

# Explaining Explain

by Robert Treat

(Inspired by Greg Sabino-Mullane)

# Explain

- Explain shows the “best” plan as determined by the planner
- Shows each step that will be used to determine a query
- Can only be used on DML commands
- Shows the width, rows, and costs
- It is only an estimate, to see the real costs, you need to use `EXPLAIN ANALYZE`

# An Example Explain Plan

oscon=# explain select \* from pg\_proc order by proname;

QUERY PLAN

-----  
Sort (cost=181.55..185.92 rows=1747 width=322)

Sort Key: proname

-> Seq Scan on pg\_proc (cost=0.00..87.47 rows=1747 width=322)

# Things to keep in Mind

- Terms to know: plan, node, operators, filter, input set
- All information comes from Scan or Result operators, which are then passed upwards
- Each operator takes in input set and passes it up until the top node is reached
- Parents assume their children's costs
- InitPlans and Subplans are used for subselects

# Explaining --> Widths

oscon=# explain select oid from pg\_proc;

QUERY PLAN

-----  
Seq Scan on pg\_proc (cost=0.00..87.47 rows=1747 **width=4**)  
(1 row)

- Shows the estimated bytes in the input set at that level
- Not terribly important

## Explain Widths of Common Datatypes

|                         |     |
|-------------------------|-----|
| smallint                | 2   |
| integer                 | 4   |
| bigint                  | 8   |
| boolean                 | 1   |
| varchar(n) / char(n)    | n+4 |
| varchar() / text        | 32  |
| timestamp / timestamptz | 8   |

# Explaining --> Rows

```
oscon=# explain select oid from pg_proc;
```

```
QUERY PLAN
```

```
-----
```

```
Seq Scan on pg_proc (cost=0.00..87.47 rows=1747 width=4)
```

- Shows the estimated number of rows
- Default value is 1000 for tables that have never been vacuumed or analyzed
- Large discrepancies are a sign you need to vacuum and/or analyze

# Explaining --> Cost

```
oscon=# explain select oid from pg_proc;
```

```
QUERY PLAN
```

```
-----
```

```
Seq Scan on pg_proc (cost=0.00..87.47 rows=1747 width=4)
```

- Costs are what drive the optimizer to pick one plan over another
- Two costs: startup cost and total cost
- Difference is the total run cost
- Some operators have startup costs, some don't
- Costs are only estimates, based on (sometimes) complex formulas
- Arbitrary number, related to page fetch (1.0) and CPU usage

# Explaining --> Cost

oscon=# explain select oid from pg\_proc;

QUERY PLAN

-----  
Seq Scan on pg\_proc (cost=0.00..87.47 rows=1747 width=4)

## Cost Parameters

| Parameter            | Description                                             | Default | vs. page read |
|----------------------|---------------------------------------------------------|---------|---------------|
| page read            | Cost to read fetch one page of data, by definition      | 1.00    | -             |
| cpu_tuple_cost       | Cost of typical CPU time to process a tuple             | 0.01    | 100x quicker  |
| cpu_index_tuple_cost | Cost of typical CPU time to process an index tuple      | 0.001   | 1000x quicker |
| cpu_operator_cost    | Cost of CPU time to process a typical WHERE operator    | 0.0025  | 400x quicker  |
| random_page_cost     | Cost of a non-sequential page fetch                     | 4       | 4x slower     |
| effective_cache_size | Amount of cache – likelihood of finding a page in cache | 1000    | N/A           |



# Explaining --> Explain Analyze

```
oscon=# explain analyze select oid from pg_proc;  
                QUERY PLAN
```

```
-----  
Seq Scan on pg_proc (cost=0.00..87.47 rows=1747 width=4) (actual time=0.077..17.082 rows=1747 loops=1)  
Total runtime: 20.125 ms  
(2 rows)  
Time: 24.107 ms
```

- Actually runs the query and returns real information
- Time in milliseconds, NOT related to EXPLAIN “cost”
- Also gives the total running time
- Loops is the number the operator is run: multiply both times and rows by the number of loops for the actual numbers

# Explaining --> Plan Operators

| Operator            | Associated With                               | Startup cost?          |
|---------------------|-----------------------------------------------|------------------------|
| Seq Scan            | almost anything                               | No                     |
| Sort                | ORDER BY, other operators (e.g. Unique)       | Yes                    |
| Index Scan          | any non-hash index                            | No                     |
| Result              | non-table queries, WHERE constants            | No                     |
| Unique              | DISTINCT, UNION                               | Yes                    |
| Limit               | LIMIT, OFFSET                                 | Yes<br>(if OFFSET > 0) |
| Aggregate           | COUNT, SUM, MIN, MAX, AVG, STDDEV, VARIANCE   | Yes                    |
| Group               | GROUP BY clause                               | Yes                    |
| Append              | UNION, inheritance                            | No                     |
| Nested Loop         | INNER JOIN, LEFT OUTER JOIN                   | No                     |
| Merge Join          | INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN | Yes                    |
| Hash                | Hash Join operator                            | Yes                    |
| Hash Join           | INNER JOIN, LEFT OUTER JOIN                   | Yes                    |
| Subquery Scan       | UNION                                         | No                     |
| Tid Scan            | "ctid =" in the query                         | No                     |
| Materialize         | subselects, other operators                   | Yes                    |
| Function Scan       | functions                                     | No                     |
| SetOp Intersect     | INTERSECT                                     | Yes                    |
| SetOp Intersect All | INTERSECT ALL                                 | Yes                    |
| SetOp Except        | EXCEPT                                        | Yes                    |
| SetOp Except All    | EXCEPT ALL                                    | Yes                    |

