# PostgreSQL Performance

# PostgreSQL Performance

- Configuration
- Explain
- Planner Statistics
- Indexes
- Understanding the Optimizer
- Tuning Example

# Performance Configuration

# Configuration

- Several configuration parameters have an affect on performance
- Most are tuned via formulas in the default parameter groups
- Checkpoint values should be set based on business rules

# shared_buffers

- Sets the primary cache for the server
- Corresponds to the number of shared memory buffers used by the database server.
- Each buffer is 8K bytes.
- Minimum value must be 16 and at least 2 x max_connections.

# work_mem

- Amount of memory in KB to be used by internal sorts and hash tables before switching to temporary disk files.

- Minimum allowed value is 64 KB.

- It is set in KB and the default is 1024 KB (1 MB).

# maintenance_work_mem

- Maximum memory in KB to be used in maintenance operations such as VACUUM, CREATE INDEX, and ALTER TABLE ADD FOREIGN KEY. Minimum allowed value is 1024 KB.

- It is set in KB and the default is 16384 KB (16 MB).

- Performance for vacuuming and restoring database dumps can be improved by increasing this value.

# wal_buffers

- Number of disk-page buffers allocated in shared memory for WAL data.
- Each buffer is 8K bytes
- Need only be large enough to hold the amount of WAL data created by a typical transaction
- Minimum allowed value is 32K.
- Default setting is 16MB.

# Explain

# EXPLAIN

- There are a number of different ways to execute a query
- These ways are called a plan
- More complex queries have more possible plans
- EXPLAIN returns the plan that PostgreSQL chose

# Explain EXPLAIN

```
 EXPLAIN SELECT * FROM cities;
                         QUERY PLAN
-------------------------------------------------------------
Seq Scan on cities  (cost=0.00..1230.11 rows=63211 width=43)
(1 row)
```

- Cost
  - First number is the cost for the first row
  - Second number is the cost for all rows
- Rows
  - Number of rows the planner thinks it will be returning
- Width
  - Each row is 43 bytes wide
- The planner has decided that it will be doing a sequential scan

# Explain

- Estimated start-up cost (time expended before output scan can start (e.g., time to do sorting in a sort node)

- Estimated total cost if all rows were to be retrieved (e.g., disregards effects of a LIMIT clause)

- Estimated # of rows output if executed to completion

- Estimated average width in bytes of rows output

- (Costs are in units of disk page fetches – CPU efforts are converted into disk-page units)

# EXPLAIN ANALYZE

```
EXPLAIN ANALYZE SELECT * FROM cities;
                                                  QUERY PLAN
---------------------------------------------------------------------
-----------------
 Seq Scan on cities  (cost=0.00..1230.11 rows=63211 width=43)
                     (actual time=0.008..5.494 rows=63211 loops=1)
 Planning Time: 0.041 ms
 Execution Time: 8.504 ms
(3 rows)
```

- The actual runtime of this query.
- Note: the estimate seems to have no correlation to the actual time
- The estimate is in an arbitrary unit which is “How long it takes sequentially read a single page from disk”

# Explain Analyze

- Actual time in milliseconds of real time (per loop if loops > 1)
- Number or rows output (per loop if loops > 1)
- Planning Time includes parsing, rewriting, or planning time
- Execution Time is executor start-up, shutdown, & processing time

# Explain with more than one step

```
EXPLAIN SELECT * FROM cities ORDER BY name;
                                QUERY PLAN
---------------------------------------------------------------------
 Sort  (cost=8216.52..8374.55 rows=63211 width=43)
   Sort Key: name
   ->  Seq Scan on cities  (cost=0.00..1230.11 rows=63211 width=43)
```

- Sort and Sequential scan
- Data flows from the lower steps to the higher steps
- Output of the seq scan is fed into the sort
- The key which is used for the sort is “name”
- First row of the sort is high (a sort can not return rows until it is done)
- Actual cost of the sort is (8216.52 - 1230.11 = 6986.41)
  - It needs all of the rows before it can sort

