PostgreSQL query planning and tuning

Federico Campoli

03 Mar 2016

Table of contents

1. Jargon
2. I find your lack of plan disturbing
3. I love it when a plan comes together
4. EXPLAIN! EXPLAIN!
5. I fight for the users!
6. Wrap up

Table of contents

1. Jargon
2. I find your lack of plan disturbing
3. I love it when a plan comes together
4. EXPLAIN! EXPLAIN!
5. I fight for the users!
6. Wrap up

Jargon

source https://commons.wikimedia.org/wiki/File:Klingon_Empire_Flag.svg

Jargon

Jargon

- **OID** Object ID, 4 bytes integer. Used to identify the system objects (tables, indices etc)
- **class** any relational object, table, index, view, sequence...
- **attribute** the table fields
- **execution plan** the steps required for executing a query
- **plan nodes** execution plan's steps
- **CBO** cost based optimizer
- **cost** arbitrary value used to determine a score for the plan nodes

Table of contents

1. Jargon
2. I find your lack of plan disturbing
3. I love it when a plan comes together
4. EXPLAIN! EXPLAIN!
5. I fight for the users!
6. Wrap up

I find your lack of plan disturbing

source http://apbialek.deviantart.com/art/Darth-Vader-171921375

Query stages

The query execution requires four stages

- Syntax validation
- Query tree generation
- Plan estimation
- Execution

Syntax validation

The query parser validates the query syntax using fixed rules.

Any error at this step will cause the execution to stop, returning a syntax error.

This early stage doesn't requires a system catalogue access.

The parser returns a normalised *parse tree* for the next step.

The query tree

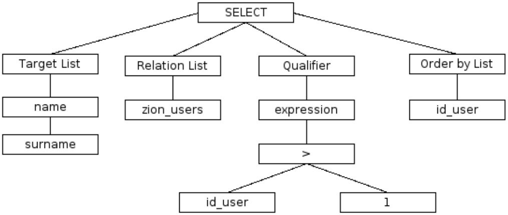
The query parser in the second stage look up the system catalogue and translates the parse tree into the *query tree*.

The query tree is the query's logical representation where any object involved is unique and described using the object id mapping.

The query tree

The query tree

SELECT name, surname FROM zion_users WHERE id_user>1 ORDER BY id_user;



The planner

The planner stage

The next stage is the query planner. The parser sends the generated query tree to the planner. The query planner reads the tree and generates all the possible execution plans. The planner, using the internal statistics,determines the **estimated** costs for each plan.

The execution plan with the minimum **estimated** cost is sent to the executor.

Old or missing statistics will result not-optimal execution plans and therefore slow queries.

The executor

The executor

The executor performs the plan nodes in the execution plan generated by the planner.

The workflow

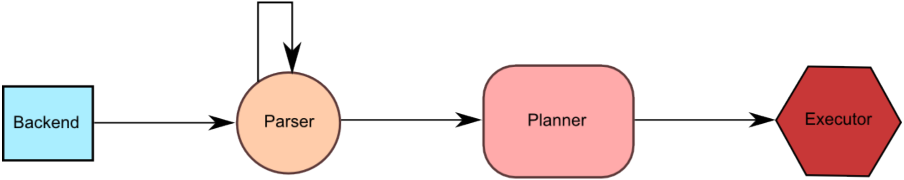


Figure : The query stages

Table of contents

1. Jargon
2. I find your lack of plan disturbing
3. I love it when a plan comes together
4. EXPLAIN! EXPLAIN!
5. I fight for the users!
6. Wrap up

I love it when a plan comes together

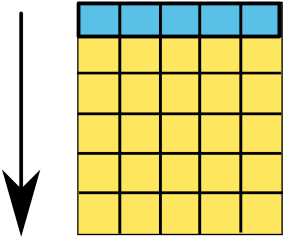
source http://clipperdata.com/blogpost/i-love-it-when-a-plan-comes-together/i-love-it-when-a-plan-comes-together/

The plan nodes

Scan nodes used by the executor to retrieve data from the relations
Join nodes used by the executor to perform joins of the data streams

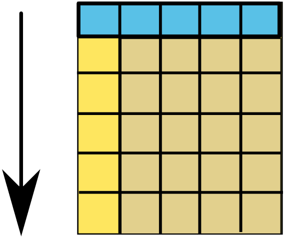
seq_scan

seq_scan: reads sequentially the table and discards the unmatched rows. The output is a data stream. Returns unsorted rows.