# Seq Scan Operator : Example

oscon=# explain select oid from pg\_proc;

QUERY PLAN

-----  
Seq Scan on pg\_proc (cost=0.00..87.47 rows=1747 width=4)

- Most basic, simply scans from start to end
- Checks each row for constraint as it goes
- Large tables prefer index scans
- Cost (no startup cost), rows (tuples), width (oid)
- Total Cost is 87.47 (?)

# Seq Scan Operator : Explanation

```
oscon=# select relpages, reltuples from pg_class where relname = 'pg_proc';
relpages | reltuples
```

```
-----+-----
      70 |    1747
```

- Estimates **only!**
  - Dead tuples, inserts (MVCC)

```
oscon=# select 87.47 - 70 as cpu_cost;
```

```
cpu_cost
```

```
-----
```

```
17.47
```



```
oscon=# show cpu_tuple_cost;
```

```
cpu_tuple_cost
```

```
-----
```

```
0.01
```



```
oscon=# select 1747 * 0.01 as cpu_cost;
```

```
cpu_cost
```

```
-----
```

```
17.47
```



```
17.47 + 70 = 87.47
```

# The cost of a WHERE clause

oscon=# explain select oid from pg\_proc where 1+1=2;

QUERY PLAN

-----  
Seq Scan on pg\_proc (cost=0.00..**87.47** rows=1747 width=4)

oscon=# explain select oid from pg\_proc where 1+1=3;

QUERY PLAN

-----  
Result (cost=0.00..**87.47** rows=1747 width=4)

One-Time Filter: false

-> Seq Scan on pg\_proc (cost=0.00..**87.47** rows=1747 width=4)

oscon=# explain select oid from pg\_proc where oid>0;

QUERY PLAN

-----  
Seq Scan on pg\_proc (cost=0.00..**91.84** rows=**583** width=4)

Filter: (oid > 0::oid)

# Seq Scan Formulas

|                                                |                                            |
|------------------------------------------------|--------------------------------------------|
| oscon=# select 0.01 + 0.0025 as cpu_per_tuple; | cpu per tuples =                           |
| -[ RECORD 1 ]--+                               | cpu_tuple_cost + restriction cost          |
| cpu_per_tuple   0.0125                         | restriction cost ~ (n * cpu_operator_cost) |

oscon=# select 0.0125 \* 1747 as total\_cpu\_cost; Therefore the total cpu cost will be:  
-[ RECORD 1 ]--+-  
total\_cpu\_cost | 21.8375  
cpu\_per\_tuple \* number of tuples

```
oscon=# select relpages*1.0 as fetch_cost from pg_class where relname='pg_proc';
-[ RECORD 1 ]--+-
fetch_cost      | 70.0
```

Fetch cost is simple

Total Cost = fetch cost + cpu cost: 21.8375 + 70.0 = **91.8375**

```
oscon=# explain select oid from pg_proc where oid<>0;
          QUERY PLAN

```

Seq Scan on pg\_proc (cost=0.00..91.84 rows=1746 width=4)  
Filter: (oid <> 0::oid)

It really is row independent

# Additional WHERE clause bits

oscon=# explain select oid from pg\_proc where oid<>0 and oid<100;

QUERY PLAN

-----  
Seq Scan on pg\_proc (cost=0.00..**96.20** rows=582 width=4)

Filter: ((oid <> 0::oid) AND (oid < 100::oid))

Adding a second conditional increases our cost because we have to account for the additional operator cost

oscon=# select (0.01+ 0.0025 + 0.0025) \* 1747 as cpu\_cost;

-[ RECORD 1 ]-----

cpu\_cost | 26.2050

Adding the cpu cost to our page fetch cost, we arrive at our total cost

oscon=# select 26.2050 + 70.00 as cpu\_cost;

-[ RECORD 1 ]-----

cpu\_cost | **96.20**50

# The Sort Operator

oscon=# explain select oid from pg\_proc order by oid;

QUERY PLAN

-----  
**Sort** (cost=181.55..185.92 rows=1747 width=4)

Sort Key: oid

-> Seq Scan on pg\_proc (cost=0.00..87.47 rows=1747 width=4)

- Explicit: ORDER BY clause
- Implicit: Unique and other operators
- Has startup cost: cannot return right away



# Index Scan

oscon=# explain select oid from pg\_proc where oid=1;

