



## PostgreSQL query planner's internals



How I Learned to Stop Worrying  
and Love the Planner



## 2 Why this talk?

- Why this query is so slow?
- Why planner is not using my index?
- What to do?

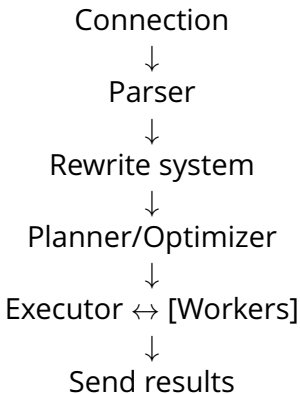


### 3 Where are we going?

- How planner works
- How we can affect it's work
- When it can go wrong
- Known limitations



## 4 The Path of a Query



all in single process (backend) beside background workers (parallel seq scan, 9.6+)



## 5 EXPLAIN command

```
explain (ANALYZE,VERBOSE,COSTS,BUFFERS,TIMING1) select * from t1;  
QUERY PLAN
```

-----  
Seq Scan on public.t1 (cost=0.00..104424.80 rows=10000000 width=8)  
(actual time=0.218..2316.688 rows=10000000 loops=1)

Output: f1, f2

Buffers: shared read=44248

I/O Timings: read=322.714<sup>2</sup>

Planning time: 0.024 ms

Execution time: 3852.588 ms

---

<sup>1</sup> COSTS and TIMING options are on by default

<sup>2</sup> I/O Timings shown when track\_io\_timing is enabled



## 6 Planner have to guess

Seq Scan on public.t1 (cost=0.00..104424.80 rows=10000000 width=8)

- startup cost
- total cost
- rows
- average row width



## 7 Cost stability principles

Quote from “Common issues with planner statistics by Tomas Vondra”: <sup>3</sup>

- **correlation to query duration:** The estimated cost is correlated with duration of the query, i.e. higher cost means longer execution.
- **estimation stability:** A small difference in estimation causes only small difference in costs, i.e. small error in estimation causes only small cost differences.
- **cost stability:** Small cost difference means small difference in duration.
- **cost comparability:** For a given query, two plans with (almost) the same costs should result in (almost) the same duration.

---

<sup>3</sup><https://blog.pgaddict.com/posts/common-issues-with-planner-statistics>



## 8 Data retrieval methods

- seq scan – sequential scan of whole table
- index scan – random io (read index + read table)
- index only scan – read only index (9.2+)<sup>4</sup>
- bitmap index scan – something in between seq scan/index scan, possible to use several indexes at same time in OR/AND conditions

---

<sup>4</sup>[https://wiki.postgresql.org/wiki/Index-only\\_scans](https://wiki.postgresql.org/wiki/Index-only_scans)





## 9 Join methods

- nested loop – optimal for small relations
- hash join – optimal for big relations
- merge join – optimal for big relations if they're sorted



## 10 Aggregate methods

- aggregate
- hash aggregate
- group aggregate



## 11 Planner Cost Constants

```
#seq_page_cost = 1.0 # cost of a sequentially-fetched disk page
#random_page_cost = 4.0 # cost of a non-sequentially-fetched disk page
#cpu_tuple_cost = 0.01 # cost of processing each row during a query
#cpu_index_tuple_cost = 0.005 # cost of processing each index entry
#cpu_operator_cost = 0.0025 # cost of processing each operator or function
```

so basically cost is just  $\sum_i c_i n_i$  . **How hard could it be?**



## 12 Well, kind of hard

- How many rows we'll get when we'll filter table by this condition?
- How many pages is that? Will we read them sequentially or not?
- How many rows we'll get when we join 2 relations?