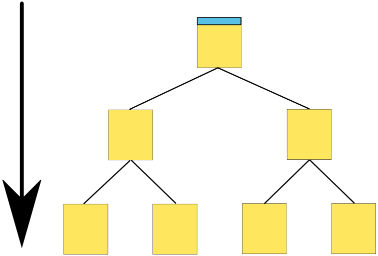
index_scan

index_scan: read the index tree with random disk read and gets the heap blocks pointed by the index. Returns sorted rows.



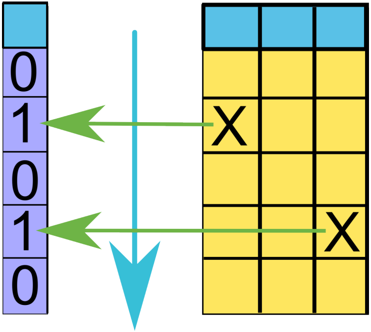
index_only_scan

index_only_scan: read the index tree with random disk read and returns the data without accessing the heap page. Returns sorted rows.



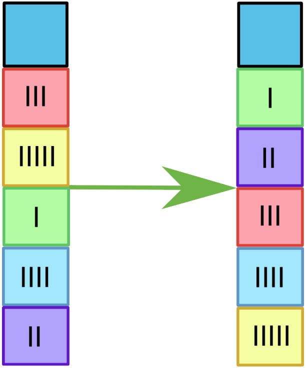
bitmap_index/heap scan

bitmap_index/heap scan: read the index sequentially generating a bitmap used to recheck on heap pages. It's a good compromise between seq_scan and a full index scan. Returns unsorted rows.



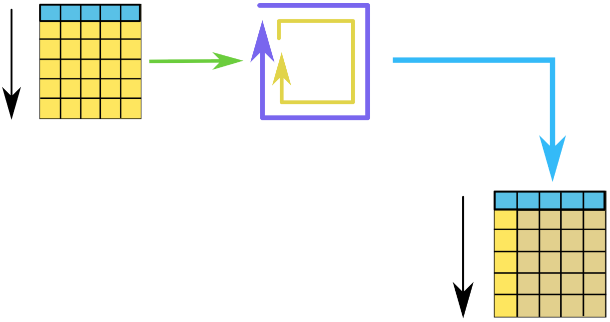
sort

sort : reads the rows and returns them in an ordered way like in queries with an ORDER BY



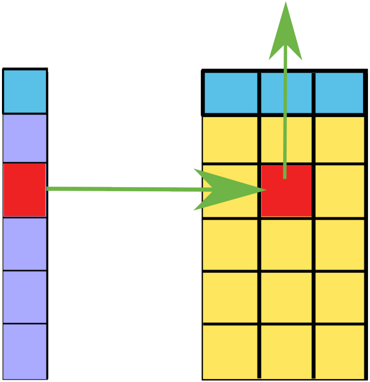
nested loop

nested loop join: The right relation is scanned once for every row found in the left relation. This strategy is easy to implement but can be very time consuming. However, if the right relation can be scanned with an index scan, this can be a good strategy. It is possible to use values from the current row of the left relation as keys for the index scan of the right.



hash join

hash join: the right relation is first scanned and loaded into a hash table, using its join attributes as hash keys. Next the left relation is scanned and the appropriate values of every row found are used as hash keys to locate the matching rows in the table.



merge join

merge join: Each relation is sorted on the join attributes before the join starts. Then the two relations are scanned in parallel, and matching rows are combined to form join rows. This kind of join is more attractive because each relation has to be scanned only once. The required sorting might be achieved either by an explicit sort step, or by scanning the relation in the proper order using an index on the join key.