QUERY PLAN

-----  
Index Scan using pg\_proc\_oid\_index on pg\_proc (cost=0.00..5.99 rows=1 width=4)  
Index Cond: (oid = 1::oid)

- Lower cost usually makes it preferred, especially on large tables

# BitMap Scan

- New in 8.1
- Creates bitmap of relations in memory

# Result Operator

oscon=# explain select oid from pg\_proc where 1+1=3;

QUERY PLAN

-----  
**Result** (cost=0.00..87.47 rows=1747 width=4)

One-Time Filter: false

-> Seq Scan on pg\_proc (cost=0.00..87.47 rows=1747 width=4)

- Non-table queries
- Inside a WHERE clause ('true' vs. 'false')

# Unique Operator

oscon=# explain select distinct oid from pg\_proc;

QUERY PLAN

-----  
**Unique** (cost=181.55..190.29 rows=1747 width=4)

-> Sort (cost=181.55..185.92 rows=1747 width=4)

Sort Key: oid

-> Seq Scan on pg\_proc (cost=0.00..87.47 rows=1747 width=4)

- Removes duplicate values from the input set
- Does not change ordering, simply fails to pass on duplicate rows
- Incoming set must be ordered (will force a Sort if needed)
- Two “cpu operations” per tuple cost
- Used with DISTINCT and UNION

# Limit Operator

oscon=# explain select oid from pg\_proc limit 5;

QUERY PLAN

-----  
**Limit** (cost=0.00..0.25 rows=5 width=4)

-> Seq Scan on pg\_proc (cost=0.00..87.47 rows=1747 width=4)

- Rows will be equal to number specified
- Can return first row immediately
- Also handles offsets, with a small additional startup cost

oscon=# explain select oid from pg\_proc limit 5 offset 5;

QUERY PLAN

-----  
**Limit** (cost=**0.25**..0.50 rows=5 width=4)

-> Seq Scan on pg\_proc (cost=0.00..87.47 rows=1747 width=4)

# Aggregate Operator

```
oscon=# explain select count(*) from pg_proc;
```

```
QUERY PLAN
```

```
-----  
Aggregate (cost=91.84..91.84 rows=1 width=0)
```

```
-> Seq Scan on pg_proc (cost=0.00..87.47 rows=1747 width=0)
```

- Used with count, sum, min, max, avg, stddev, variance
- You may see differences when GROUP BY is used

```
oscon=# explain select count(oid), oid from pg_proc group by oid;
```

```
QUERY PLAN
```

```
-----  
HashAggregate (cost=96.20..100.57 rows=1747 width=4)
```

```
-> Seq Scan on pg_proc (cost=0.00..87.47 rows=1747 width=4)
```

# GroupAggregate Operator

oscon=# explain select count(\*) from pg\_foo group by oid;  
QUERY PLAN

---

**GroupAggregate** (cost=37442.53..39789.07 rows=234654 width=4)  
-> Sort (cost=37442.53..38029.16 rows=234654 width=4)  
    Sort Key: oid  
        -> Seq Scan on pg\_foo (cost=0.00..13520.54 rows=234654 width=4)

- Used with GROUP BY and some aggregates on larger result sets

# Append Operator

oscon=# explain select oid from pg\_proc union all select oid from pg\_proc;  
QUERY PLAN

---

Append (cost=0.00..209.88 rows=3494 width=4)

- > Subquery Scan "\*SELECT\* 1" (cost=0.00..104.94 rows=1747 width=4)
  - > Seq Scan on pg\_proc (cost=0.00..87.47 rows=1747 width=4)
- > Subquery Scan "\*SELECT\* 2" (cost=0.00..104.94 rows=1747 width=4)
  - > Seq Scan on pg\_proc (cost=0.00..87.47 rows=1747 width=4)

- Triggered by UNION (ALL), inheritance
- No startup cost
- Cost is simply the sum of all inputs



# Nested Loop Operator

```
oscon=# pg_namespace on (pg_foo.pronamespace=pg_namespace.oid);  
QUERY PLAN
```

---

**Nested Loop** (cost=1.05..39920.17 rows=5867 width=68)

Join Filter: ("outer".pronamespace = "inner".oid)

-> Seq Scan on pg\_foo (cost=0.00..13520.54 rows=234654 width=68)

-> Materialize (cost=1.05..1.10 rows=5 width=4)

-> Seq Scan on pg\_namespace (cost=0.00..1.05 rows=5 width=4)

- Joins two tables (two input sets)
- USED with INNER JOIN and LEFT OUTER JOIN
- Scans 'outer' table, finds matches in 'inner' table
- No startup cost
- Can lead to slow queries, especially when functions are in the select clause

# Merge Joins

```
oscon=# explain select relname,nspname from pg_class left join  
oscon=# pg_namespace on (pg_class.relnamespace = pg_namespace.oid);  
QUERY PLAN
```

---

**Merge Right Join** (cost=14.98..17.79 rows=186 width=128)

Merge Cond: ("outer".oid = "inner".relnamespace)

-> Sort (cost=1.11..1.12 rows=5 width=68)