## 13 We have stats!

- pg\_statistic
- pg\_stats view – the same but human-readable



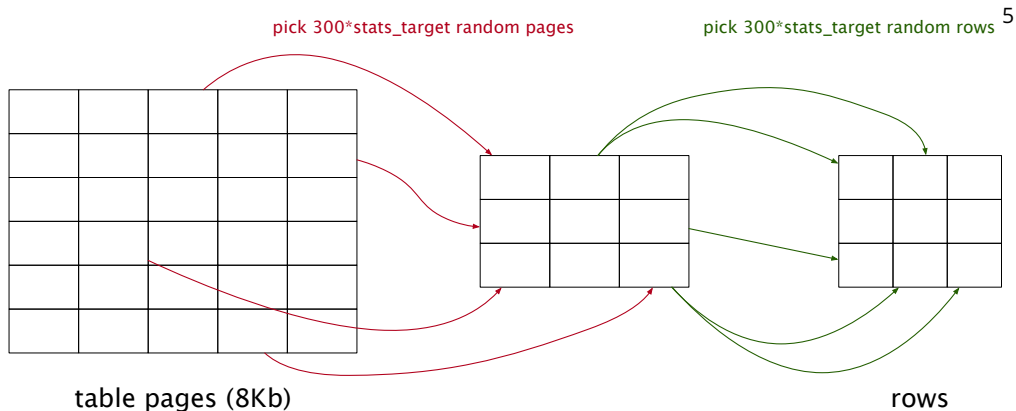
## 14 pg\_stats

```
pgday=# \d pg_stats
```

Column	Type	
tablename	name	name of the table or functional index
attname	name	name of the column or index column
null_frac	real	fraction of column entries that are null
avg_width	integer	average width in bytes of column's entries
n_distinct	real	number (or fraction of number of rows) of distinct values
most_common_vals	anyarray	list of the most common values in the column
most_common_freqs	real[]	list of the frequencies of the most common values
histogram_bounds	anyarray	list of intervals with approximately equal population
correlation	real	correlation between physical row ordering and logical ordering
most_common_elems	anyarray	
most_common_elem_freqs	real[]	
elem_count_histogram	real[]	

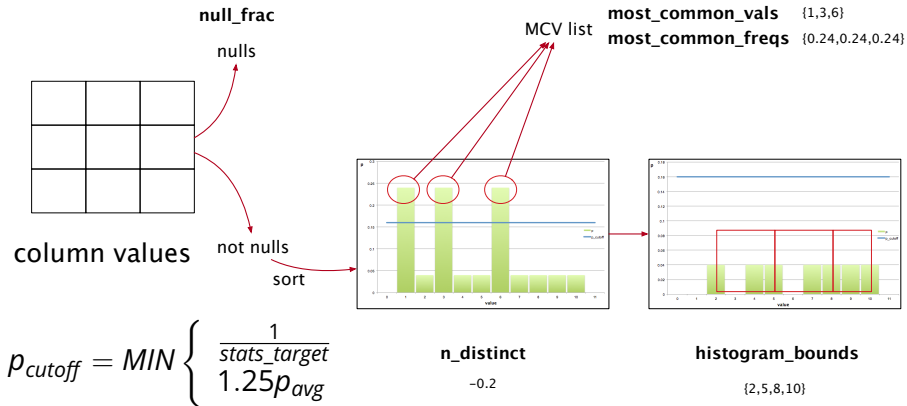


## 15 Analyze



<sup>5</sup>Algorithm Z from Vitter, Jeffrey S. (1 March 1985). "Random sampling with a reservoir"

## 16 Analyze





- $\text{inserted} + \text{updated} + \text{deleted} > \text{threshold} \Rightarrow \text{run autoanalyze}$
- $\text{threshold} = \text{autovacuum\_analyze\_threshold} + \text{reltuples} * \text{autovacuum\_analyze\_scale\_factor}$
- $\text{autovacuum\_analyze\_scale\_factor}$  (default = 0.1)
- $\text{autovacuum\_analyze\_threshold}$  (default = 50)
- $\text{default\_statistics\_target}$  (default = 100)
- $\text{rows in sample} = 300 * \text{stats\_target}$



## 18 n\_distinct underestimation example

```
select setseed(0.5);

create table test_ndistinct as
select
(case when random() < 0.1 then f1 end)::int f1
from normal_rand(10000000, 50000, 50000/3) as nr(f1);
```

**10M rows, 90% nulls,  $\approx$  99.7% of values in between 0..100000**



## 19 n\_distinct underestimation example

```
# analyze verbose test_ndistinct;
INFO:  analyzing "public.test_ndistinct"
INFO:  "test_ndistinct": scanned 30000 of 35314 pages, containing 8495268 live rows and 0 dead rows;
30000 rows in sample, 10000067 estimated total rows
```

```
select * from pg_stats where tablename = 'test_ndistinct' and attname = 'f1';
```

```
...
null_frac          | 0.904067
avg_width          | 4
n_distinct         | 3080
most_common_vals   |
most_common_freqs  |
histogram_bounds   | {-8505,10072,15513,18933,21260,22574,24082,25695,26953,27898,28645...
correlation        | -0.00286606
```