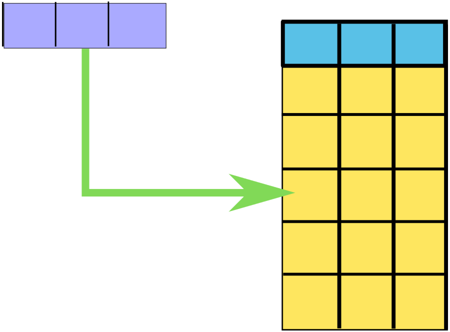


Table of contents

1. Jargon
2. I find your lack of plan disturbing
3. I love it when a plan comes together
4. EXPLAIN! EXPLAIN!
5. I fight for the users!
6. Wrap up

EXPLAIN! EXPLAIN!

source http://keepcalmandwatchdoctorwho.tumblr.com/post/5450959631/the-daleks-personally-i-like-it-when-they-shout

EXPLAIN

- Prepending EXPLAIN to any query will display the query's estimated execution plan .

EXPLAIN

- Prepending EXPLAIN to any query will display the query's estimated execution plan .
- The **ANALYZE** clause executes the query, discards the results and returns the real execution plan.

EXPLAIN

- Prepending EXPLAIN to any query will display the query's estimated execution plan .
- The **ANALYZE** clause executes the query, discards the results and returns the real execution plan.
- Using EXPLAIN ANALYZE with the DML queries will change the data. It is safe to wrap the EXPLAIN ANALYZE between BEGIN; and ROLLBACK;

Explain in action

For our example we'll create a test table with two fields, a serial and character varying.

```
test=# CREATE TABLE t_test
       (
             i_id   serial,
             v_value character varying(50)
       )
       ;
CREATE TABLE
```

Explain in action

Now let's add some rows to our table

Listing 1: Insert in table

```
test=# INSERT INTO t_test
              (v_value)
       SELECT
              v_value
       FROM
       (
              SELECT
                     generate_series(1,1000) as i_cnt,
                     md5(random()::text) as v_value
       ) t_gen
           ;
INSERT 0 1000
```

Explain in action

Let's generate the estimated plan for one row result

```
test=# EXPLAIN SELECT * FROM t_test WHERE i_id=20;
                       QUERY PLAN
----------------------------------------------------------
 Seq Scan on t_test  (cost=0.00..16.62 rows=3 width=122)
   Filter: (i_id = 20)
(2 rows)
```

- The cost is just an arbitrary value
- The values after the cost are the the startup and total cost
- The start up cost is the cost to deliver the first row to the next step
- The total cost is cost to deliver all the rows to the next step
- The value in rows is the planner's estimation for the total rows returned by the plan node
- The value in width is the estimated average row width in bytes

EXPLAIN ANALYZE

Now let's generate the real execution plan for one row result

```
test=# EXPLAIN ANALYZE SELECT * FROM t_test WHERE i_id=20;
                                             QUERY PLAN
--
    -----------------------------------------------------------------------------------

 Seq Scan on t_test  (cost=0.00..21.50 rows=1 width=37) (actual time
     =0.022..0.262 rows=1 loops=1)
   Filter: (i_id = 20)
   Rows Removed by Filter: 999
 Planning time: 0.066 ms
 Execution time: 0.286 ms
(5 rows)
```

- The values in actual time are the time, in milliseconds, required for the startup and total cost
- The value in rows is the number of rows returned by the step
- The value loops value is the number of times the step is executed
- On the bottom we have the planning time and the total execution time

Indices

Let's add an index on the i_id field

```
test=# CREATE INDEX idx_i_id ON t_test (i_id);
CREATE INDEX
```

Indices

```
test=# EXPLAIN ANALYZE SELECT * FROM t_test WHERE i_id=20;
                                                    QUERY PLAN
--
    ------------------------------------------------------------------------------------
 Index Scan using idx_i_id on t_test  (cost=0.28..8.29 rows=1 width=37) (
     actual time=0.035..0.036 rows=1 loops=1)
   Index Cond: (i_id = 20)
 Planning time: 0.252 ms
 Execution time: 0.058 ms
(4 rows)
```

The query is several times faster.