Sort Key: pg\_namespace.oid

-> Seq Scan on pg\_namespace (cost=0.00..1.05 rows=5 width=68)

-> Sort (cost=13.87..14.34 rows=186 width=68)

Sort Key: pg\_class.relnamespace

-> Seq Scan on pg\_class (cost=0.00..6.86 rows=186 width=68)

- Joins two sets: outer and an inner
- Merge Right Joins, Merge In Joins
- Sets must be pre-ordered (sorts), walk through both simultaneously

# Hash & Hash Join

- Hash creates a hash table for use by different hash join operators

oscon=# explain select relname, nspname from pg\_class join

oscon=# pg\_namespace on (pg\_class.relnamespace=pg\_namespace.oid);

QUERY PLAN

---

**Hash Join** (cost=1.06..10.71 rows=186 width=128)

Hash Cond: ("outer".relnamespace = "inner".oid)

-> Seq Scan on pg\_class (cost=0.00..6.86 rows=186 width=68)

-> **Hash** (cost=1.05..1.05 rows=5 width=68)

-> Seq Scan on pg\_namespace (cost=0.00..1.05 rows=5 width=68)

- Compares two input sets by building hash tables
- Used with INNER JOIN
- Creating a hash incurs startup cost
- Generally more efficient than Nested Loop

# Hash & Hash Left Join

```
oscon=# explain select relname, nspname from pg_class left join  
oscon-# pg_namespace on (pg_class.relnamespace=pg_namespace.oid);  
QUERY PLAN
```

---

**Hash Left Join** (cost=1.06..10.71 rows=186 width=128)

Hash Cond: ("outer".relnamespace = "inner".oid)

-> Seq Scan on pg\_class (cost=0.00..6.86 rows=186 width=68)

-> **Hash** (cost=1.05..1.05 rows=5 width=68)

-> Seq Scan on pg\_namespace (cost=0.00..1.05 rows=5 width=68)

- Similar to HOSH / HASH JOIN
- Incurs a startup cost
- Used with LEFT JOIN

# Subquery Scan

```
oscon=# explain select oid from pg_proc union all select oid from pg_proc;  
QUERY PLAN
```

---

Append (cost=0.00..209.88 rows=3494 width=4)

- > Subquery Scan "\*SELECT\* 1" (cost=0.00..104.94 rows=1747 width=4)
  - > Seq Scan on pg\_proc (cost=0.00..87.47 rows=1747 width=4)
- > Subquery Scan "\*SELECT\* 2" (cost=0.00..104.94 rows=1747 width=4)
  - > Seq Scan on pg\_proc (cost=0.00..87.47 rows=1747 width=4)

- Used with unions
- Generally not a significant problem

# Tid Scan

oscon=# explain select oid from pg\_proc where ctid = '(0,1)';

QUERY PLAN

-----

Tid Scan on pg\_proc (cost=0.00..4.01 rows=1 width=4)

Filter: (ctid = '(0,1)::tid)

- Column tuple ID
- Only used when “ctid=” appears in your query
- Very rare, very fast

# Materialize Operator

# Function Scan

```
oscon=# create or replace function foo(integer) returns setof integer as '  
oscon'# select $1;  
oscon'# ' language 'sql';  
CREATE FUNCTION  
oscon=# explain select * from foo(12);  
                QUERY PLAN
```

-----  
**Function Scan** on foo (cost=0.00..12.50 rows=1000 width=4)

- Seen when a function is gathering data
- Somewhat mysterious for troubleshooting
- Run explain on queries used inside function



# SetOp Operators

- SetOp Intersect
- SetOp Intersect All

- SetOp Except
- SetOp Except All

oscon=# explain select oid from pg\_proc INTERSECT select oid from pg\_proc;  
QUERY PLAN

---

**SetOp Intersect** (cost=415.51..432.98 rows=349 width=4)

-> Sort (cost=415.51..424.25 rows=3494 width=4)

Sort Key: oid

-> Append (cost=0.00..209.88 rows=3494 width=4)

-> Subquery Scan "\*SELECT\* 1" (cost=0.00..104.94 rows=1747 width=4)

-> Seq Scan on pg\_proc (cost=0.00..87.47 rows=1747 width=4)

-> Subquery Scan "\*SELECT\* 2" (cost=0.00..104.94 rows=1747 width=4)

-> Seq Scan on pg\_proc (cost=0.00..87.47 rows=1747 width=4)

- Used (obviously) for INTERSECT, INTERSECT ALL, EXCEPT, EXCEPT ALL

# Forcing a plan

- You can “strongly discourage” the planner from attempting certain operations
- `SET enable_seqscan = false;`
- True / False
  - ♦ `enable_seqscan`
  - ♦ `enable_indexscan`
  - ♦ `enable_sort`
  - ♦ `enable_nestloop`
  - ♦ `enable_hashjoin`
  - ♦ `enable_mergejoin`
  - ♦ `enable_tidscan`
  - ♦ `enable_hasagg`

