scan + Bitmap heap scan
  - fetch data from table in table order
  - requires additional memory for intermediate bitmap
  -
- Index-Only scan (a.k.a. covering index)
  - fetch data from index only

Join types

- Nested loop
  - for each row find matching rows using join condition
  - basically a nested "for loop"
- Hash join
  - create intermediate hash table from smaller table
  - loop through larger table and probe against hash
  - recheck & emit matching rows
- Sort-Merge join
  - sort both tables on join attribute (if necessary)
  - merge using interleaved linear scan

Sort, Aggregate, Pipelining

- Sort
- Aggregate
  - Plain Aggregate
  - Sort Aggregate
  - Hash Aggregate
- Pipelining
  - stop execution after X rows are emitted
  - SELECT * FROM students LIMIT 10

Reading EXPLAIN (in PostgreSQL)

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM documents;
```

Plan

actual execution

I/O heavy?

est. startup cost

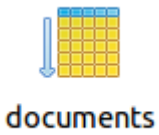
est. total cost

Node execution count

est. emitted rows

est. row size (B)

```
Seq Scan on do                .00..4688.45          3445 width=95)
(actual time=0.008..19        ws=183445 loops=1)
  Buffers: shared hit=2854
Total runtime: 28.376 ms
```



Reading EXPLAIN (2)

```
EXPLAIN SELECT * FROM documents WHERE id < 72284

Index Scan using documents_pkey on documents  (cost=0.00..19.74
rows=76 width=95) (actual time=0.008..0.076 rows=81 loops=1)
  Index Cond: (id < 72284)
  Buffers: shared hit=75
Total runtime: 0.102 ms
```

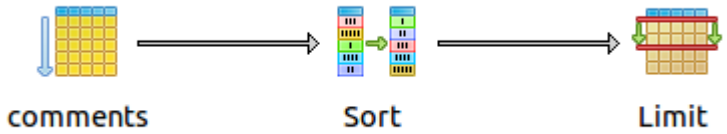
!!!


```
EXPLAIN SELECT * FROM documents WHERE id > 72284

Seq Scan on documents  (cost=0.00..5147.06 rows=183369
width=95) (actual time=0.017..28.905 rows=183363 loops=1)
  Filter: (id > 72284)
  Buffers: shared hit=2854
Total runtime: 37.752 ms
```

Reading EXPLAIN (3)

```
SELECT * FROM comments ORDER BY created_at DESC LIMIT 10
```

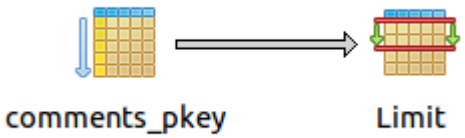



```
Limit  (cost=5.37..5.40 rows=10 width=245) (actual time=0.160..0.166 rows=10 loops=1)
  Buffers: shared hit=5
  ->  Sort  (cost=5.37..5.56 rows=75 width=245) (actual time=0.157..0.160 rows=10
loops=1)
        Sort Key: created_at
        Sort Method: top-N heapsort  Memory: 28kB"
        Buffers: shared hit=5
        ->  Seq Scan on comments  (cost=0.00..3.75 rows=75 width=245) (actual
           time=0.009..0.043 rows=91 loops=1)
              Buffers: shared hit=3
Total runtime: 0.229 ms
```

Reading EXPLAIN (4)

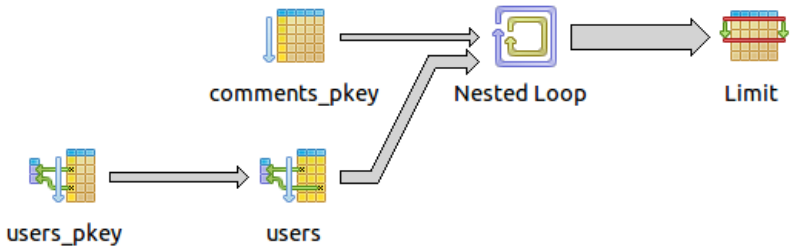
```
SELECT * FROM comments ORDER BY id DESC LIMIT 10

Limit  (cost=0.00..2.11 rows=10 width=245)
       (actual time=0.018..0.029 rows=10 loops=1)
  Buffers: shared hit=2
  ->  Index Scan Backward using comments_pkey on comments
      (cost=0.00..15.86 rows=75 width=245)
      (actual time=0.016..0.027 rows=10 loops=1)
        Buffers: shared hit=2
Total runtime: 0.081 ms
```



Reading EXPLAIN (5)

```
SELECT * FROM comments c
   JOIN users u ON c.user_id = u.id
   ORDER BY c.id DESC
   LIMIT 10
```