Controlling the planner

The cost based optimiser becomes cleverer for each major release. For example, if the query's filter returns almost the entire table, the database will choose the sequential scan which is by default 4 times cheaper than a full index scan.

```
test=# EXPLAIN ANALYZE SELECT * FROM t_test WHERE i_id>2;
                                              QUERY PLAN
--
    ---------------------------------------------------------------------------
 Seq Scan on t_test  (cost=0.00..21.50 rows=999 width=37) (actual time
     =0.012..0.467 rows=998 loops=1)
   Filter: (i_id > 2)
   Rows Removed by Filter: 2
 Planning time: 0.142 ms
 Execution time: 0.652 ms
(5 rows)
```

Controlling the planner

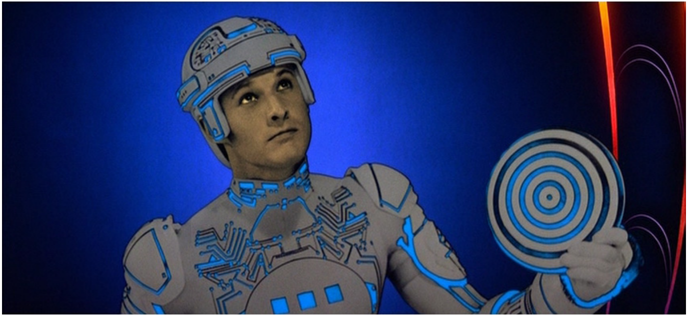
```
test=# SET enable_seqscan='off';
SET
test=# EXPLAIN ANALYZE SELECT * FROM t_test WHERE i_id>2;
                                                       QUERY PLAN

--
    ------------------------------------------------------------------------------
 Index Scan using idx_i_id on t_test  (cost=0.28..49.76 rows=999 width=37) (
     actual time=0.029..0.544 rows=998 loops=1)
   Index Cond: (i_id > 2)
 Planning time: 0.145 ms
 Execution time: 0.741 ms
(4 rows)
```

Table of contents

1. Jargon
2. I find your lack of plan disturbing
3. I love it when a plan comes together
4. EXPLAIN! EXPLAIN!
5. I fight for the users!
6. Wrap up

I fight for the users!



source http://tron.wikia.com/wiki/Tron

ANALYZE

ANALYZE

- ANALYZE gather statistics runs random reads on the tables
- The statistics are stored in the pg_statistic system table
- The view pg_stats translates the statistics in human readable format
- The parameter default_statistics_target sets the limit for the random read
- The statistic target can be fine tuned per column

Table statistics

Before starting the performance analysis check the statistics are up to date querying the view pg_stat_all_tables

```
test=# SELECT * FROM pg_stat_all_tables WHERE relname='t_test';
-[ RECORD 1 ]-------+------------------------------
relid               | 16546
schemaname          | public
relname             | t_test
seq_scan            | 6
seq_tup_read        | 5000
idx_scan            | 7
idx_tup_fetch       | 2980
n_tup_ins           | 1000
n_tup_upd           | 0
n_tup_del           | 0
n_tup_hot_upd       | 0
n_live_tup          | 1000
n_dead_tup          | 0
n_mod_since_analyze | 0
last_vacuum         |
last_autovacuum     |
last_analyze        |
last_autoanalyze    | 2015-09-24 04:51:51.622906+00
vacuum_count        | 0
autovacuum_count    | 0
analyze_count       | 0
autoanalyze_count   | 1
```

index statistics

Before starting the performance analysis check the indices are used querying the view pg_stat_all_indexes

```
test=# SELECT * FROM pg_stat_all_indexes WHERE relname='t_test';
-[ RECORD 1 ]-+---------
relid         | 16546
indexrelid    | 16552
schemaname    | public
relname       | t_test
indexrelname  | idx_i_id
idx_scan      | 7
idx_tup_read  | 2980
idx_tup_fetch | 2980
```

Controlling the planner

- enable_bitmapscan
- enable_hashagg
- enable_hashjoin
- enable_indexscan
- enable_indexonlyscan
- enable_material
- enable_mergejoin
- enable_nestloop
- enable_seqscan
- enable_sort
- enable_tidscan

Table of contents

1. Jargon
2. I find your lack of plan disturbing
3. I love it when a plan comes together
4. EXPLAIN! EXPLAIN!
5. I fight for the users!
6. Wrap up

Questions

Questions?

Boring legal stuff

Contacts and license

- Twitter: 4thdoctor_scarf
- Blog:http://www.pgdba.co.uk
- Brighton PostgreSQL Meetup: http://www.meetup.com/Brighton-PostgreSQL-Meetup/

PostgreSQL query planning and tuning

Federico Campoli

03 Mar 2016