# Forcing a Seq Scan I

/src/backend/optimizer/path/costsize.c

```
Cost          disable_cost = 1000000000.0;
```

```
<snip>
```

```
    if (!enable_seqscan)
        startup_cost += disable_cost;
```

```
oscon=# explain select * from pg_class;
```

```
        QUERY PLAN
```

```
-----  
Seq Scan on pg_class (cost=1000000000.00..1000000006.86 rows=186  
width=164)
```

# Force a Scan, Change a Plan

```
oscon=# explain analyze select * from pg_class where oid > 2112;  
QUERY PLAN
```

---

Seq Scan on pg\_class (cost=0.00..7.33 rows=62 width=164)  
(actual time=0.087..**1.700** rows=174 loops=1)

Filter: (oid > 2112::oid)

Total runtime: 2.413 ms

```
oscon=# set enable_seqscan = false;
```

```
SET
```

```
oscon=# explain analyze select * from pg_class where oid > 2112;  
QUERY PLAN
```

---

Index Scan using pg\_class\_oid\_index on pg\_class (cost=0.00..22.84 rows=62  
width=164) (actual time=0.144..**1.802** rows=174 loops=1)

Index Cond: (oid > 2112::oid)

Total runtime: 2.653 ms

# Things to Keep in Mind

- Forcing Plans: Good for development, Bad for production
- You are not smarter than the planner (well, maybe Tom)
- OTOH, the planner is just guessing
- Use explain analyze whenever possible

# Real World Debugging

- Incorrect row estimates can mean you need to analyze

ms=# explain analyze select exception\_id from exception join exception\_notice\_map using (exception\_id) where complete IS FALSE and notice\_id = 3;

QUERY PLAN

---

Nested Loop (cost=0.00..2654.65 rows=199 width=8) (actual time=151.16..538.45 rows=124 loops=1)  
-> Seq Scan on exception\_notice\_map (cost=0.00..250.50 rows=399 width=4) (actual time=0.10..101.61 rows=15181 loops=1)  
Filter: (notice\_id = 3)  
-> Index Scan using **exception\_pkey** on exception (cost=0.00..6.01 rows=1 width=4) (actual time=0.03..0.03 rows=0 loops=15181)  
Index Cond: (exception.exception\_id = "outer".exception\_id)  
Filter: (complete IS FALSE)  
Total runtime: 538.76 msec

- Partial index on exception “WHERE complete IS False”
- Only 251 rows where complete is false

# Real World Debugging

- Incorrect row estimates can mean you need to analyze

```
rms=# analyze exception;
```

```
ANALYZE
```

```
rms=# explain analyze select exception_id from exception join exception_notice_map using (exception_id) where  
complete IS FALSE and notice_id = 3;
```

QUERY PLAN

---

```
Hash Join (cost=28.48..280.98 rows=1 width=8) (actual time=31.45..97.78 rows=124 loops=1)  
  Hash Cond: ("outer".exception_id = "inner".exception_id)  
    -> Seq Scan on exception_notice_map (cost=0.00..250.50 rows=399 width=4) (actual time=0.12..77.12 rows=15181  
loops=1)  
      Filter: (notice_id = 3)  
    -> Hash (cost=26.31..26.31 rows=251 width=4) (actual time=2.96..2.96 rows=0 loops=1)  
      -> Index Scan using active_exceptions on exception (cost=0.00..26.31 rows=251 width=4) (actual time=0.24..2.55  
rows=251 loops=1)  
        Filter: (complete IS FALSE)  
Total runtime: 97.99 msec
```

- Uses our partial index on exception “WHERE complete IS False”
- Rows from index match what we know

# Real World Debugging

- Row estimates of 1000 are a dead giveaway
- So are nice, pretty, round numbers

oscon=# explain analyze select exception\_id from exception join exception\_notice\_map using (exception\_id) where complete IS FALSE and notice\_id = 3;

## QUERY PLAN

```
-----
Hash Join (cost=22.51..45.04 rows=2 width=8) (actual time=9961.14..11385.11 rows=105 loops=1)
  Hash Cond: ("outer".exception_id = "inner".exception_id)
    -> Seq Scan on exception (cost=0.00..20.00 rows=500 width=4) (actual time=365.12..10659.11 rows=228 loops=1)
      Filter: (complete IS FALSE)
    -> Hash (cost=22.50..22.50 rows=5 width=4) (actual time=723.39..723.39 rows=0 loops=1)
      -> Seq Scan on exception_notice_map (cost=0.00..22.50 rows=5 width=4) (actual time=10.12..694.57 rows=15271 loops=1)
        Filter: (notice_id = 3)
Total runtime: 11513.78 msec
```

- Since exception\_notice\_map is way off, let's analyze that first



# Real World Debugging

- Row estimates of 1000 are a dead giveaway
- So are nice, pretty, round numbers

```
oscon=# analyze exception_notice_map ;
```

```
ANALYZE
```

```
oscon=# explain analyze select exception_id from exception join exception_notice_map using (exception_id) where  
complete IS FALSE and notice_id = 3;
```