# Multiple levels

```
EXPLAIN ANALYZE SELECT * FROM episodes JOIN titles ON (parent_id = title_id);
                                                    QUERY PLAN
----------------------------------------------------------------------------------
--
Hash Join  (cost=267825.51..471037.50 rows=4033979 width=93)
           (actual time=2458.706..6010.320 rows=4033979 loops=1)
  Hash Cond: ((episodes.parent_id)::text = (titles.title_id)::text)
  ->  Seq Scan on episodes  (cost=0.00..69198.79 rows=4033979 width=26)
                            (actual time=0.007..411.083 rows=4033979 loops=1)
  ->  Hash  (cost=126734.67..126734.67 rows=5825667 width=67)
            (actual time=2455.520..2455.520 rows=5825667 loops=1)
        Buckets: 65536  Batches: 256  Memory Usage: 2679kB
        ->  Seq Scan on titles  (cost=0.00..126734.67 rows=5825667 width=67)
                                (actual time=0.007..758.389 rows=5825667 loops=1)
```

- Indentation is used to show what steps feed into the next step above
  - The Hash is fed from the Seq scan
- The hash join does not wait for the scan, it can start returning immediately

# Diagnosing performance issues

```
=> EXPLAIN ANALYZE SELECT * FROM game_changers;
                                                                            QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------
---
 GroupAggregate  (cost=546632.16..549163.19 rows=37497 width=22) (actual time=34264.743..37864.405 rows=2933 loops=1)
   Group Key: p.name
   Filter: (count(*) > 10)
   Rows Removed by Filter: 1180924
   ->  Sort  (cost=546632.16..546913.39 rows=112490 width=14) (actual time=34263.350..37271.095 rows=2110225 loops=1)
         Sort Key: p.name
         Sort Method: external merge  Disk: 50944kB
         ->  Nested Loop  (cost=177244.72..535269.57 rows=112490 width=14) (actual time=1533.309..25609.351 rows=2110225 loops=1)
               ->  Hash Join  (cost=177244.29..480421.40 rows=114485 width=10) (actual time=1533.177..6251.774 rows=2218756 loops=1)
                     Hash Cond: ((pt.title_id)::text = (r.title_id)::text)
                     ->  Seq Scan on people_titles pt  (cost=0.00..245737.43 rows=15011943 width=20) (actual time=0.007..1398.222 rows=15012000 loops=1)
                     ->  Hash  (cost=177155.74..177155.74 rows=7084 width=20) (actual time=1533.154..1533.154 rows=35358 loops=1)
                           Buckets: 65536 (originally 8192)  Batches: 1 (originally 1)  Memory Usage: 2308kB
                           ->  Hash Join  (cost=21082.40..177155.74 rows=7084 width=20) (actual time=172.764..1522.431 rows=35358 loops=1)
                                 Hash Cond: ((t.title_id)::text = (r.title_id)::text)
                                 ->  Hash Join  (cost=1.14..148509.24 rows=582567 width=10) (actual time=0.017..1184.387 rows=516421 loops=1)
                                       Hash Cond: (t.kind_id = k.kind_id)
                                       ->  Seq Scan on titles t  (cost=0.00..126734.67 rows=5825667 width=14) (actual time=0.005..538.236 rows=5825667
loops=1)
                                       ->  Hash  (cost=1.12..1.12 rows=1 width=2) (actual time=0.007..0.007 rows=1 loops=1)
                                             Buckets: 1024  Batches: 1  Memory Usage: 9kB
                                             ->  Seq Scan on kind k  (cost=0.00..1.12 rows=1 width=2) (actual time=0.004..0.005 rows=1 loops=1)
                                                   Filter: ((kind)::text = 'movie'::text)
                                                   Rows Removed by Filter: 9
                                 ->  Hash  (cost=19849.79..19849.79 rows=70837 width=10) (actual time=172.427..172.427 rows=70794 loops=1)
                                       Buckets: 131072  Batches: 2  Memory Usage: 2474kB
                                       ->  Seq Scan on ratings r  (cost=0.00..19849.79 rows=70837 width=10) (actual time=0.009..154.582 rows=70794 loops=1)
                                             Filter: ((rating > '5'::numeric) AND (votes > 500))
                                             Rows Removed by Filter: 858059
               ->  Index Scan using people_pkey on people p  (cost=0.43..0.48 rows=1 width=24) (actual time=0.008..0.008 rows=1 loops=2218756)
                     Index Cond: ((name_id)::text = (pt.name_id)::text)
                     Filter: (death_year IS NULL)
                     Rows Removed by Filter: 0
```

# Top level Expensive path

```
GroupAggregate  (cost=546632.16..549163.19 rows=37497 width=22)
                (actual time=34264.743..37864.405 rows=2933 loops=1)
  Group Key: p.name
  Filter: (count(*) > 10)
  Rows Removed by Filter: 1180924
```