## 20 n\_distinct underestimation example

```
# explain analyze select distinct f1 from test_ndistinct ;
```

```
QUERY PLAN
```

```
-----  
HashAggregate  (cost=160314.84..160345.64 rows=3080 width=4)
```

```
    (actual time=2558.751..2581.286 rows=90020 loops=1)
```

```
    Group Key: f1
```

```
    -> Seq Scan on test_ndistinct  (cost=0.00..135314.67 rows=10000067 width=4)
```

```
        (actual time=0.045..931.687 rows=10000000 loops=1)
```

```
Planning time: 0.048 ms
```

```
Execution time: 2586.550 ms
```



## 21 n\_distinct underestimation example

```
# set default_statistics_target = 50;
# analyze verbose test_ndistinct;
INFO: analyzing "public.test_ndistinct"
INFO: "test_ndistinct": scanned 15000 of 35314 pages, containing 4247361 live rows and 0 dead rows;
15000 rows in sample, 9999792 estimated total rows
# explain analyze select distinct f1 from test_ndistinct ;
```

### QUERY PLAN

```
-----
HashAggregate  (cost=160311.40..160328.51 rows=1711 width=4)
               (actual time=2436.392..2455.851 rows=90020 loops=1)
  Group Key: f1
    -> Seq Scan on test_ndistinct  (cost=0.00..135311.92 rows=9999792 width=4)
                                   (actual time=0.029..892.596 rows=10000000 loops=1)

Planning time: 0.096 ms
Execution time: 2461.160 ms
```



## 22 n\_distinct underestimation example

```
# explain analyze select * from test_ndistinct where f1 < 5000;
```

QUERY PLAN

-----  
Seq Scan on test\_ndistinct (cost=0.00..160316.36 rows=99 width=4)

(actual time=2.325..1436.792 rows=3480 loops=1)

Filter: (f1 < 5000)

Rows Removed by Filter: 9996520

Planning time: 0.058 ms

Execution time: 1437.424 ms



## 23 n\_distinct underestimation example

```
alter table test_ndistinct alter column f1 set (n_distinct = 100000);  
analyze verbose test_ndistinct;  
INFO:  analyzing "public.test_ndistinct"  
INFO:  "test_ndistinct": scanned 15000 of 35314 pages, containing 4247670 live rows and 0 dead rows;  
15000 rows in sample, 10000012 estimated total rows  
ANALYZE
```



## 24 n\_distinct underestimation example

```
# explain analyze select distinct f1 from test_ndistinct ;
```

```
QUERY PLAN
```

```
-----  
Unique  (cost=1571431.43..1621431.49 rows=100000 width=4)
```

```
  (actual time=4791.872..7551.150 rows=90020 loops=1)
```

```
    -> Sort  (cost=1571431.43..1596431.46 rows=10000012 width=4)
```

```
      (actual time=4791.870..6893.413 rows=10000000 loops=1)
```

```
        Sort Key: f1
```

```
        Sort Method: external merge  Disk: 101648kB
```

```
          -> Seq Scan on test_ndistinct  (cost=0.00..135314.12 rows=10000012 width=4)
```

```
            (actual time=0.041..938.093 rows=10000000 loops=1)
```

```
Planning time: 0.099 ms
```

```
Execution time: 7714.701 ms
```





## 25 n\_distinct underestimation example

```
set work_mem = '8MB';
```

```
SET
```

```
# explain analyze select distinct f1 from test_ndistinct ;
```

```
QUERY PLAN
```

```
-----  
HashAggregate  (cost=160314.15..161314.15 rows=100000 width=4)
```

```
    (actual time=2371.902..2391.415 rows=90020 loops=1)
```

```
    Group Key: f1
```

```
    -> Seq Scan on test_ndistinct  (cost=0.00..135314.12 rows=10000012 width=4)
```

```
        (actual time=0.093..871.619 rows=10000000 loops=1)
```

```
Planning time: 0.048 ms
```

```
Execution time: 2396.186 ms
```