## QUERY PLAN

---

Merge Join (cost=42.41..802.93 rows=390 width=8) (actual time=10268.79..10898.29 rows=105 loops=1)

Merge Cond: ("outer".exception\_id = "inner".exception\_id)

-> **Index Scan** using exceptionnoticemap\_exceptionid on exception\_notice\_map (cost=0.00..714.22 **rows=15562**  
width=4) (actual time=50.80..1063.05 **rows=15271** loops=1)

Filter: (notice\_id = 3)

-> Sort (cost=42.41..43.66 rows=500 width=4) (actual time=9800.32..9800.65 rows=222 loops=1)

Sort Key: exception.exception\_id

-> Seq Scan on exception (cost=0.00..20.00 **rows=500** width=4) (actual time=357.18..9799.63 rows=228 loops=1)

Filter: (complete IS FALSE)

Total runtime: 10898.57 msec

- Better, but the 500 still looks too coincidental, let's analyze exception

# Real World Debugging

- Row estimates of 1000 are a dead give away
- So are nice, pretty, round numbers

```
oscon=# analyze exception;
```

```
ANALYZE
```

```
oscon=# explain analyze select exception_id from exception join exception_notice_map using (exception_id) where  
complete IS FALSE and notice_id = 3;
```

## QUERY PLAN

---

Merge Join (cost=0.00..796.57 rows=31 width=8) (actual time=425.41..971.81 rows=105 loops=1)

Merge Cond: ("outer".exception\_id = "inner".exception\_id)

-> **Index Scan** using active\_exceptions on exception (cost=0.00..41.86 **rows=651** width=4) (actual  
time=54.04..84.22 **rows=222** loops=1)

Filter: (complete IS FALSE)

-> **Index Scan** using exceptionnoticemap\_exceptionid on exception\_notice\_map (cost=0.00..714.22 rows=15562  
width=4) (actual time=34.42..843.10 rows=15271 loops=1)

Filter: (notice\_id = 3)

Total runtime: **972.05 msec**

- No more round numbers, all indexes used, much faster
- Still a black art: the row estimation got worse

# Real World Debugging

- Watch for high costs on ordinary operations, could be time to vacuum

```
oscon=# EXPLAIN ANALYZE SELECT s.site_id, s.name, i.image_name FROM images i
oscon=# JOIN host h USING (host_id) JOIN site s USING (site_id) WHERE images_id > 2112;
```

## QUERY PLAN

```
-----
Hash Join (cost=113.88..253.51 rows=534 width=53) (actual time=610.52..627.11 rows=534 loops=1)
  Hash Cond: ("outer".site_id = "inner".site_id)
    -> Seq Scan on site s (cost=0.00..73.74 rows=1974 width=34) (actual time=5.25..17.43 rows=1974 loops=1)
    -> Hash (cost=112.54..112.54 rows=534 width=19) (actual time=605.15..605.15 rows=0 loops=1)
      -> Hash Join (cost=15.01..112.54 rows=534 width=19) (actual time=590.89..604.06 rows=534 loops=1)
        Hash Cond: ("outer".host_id = "inner".host_id)
          -> Seq Scan on host h (cost=0.00..77.24 rows=2724 width=8) (actual time=567.99..581.30 rows=2724
loops=1)
          -> Hash (cost=13.68..13.68 rows=534 width=11) (actual time=17.30..17.30 rows=0 loops=1)
            -> Seq Scan on images i (cost=0.00..13.68 rows=534 width=11) (actual time=14.55..16.47 rows=534
loops=1)
        Filter: (images_id > 2112)
```

- Notice Host seq scan time is much larger than Site / Images

# Real World Debugging

- Watch for high costs on ordinary operations, could be time to vacuum

```
rms=# vacuum full verbose host;
INFO: --Relation public.host--
INFO: Pages 4785: Changed 0, reaped 4761, Empty 0, New 0; Tup 2724: Vac 0, Keep/VTL 0/0, UnUsed 267553,
MinLen 100, MaxLen 229; Re-using: Free/Avail. Space 37629760/37627880; EndEmpty/Avail. Pages 0/4751.
      CPU 0.30s/0.03u sec elapsed 0.32 sec.
INFO: Index host_pkey: Pages 1214; Tuples 2724: Deleted 0.
      CPU 0.07s/0.01u sec elapsed 0.08 sec.
INFO: Rel host: Pages: 4785 --> 50; Tuple(s) moved: 2724.
      CPU 0.52s/1.09u sec elapsed 1.66 sec.
INFO: Index host_pkey: Pages 1214; Tuples 2724: Deleted 2724.
      CPU 0.14s/0.00u sec elapsed 0.14 sec.
INFO: --Relation pg_toast.pg_toast_2124348104--
INFO: Pages 0: Changed 0, reaped 0, Empty 0, New 0; Tup 0: Vac 0, Keep/VTL 0/0, UnUsed 0, MinLen 0,
MaxLen 0; Re-using: Free/Avail. Space 0/0; EndEmpty/Avail. Pages 0/0.
      CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO: Index pg_toast_2124348104_index: Pages 1; Tuples 0.
      CPU 0.00s/0.00u sec elapsed 0.00 sec.
VACUUM
```