- This is just the top and the real work is below

# Where does the above group get its data?

```
    ->  Sort  (cost=546632.16..546913.39 rows=112490 width=14)
              (actual time=34263.350..37271.095 rows=2110225 loops=1)
          Sort Key: p.name
          Sort Method: external merge  Disk: 50944kB
          ->  Nested Loop  (cost=177244.72..535269.57 rows=112490
width=14)
                           (actual time=1533.309..25609.351
rows=2110225 loops=1)
```

- This pushes data up to the group nodes
- The sort is spilling to disk, but only accounts for 11.7 seconds of the total 37.8 seconds of the total query time
- Inside the Nested Loop is taking 25.6 seconds

# Finally we get to the node taking all the time

```
-> Index Scan using people_pkey on people p  (cost=0.43..0.48 rows=1
width=24)
                                (actual time=0.008..0.008 rows=1
loops=2218756)
      Index Cond: ((name_id)::text = (pt.name_id)::text)
      Filter: (death_year IS NULL)
      Rows Removed by Filter: 0
```

- The index scan takes most of the time
- A single iteration takes 0.008ms
- It is performed 2.2 million times taking over 17 seconds

# Explain Options

- ANALYZE [ boolean ]
  - Run the query and return actual values
- COSTS [ boolean ]
  - show the costs of each node, on by default
- BUFFERS [ boolean ]
  - show buffer usage
- SUMMARY [ boolean ]
  - show the timing information after the plan
- TIMING [ boolean ]
  - show actual timings, on by default
- VERBOSE [ boolean ]
  - show additional details like the output columns
- FORMAT { TEXT | XML | JSON | YAML }
  - output format, TEXT by default

# Explain Options

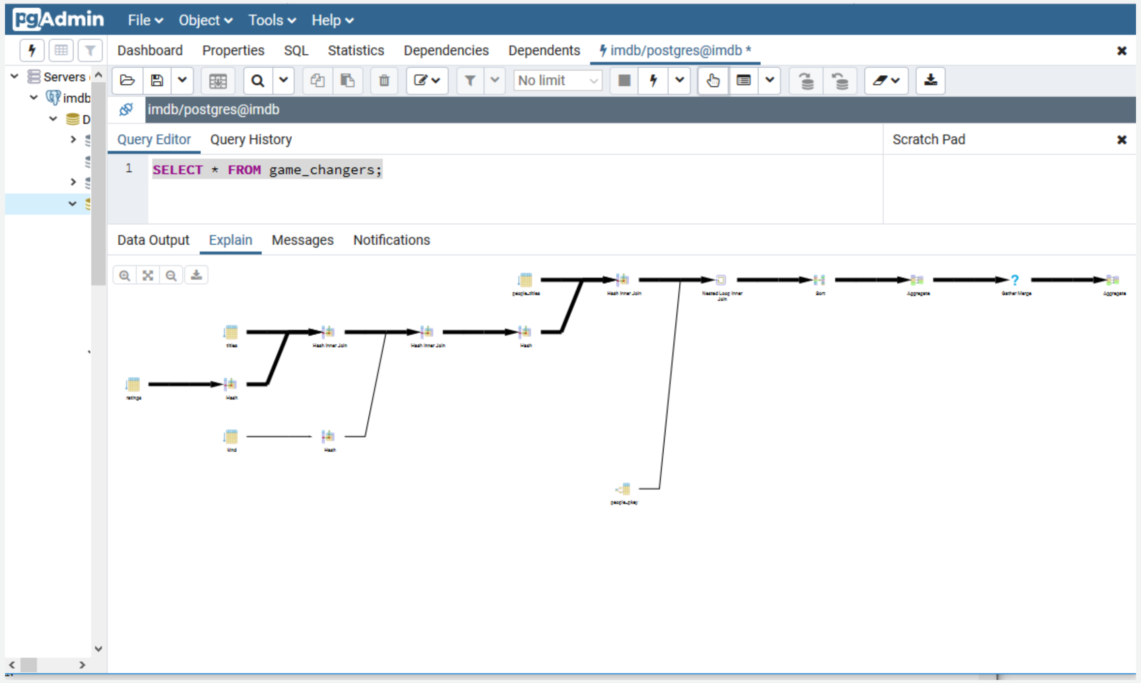
```
=> EXPLAIN (ANALYZE, BUFFERS true, VERBOSE true) SELECT * FROM cities
ORDER BY name;
                                                   QUERY PLAN
-------------------------------------------------------------------------
-------
 Sort  (cost=8216.52..8374.55 rows=63211 width=43)
       (actual time=140.296..176.616 rows=63211 loops=1)
   Output: id, name, state, state_name, county, alias
   Sort Key: cities.name
   Sort Method: external merge  Disk: 3448kB
   Buffers: shared hit=598, temp read=431 written=433
   ->  Seq Scan on public.cities  (cost=0.00..1230.11 rows=63211
width=43)
                                  (actual time=0.011..5.731 rows=63211
loops=1)
         Output: id, name, state, state_name, county, alias
         Buffers: shared hit=598
 Planning Time: 0.053 ms
 Execution Time: 180.949 ms
```