```
Limit  (cost=94.35..2375.31 rows=10 width=946)
  ->  Nested Loop  (cost=94.35..7393.42 rows=32 width=946)
        ->  Index Scan Backward using comments_pkey on comments c
            (cost=0.00..15.86 rows=75 width=245)
        ->  Bitmap Heap Scan on users u  (cost=94.35..98.36 rows=1 width=701)
              Recheck Cond: (id = c.user_id)
              ->  Bitmap Index Scan on users_pkey
                    (cost=0.00..94.34 rows=1 width=0)
                    Index Cond: (id = c.user_id)
```

Reading EXPLAIN

http://explain.depesz.com

exclusive	inclusive	rows x	rows	loops	node
0.002	0.438	↑ 1.0	10	1	→ Limit (cost=7382.01..7382.03 rows=10 width=946) (actual time=0.437..0.438 rows=10 loops=1)
0.099	0.436	↑ 3.2	10	1	→ Sort (cost=7382.01..7382.09 rows=32 width=946) (actual time=0.436..0.436 rows=10 loops=1) Sort Key: c.created_at Sort Method: top-N heapsort Memory: 28kB
0.133	0.337	↓ 1.6	51	1	→ Nested Loop (cost=94.35..7381.32 rows=32 width=946) (actual time=0.017..0.337 rows=51 loops=1)
0.022	0.022	↓ 1.2	91	1	→ Seq Scan on comments c (cost=0.00..3.75 rows=75 width=245) (actual time=0.005..0.022 rows=91 loops=1)
0.091	0.182	↑ 1.0	1	91	→ Bitmap Heap Scan on users u (cost=94.35..98.36 rows=1 width=701) (actual time=0.002..0.002 rows=1 loops=91) Recheck Cond: (id = c.user_id)
0.091	0.091	↑ 1.0	1	91	→ Bitmap Index Scan on users_pkey (cost=0.00..94.34 rows=1 width=0) (actual time=0.001..0.001 rows=1 loops=91) Index Cond: (id = c.user_id)

Tricks

Optimizing ORDER BY

```
SELECT * FROM documents ORDER BY created_at DESC

Sort  (cost=20726.12..21184.73 rows=183445 width=95)
      (actual time=199.680..231.948 rows=183445 loops=1)
  Sort Key: created_at
  Sort Method: external sort  Disk: 19448kB
  ->  Seq Scan on documents
      (cost=0.00..4688.45 rows=183445 width=95)
      (actual time=0.016..18.877 rows=183445 loops=1)
Total runtime: 245.865 ms

Index Scan using index_on_created_at on documents
  (cost=0.00..7894.56 rows=183445 width=95)
  (actual time=0.162..57.519 rows=183445 loops=1)
Total runtime: 67.679 ms
```

Optimizing GROUP BY

```
SELECT attachment_id, COUNT(*) FROM pages GROUP BY attachment_id
  LIMIT 10

Limit  (cost=0.00..65.18 rows=10 width=4)
  ->  GroupAggregate  (cost=0.00..267621.87 rows=41056 width=4)
        ->  Index Scan using idx_attachment_id on pages
            (cost=0.00..261638.63 rows=1114537 width=4)


SELECT attachment_id, COUNT(*) FROM pages GROUP BY attachment_id

HashAggregate  (cost=169181.05..169591.61 rows=41056 width=4)
  ->  Seq Scan on pages
      (cost=0.00..163608.37 rows=1114537 width=4)
```

Multicolumn / composite indexes

```
SELECT * FROM pages WHERE number = 1 ORDER BY created_at LIMIT 100

Limit  (cost=173572.08..173572.33 rows=100 width=1047)
  ->  Sort  (cost=173572.08..174041.45 rows=187749 width=1047)
        Sort Key: created_at
        ->  Seq Scan on pages
            (cost=0.00..166396.45 rows=187749 width=1047)
              Filter: (number = 1)

CREATE INDEX idx_number_created_at ON pages (number, created_at);

Limit  (cost=0.00..247.86 rows=100 width=1047)
  ->  Index Scan Backward using idx_number_created_at on pages
        Index Cond: (number = 1)
```

Multicolumn / composite indexes

- Order is important!
  - equality conditions first
- Usable for GROUP BY with WHERE conditions
- Mixed ordering
  - `ORDER BY a ASC, b DESC`
  - `CREATE INDEX idx ON t (a ASC , b DESC);`

Index-only scan / covering index

- MySQL, PostgreSQL 9.2+, ...
- Very useful for fast lookup
  - on m:n join tables - order is important!

Partial index

```
CREATE INDEX idx ON pages (number, created_at);

CREATE INDEX idx ON pages (number, created_at)
      WHERE number = 1;
```

- Index size
  - 34MB vs 5MB

Function / expression index

```
SELECT 1 FROM users
   WHERE lower(username) = 'johno';

CREATE INDEX idx_username
   ON users (lower(username));
```

"Hacks"

UNNEST

Problem: Show last 6 checkins for N given users

```
SELECT *,
    UNNEST(ARRAY(
        SELECT movie_id FROM checkins WHERE user_id = u.id
        ORDER BY created_at DESC LIMIT 6)
    ) as movie_id

    FROM users AS u
      WHERE u.id IN (2079077510, 1355625182, ...)
```

Exploiting knowledge about data

- Correlated columns
  - **Example:** Events stream
  - `ORDER BY created_at` vs. `ORDER BY id`
  - Exploit primary key

- Fetch top candidates, filter and compute
  - Fast query for candidates
  - expensive operations in outer query

```
SELECT * FROM (
    SELECT * FROM table ORDER BY c LIMIT 100
) AS t GROUP BY another column
```

Now what?

- Denormalization = redundancy
  - counters
  - duplicate columns
  - materialized views
- Specialized structures
  - nested set
- Specialized indexes
  - GIN, GiST, SP-GiST, Spatial
- Specialized engines
  - column stores, full text, graphs

Common caveats

- most selective first myth
- unique vs non-unique indexes
  - additional information planner can use
- Unnecessary subqueries
  - Good query planner **might** restructure it to join
- LEFT JOIN vs. JOIN
  - LEFT JOIN constraints joining order possibilities
  - not semantically equivalent!

Resources

- http://use-the-index-luke.com/
- http://www.postgresql.org/docs/9.2/static/indexes.html
- http://www.postgresql.org/docs/9.2/static/using-explain.html