- Vacuum verbose shows us just how bloated it had become

# Real World Debugging

- Watch for high costs on ordinary operations, could be time to vacuum

```
oscon=# EXPLAIN ANALYZE SELECT s.site_id, s.name, i.image_name FROM images i
```

```
oscon=# JOIN host h USING (host_id) JOIN site s USING (site_id) WHERE images_id > 2112;
```

QUERY PLAN

-----  
Hash Join (cost=113.88..253.51 rows=534 width=53) (actual time=16.19..28.84 rows=534 loops=1)

Hash Cond: ("outer".site\_id = "inner".site\_id)

-> Seq Scan on site s (cost=0.00..73.74 rows=1974 width=34) (actual time=0.11..8.44 rows=1974 loops=1)

-> Hash (cost=112.54..112.54 rows=534 width=19) (actual time=15.93..15.93 rows=0 loops=1)

-> Hash Join (cost=15.01..112.54 rows=534 width=19) (actual time=3.11..14.94 rows=534 loops=1)

Hash Cond: ("outer".host\_id = "inner".host\_id)

-> Seq Scan on host h (cost=0.00..77.24 rows=2724 width=8) (actual time=0.01..**6.99** rows=2724 loops=1)

-> Hash (cost=13.68..13.68 rows=534 width=11) (actual time=2.94..2.94 rows=0 loops=1)

-> Seq Scan on imaging\_app ia (cost=0.00..13.68 rows=534 width=11) (actual time=0.14..2.09 rows=534

loops=1)

Filter: (imaging\_app\_id > 2112)

Total runtime: 29.39 msec

- We can now traverse the host table rather quickly

# Real World Debugging

- Watch out for functions, they can hide additional costs

```
oscon=# explain analyze SELECT site.name, host.name, get_license(host.host_id)
```

```
oscon=# FROM site site JOIN host USING (site_id) ;
```

```
QUERY PLAN
```

```
-----  
Hash Join (cost=243.29..745.50 rows=2745 width=55) (actual time=25.39..27999.14 rows=2745 loops=1)
```

```
Hash Cond: ("outer".site_id = "inner".site_id)
```

```
-> Seq Scan on host (cost=0.00..440.45 rows=2745 width=21) (actual time=0.02..41.03 rows=2745 loops=1)
```

```
-> Hash (cost=238.23..238.23 rows=2023 width=34) (actual time=19.98..19.98 rows=0 loops=1)
```

```
    -> Seq Scan on site (cost=0.00..238.23 rows=2023 width=34) (actual time=0.02..16.95 rows=2024  
loops=1)
```

```
Total runtime: 28007.81 msec
```

- Earlier steps have very small costs
- Hash join results in very large cost
- Nothing is self-evident

# Real World Debugging

- Watch out for functions, they can hide additional costs

```
oscon=# explain analyze SELECT site.name, host.name FROM site site JOIN host USING (site_id) ;  
QUERY PLAN
```

```
-----  
Hash Join (cost=243.89..746.10 rows=2745 width=51) (actual time=15.15..49.20 rows=2745 loops=1)  
  Hash Cond: ("outer".site_id = "inner".site_id)  
    -> Seq Scan on host (cost=0.00..440.45 rows=2745 width=17) (actual time=0.01..18.75 rows=2745 loops=1)  
    -> Hash (cost=238.23..238.23 rows=2023 width=34) (actual time=15.10..15.10 rows=0 loops=1)  
          -> Seq Scan on site (cost=0.00..238.23 rows=2023 width=34) (actual time=0.02..12.55 rows=2024 loops=1)  
Total runtime: 50.99 msec
```

- Removing the function reduces the query time considerably
- You will need to optimize the function

# Real World Debugging

- Watch out for functions, they can hide additional costs
- Explain Analyze will not show triggers costs separately in 8.1



# Real World Debugging

- A seq scan may be an opportunity for an index

```
Sort (cost=10249.87..10316.53 rows=26665 width=191) (actual time=1503.000..1553.000 rows=27363 loops=1)
  Sort Key: c.lwrp
  -> Hash Left Join (cost=5363.06..6246.11 rows=26665 width=191) (actual time=621.000..1172.000 rows=27363
loops=1)
    Hash Cond: ("outer".serverid = "inner".serverid)
    -> Merge Left Join (cost=5346.31..5762.72 rows=26665 width=76) (actual time=621.000..942.000 rows=27363
loops=1)
      Merge Cond: (("outer".serverid = "inner".serverid) AND ("outer".guildid = "inner".guildid))
      -> Sort (cost=4546.72..4613.38 rows=26665 width=58) (actual time=501.000..562.000 rows=27363 loops=1)
        Sort Key: c.serverid, c.guildid
        -> Index Scan using idx_server on "char" c (cost=0.00..1914.49 rows=26665 width=58) (actual
time=0.000..241.000 rows=27363 loops=1)
          Index Cond: (serverid = 5)
        -> Sort (cost=799.59..819.84 rows=8097 width=28) (actual time=110.000..160.000 rows=30507 loops=1)
          Sort Key: g.serverid, g.guildid
          -> Seq Scan on guild g (cost=0.00..273.97 rows=8097 width=28) (actual time=0.000..20.000 rows=8097
loops=1)
        -> Hash (cost=15.40..15.40 rows=540 width=117) (actual time=0.000..0.000 rows=0 loops=1)
          -> Seq Scan on server s (cost=0.00..15.40 rows=540 width=117) (actual time=0.000..0.000 rows=5 loops=1)
Total runtime: 1573.000 ms
```