# Explain Options

```
=> EXPLAIN (FORMAT json) SELECT * FROM game_changers;
                   QUERY PLAN
---------------------------------------------------
 [                                                 +
   {                                               +
     "Plan": {                                     +
       "Node Type": "Aggregate",                   +
       "Strategy": "Sorted",                       +
       "Partial Mode": "Simple",                   +
       "Parallel Aware": false,                    +
       "Startup Cost": 546632.16,                  +
       "Total Cost": 549163.19,                    +
       "Plan Rows": 37497,                         +
       "Plan Width": 22,                           +
       "Group Key": ["p.name"],                    +
       "Filter": "(count(*) > 10)",                +
       "Plans": [                                  +
...
```

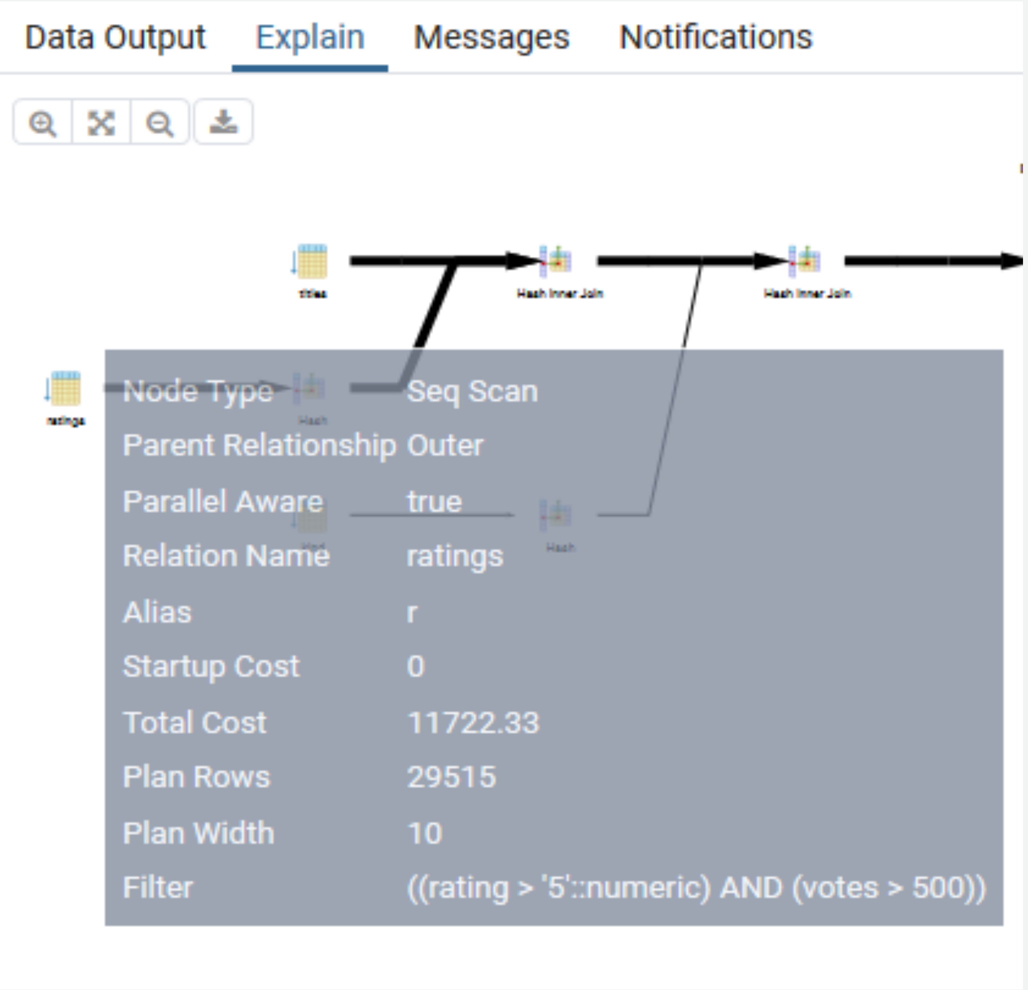
# pgAdmin Visual EXPLAIN

- pgAdmin has a visual EXPLAIN feature built into the query tool
- Produces a graphical output of the explain plan
- Simpler to see the flow of an execution



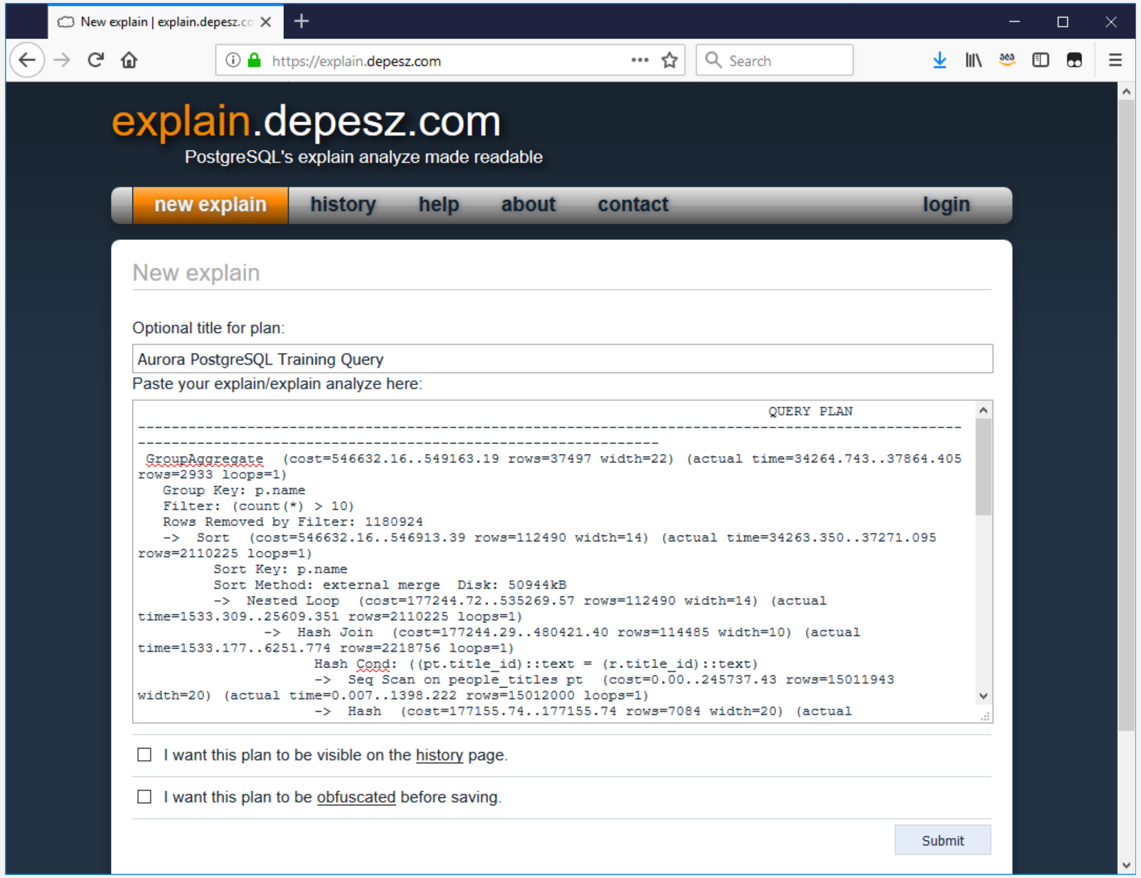
# pgAdmin Visual EXPLAIN

- Diagnosing the exact place in the plan can be more difficult
- Does not return all of the details necessary