## 26 n\_distinct underestimation example

```
# explain analyze select * from test_ndistinct where f1 < 5000;
```

QUERY PLAN

-----  
Seq Scan on test\_ndistinct (cost=0.00..160316.61 rows=7550 width=4)  
(actual time=0.723..839.347 rows=3480 loops=1)

Filter: (f1 < 5000)

Rows Removed by Filter: 9996520

Planning time: 0.262 ms

Execution time: 839.774 ms



- n\_distinct plays important role in rows estimation when values are not in MCV list
- In very big tables it's possible to underestimate it in some cases
- It's possible to override n\_distinct estimation via  
**alter table xx alter column yy set (n\_distinct = zz);**



- Increasing default\_statistics\_target setting in config could help in this case but not recommended
- Default value 100 usually is good enough
- When it's not enough better increase it on selected columns only via **alter table xx alter column yy set statistics zz;**
- Otherwise it could lead to much longer planning time (autoanalyze will work longer too)



## 29 high default\_statistics\_target real example

```
# show default_statistics_target ;
```

```
default_statistics_target
```

```
-----
```

```
6000
```

```
explain analyze SELECT "seven_charlie"."id" FROM "xray" JOIN "seven_charlie" ON  
( "xray"."lima_seven" = "seven_charlie"."lima_seven" ) WHERE  
( "xray"."alpha" = '139505' ) AND ( "seven_charlie"."seven_five" IS TRUE );
```

```
Nested Loop (cost=0.850..798.110 rows=58 width=4) (actual time=0.081..3.314 rows=169 loops=1)
```

```
-> Index Scan using romeo on xray (cost=0.420..205.390 rows=242 width=4) (actual time=0.023..0.798 rows=169 loops=1)
```

```
Index Cond: (alpha = 139505)
```

```
-> Index Scan using lima_two on seven_charlie (cost=0.430..2.440 rows=1 width=8) (actual time=0.058..0.058 rows=1 loops=1)
```

```
Index Cond: ((lima_seven = papa3six.lima_seven) AND (seven_five = true))
```

```
Planning time: 433.630 ms
```

```
Execution time: 3.397 ms
```



## 30 high default\_statistics\_target real example

```
set default_statistics_target = 1000;
SET
# analyze verbose xray;
INFO:  analyzing "public.xray"
INFO:  "xray": scanned 6760 of 6760 pages, containing 851656 live rows and 2004 dead rows; 300000 rows
ANALYZE

Nested Loop  (cost=0.850..782.110 rows=57 width=4) (actual time=0.066..2.992 rows=169 loops=1)
->  Index Scan using romeo on xray  (cost=0.420..199.220 rows=238 width=4) (actual time=0.021..0.700 rows=169 loops=1)
      Index Cond: (alpha = 139505)
->  Index Scan using lima_two on seven_charlie  (cost=0.430..2.440 rows=1 width=8) (actual time=0.000..0.000 rows=1 loops=1)
      Index Cond: ((lima_seven = papa3six.lima_seven) AND (seven_five = true))

Planning time: 75.196 ms
Execution time: 3.071 ms
```



## 31 high default\_statistics\_target real example

```
# analyze verbose seven_charlie;
```

```
INFO: "seven_charlie": scanned 300000 of 1548230 pages, containing 2184079 live rows and 8293 dead r
```

```
Nested Loop (cost=0.850..782.110 rows=65 width=4) (actual time=0.197..26.517 rows=169 loops=1)
```

```
-> Index Scan using romeo on xray (cost=0.420..199.220 rows=238 width=4) (actual time=0.064..3.15
```

```
Index Cond: (alpha = 139505)
```

```
-> Index Scan using lima_two on seven_charlie (cost=0.430..2.440 rows=1 width=8) (actual time=0.0
```

```
Index Cond: ((lima_seven = papa3six.lima_seven) AND (seven_five = true))
```

```
Planning time: 14.256 ms
```

```
Execution time: 26.617 ms
```



```
select name from pg_settings where name ~ 'enable_';
```

```
enable_bitmapscan
```

```
enable_indexscan
```

```
enable_indexonlyscan
```

```
enable_seqscan
```

```
enable_tidscan
```

```
enable_nestloop
```

```
enable_hashjoin
```

```
enable_mergejoin
```

```
enable_sort
```

```
enable_hashagg
```

```
enable_material
```

```
startup_cost += disable_cost
```

```
disable_cost = 1010
```