- Guild should only returns a small number of rows, but we seq scan it

# Real World Debugging

- Of course, it is handy for query rewriting as well

```
oscon=# EXPLAIN ANALYZE SELECT count(*) FROM advertiser_contact
```

```
oscon=# JOIN advertiser USING (advertiser_id) WHERE type=1;
```

QUERY PLAN

---

```
Aggregate (cost=1.87..1.87 rows=1 width=0) (actual time=8.790..8.791 rows=1 loops=1)
```

```
-> Merge Join (cost=1.03..1.86 rows=2 width=0) (actual time=8.752..8.766 rows=2 loops=1)
```

```
  Merge Cond: ("outer".advertiser_id = "inner".advertiser_id)
```

```
    -> Index Scan using advertiser_id_pkey on advertiser (cost=0.00..2.11 rows=8 width=4) (actual  
time=8.627..8.650 rows=4 loops=1)
```

```
      Filter: ("type" = 1)
```

```
    -> Sort (cost=1.03..1.03 rows=2 width=4) (actual time=0.073..0.075 rows=2 loops=1)
```

```
      Sort Key: advertiser_contact.advertiser_id
```

```
    -> Seq Scan on advertiser_contact (cost=0.00..1.02 rows=2 width=4) (actual time=0.021..0.027  
rows=2 loops=1)
```

```
Total runtime: 8.978 ms
```

- This is the most straightforward way to write this query,  
could it be faster?

# Real World Debugging

- Of course, it is handy for query rewriting as well

```
oscon=# explain analyze select count(*) from advertiser_contact where advertiser_id IN
oscon=# (select advertiser_id from advertiser where type = 1);
```

## QUERY PLAN

```
-----
Aggregate (cost=2.23..2.23 rows=1 width=0) (actual time=0.261..0.261 rows=1 loops=1)
  -> Hash Join (cost=2.15..2.23 rows=2 width=0) (actual time=0.231..0.246 rows=2 loops=1)
    Hash Cond: ("outer".advertiser_id = "inner".advertiser_id)
    -> HashAggregate (cost=1.12..1.12 rows=8 width=4) (actual time=0.091..0.112 rows=8 loops=1)
      -> Seq Scan on advertiser (cost=0.00..1.10 rows=8 width=4) (actual time=0.051..0.068 rows=8
loops=1)
        Filter: ("type" = 1)
      -> Hash (cost=1.02..1.02 rows=2 width=4) (actual time=0.101..0.101 rows=0 loops=1)
        -> Seq Scan on advertiser_contact (cost=0.00..1.02 rows=2 width=4) (actual time=0.088..0.094
rows=2 loops=1)
Total runtime: 0.422 ms
```

- Rewriting this query did make it faster

# Real World Debugging

- Of course, it is handy for query rewriting as well

```
live=# EXPLAIN ANALYZE SELECT count(*) FROM advertiser_contact WHERE EXISTS (SELECT  
1 FROM advertiser WHERE advertiser_id=advertiser_contact.advertiser_id AND type = 1);
```

QUERY PLAN

```
-----  
Aggregate (cost=3.27..3.27 rows=1 width=0) (actual time=0.200..0.201 rows=1 loops=1)  
  -> Seq Scan on advertiser_contact (cost=0.00..3.26 rows=1 width=0) (actual time=0.162..0.179  
rows=2 loops=1)  
    Filter: (subplan)  
      SubPlan  
        -> Seq Scan on advertiser (cost=0.00..1.12 rows=1 width=0) (actual time=0.034..0.034 rows=1  
loops=2)  
          Filter: ((advertiser_id = $0) AND ("type" = 1))  
Total runtime: 0.333 ms
```

- And faster still...
- Multiple ways to approach a query
- Make sure to test again real world data scenarios

# Things to Keep in Mind

- Make sure you have Vacuumed and Analyzed your tables first
- Run queries twice or more per plan (caching)
- Verify explain output (count(\*) vs. rows)
- Experiment with indexes
- Use real world data (Slony)

# Asking for help

- Make sure you have done your own debugging first
- State your PostgreSQL version
- Make sure you have vacuumed and/or analyzed appropriately
- Always include explain analyze output
- Include queries/tables/data when possible

[pgsql-performance@postgresql.org](mailto:pgsql-performance@postgresql.org)

# Thanks

Greg Sabino-Mullane

AndrewSN@#postgresql

Magnifikus@#postgresql