# explain.depesz.com

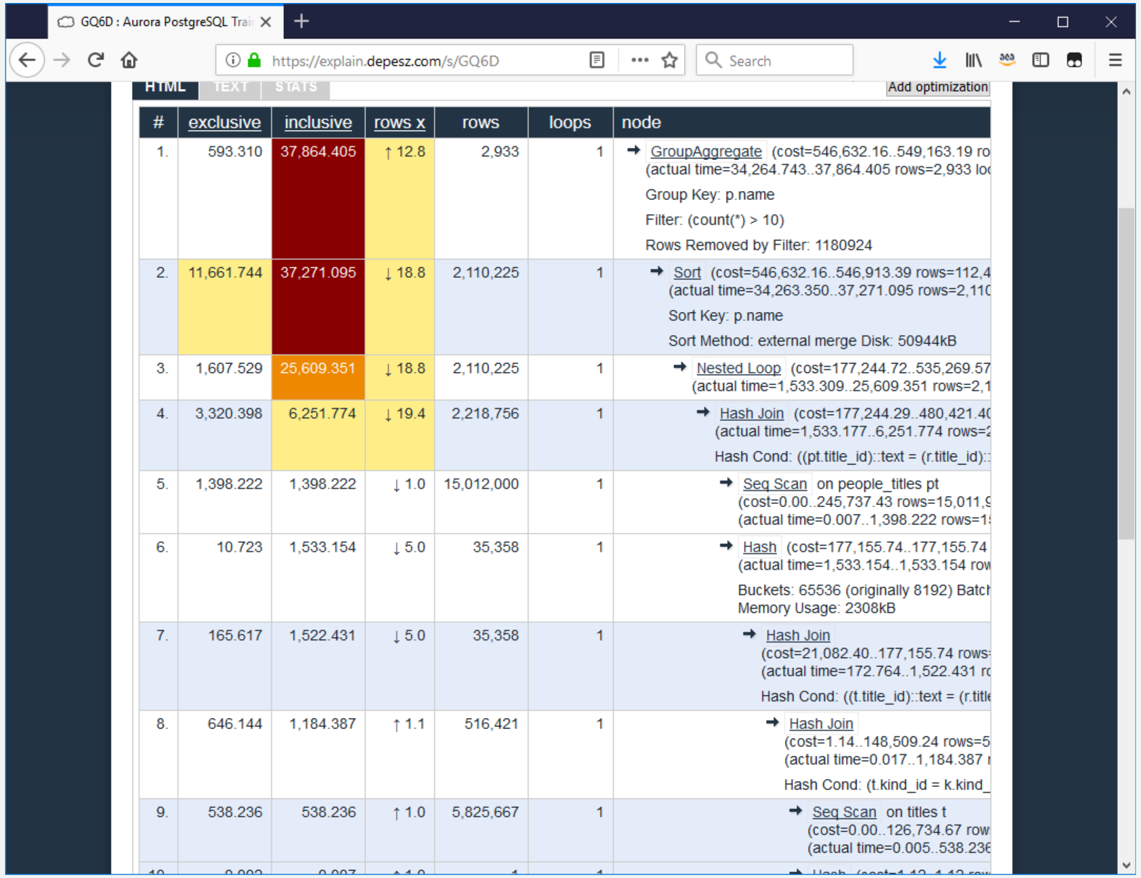
- Simple online tool to interpet EXPLAIN plans
- Paste and existing plan into the site
- Fully open source so can be run in secure environments