- Very good for testing
- Affects whole query
- Possible to use in functions in some bad cases via **alter function xxx() set enable\_? = false**
- pg\_hint\_plan<sup>6</sup> extension (not in contrib) which provide hints

---

<sup>6</sup><https://osdn.net/projects/pghintplan/>



## 34 Join ordering problem

- There are  $O(n!)$  ways to join  $n$  relations which grows very fast ( $10! \approx 3.6M$ )
- ORMs like to join everything
- It's possible to break this number down by using CTEs but be careful
- `join_collapse_limit`, `from_collapse_limit` (default 8 relations)



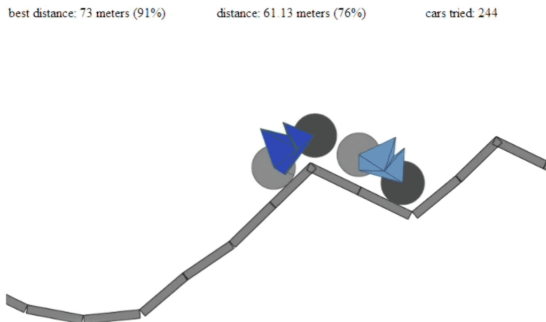
## 35 Genetic Query Optimizer (geqo)

- geqo\_threshold (default 12 relations)
- Chose suboptimal plan in reasonable time
- “Mutation” and selection phases



## 36 Genetic algorithm fun demo

7



---

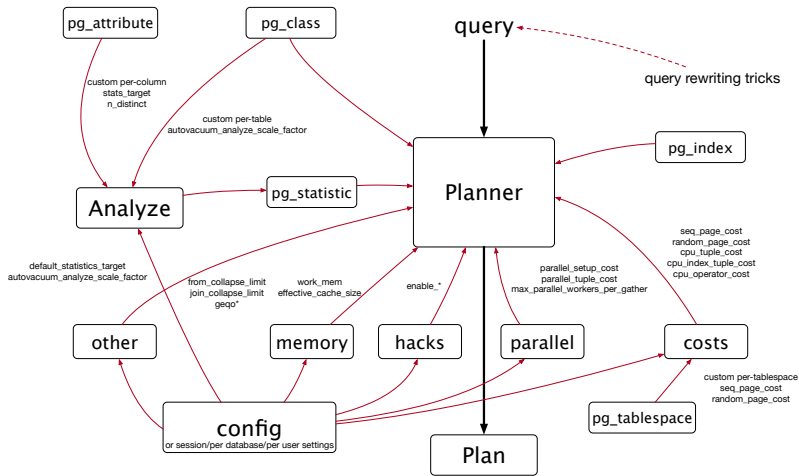
<sup>7</sup><http://boxcar2d.com/>

## 37 Query rewriting tricks

- Disable index usage in where clause:  $a = x \Rightarrow a+0 = x$
- Disable index usage in order by clause: order by  $a \Rightarrow$  order by  $a+0$
- Restrict push up/pull downs from subquery with offset 0
- Move non-limiting join after limit



## 38 What have we learned?



## 39 Troubleshooting

- Don't panic!
- Check if planner's estimates are wrong (off by orders of magnitude)
- Check for missing indexes when a lot of filtering is done
- For complex plans <https://explain.depesz.com/> could help
- Extract problem part
- Check for outdated/incomplete stats
- Play with hacks and query rewriting tricks



## 40 Would you like to know more?

- Robert Haas – The PostgreSQL Query Planner, PostgreSQL East 2010
- Tom Lane – Hacking the Query Planner, PGCon 2011
- Bruce Momjian – Explaining the Postgres Query Optimizer
- PostgreSQL Manual 67.1. Row Estimation Examples
- PostgreSQL Manual 14.1. Using EXPLAIN
- depesz: Implement multivariate n-distinct coefficients
- depesz: Explaining the unexplainable
- [www.slideshare.net/alexius2/](http://www.slideshare.net/alexius2/)





## 41 Questions?

[alexey.ermakov@dataegret.com](mailto:alexey.ermakov@dataegret.com)