# explain.depesz.com

- Highlights areas of the plan that are most expensive

- Red and Orange and places to focus

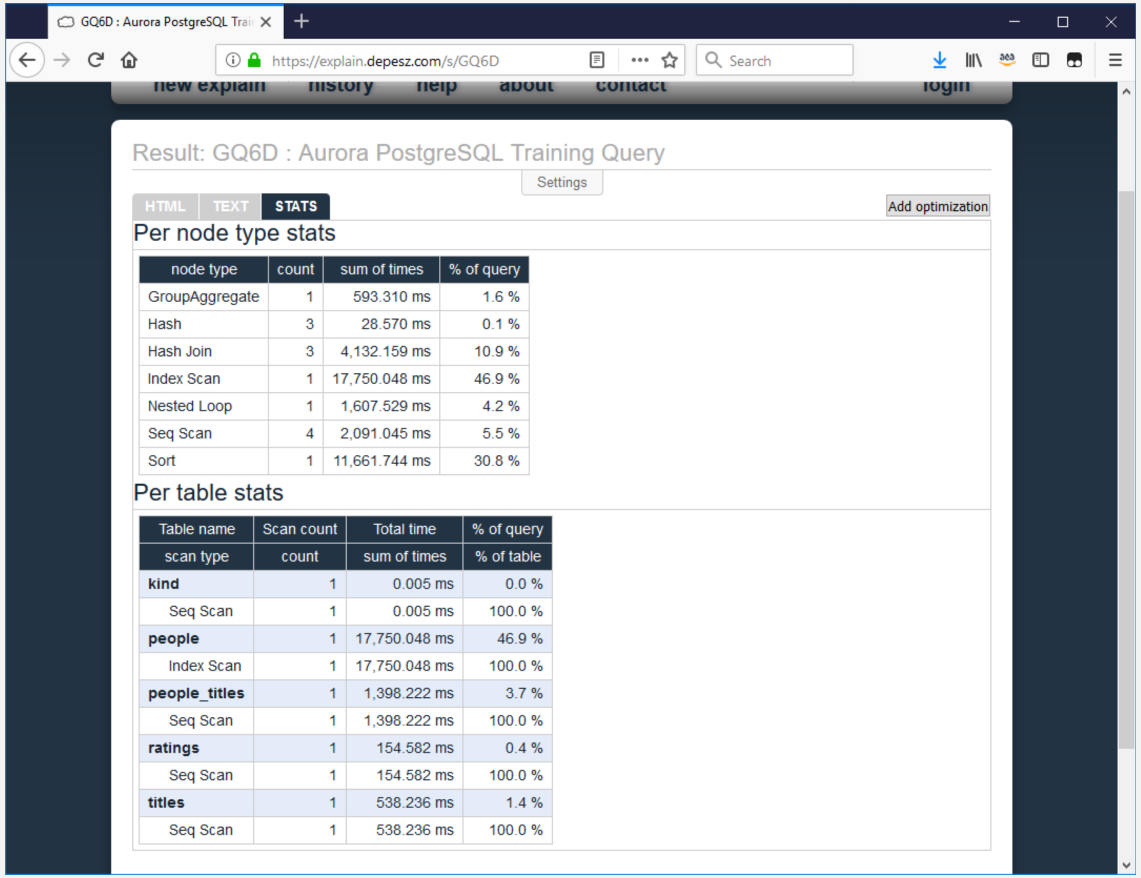



| # | exclusive | inclusive | rows x | rows | loops | node |
|---|---|---|---|---|---|---|
| 1. | 593.310 | 37,864.405 | ↑ 12.8 | 2,933 | 1 | → GroupAggregate (cost=546,632.16..549,163.19 ro (actual time=34,264.743..37,864.405 rows=2,933 lo Group Key: p.name Filter: (count(*) > 10) Rows Removed by Filter: 1180924 |
| 2. | 11,661.744 | 37,271.095 | ↓ 18.8 | 2,110,225 | 1 | → Sort (cost=546,632.16..546,913.39 rows=112,4 (actual time=34,263.350..37,271.095 rows=2,110 Sort Key: p.name Sort Method: external merge Disk: 50944kB |
| 3. | 1,607.529 | 25,609.351 | ↓ 18.8 | 2,110,225 | 1 | → Nested Loop (cost=177,244.72..535,269.57 (actual time=1,533.309..25,609.351 rows=2,1 |
| 4. | 3,320.398 | 6,251.774 | ↓ 19.4 | 2,218,756 | 1 | → Hash Join (cost=177,244.29..480,421.40 (actual time=1,533.177..6,251.774 rows=2 Hash Cond: ((pt.title_id)::text = (r.title_id):: |
| 5. | 1,398.222 | 1,398.222 | ↓ 1.0 | 15,012,000 | 1 | → Seq Scan on people_titles pt (cost=0.00..245,737.43 rows=15,011,9 (actual time=0.007..1,398.222 rows=1 |
| 6. | 10.723 | 1,533.154 | ↓ 5.0 | 35,358 | 1 | → Hash (cost=177,155.74..177,155.74 (actual time=1,533.154..1,533.154 row Buckets: 65536 (originally 8192) Batch Memory Usage: 2308kB |
| 7. | 165.617 | 1,522.431 | ↓ 5.0 | 35,358 | 1 | → Hash Join (cost=21,082.40..177,155.74 rows (actual time=172.764..1,522.431 r Hash Cond: ((t.title_id)::text = (r.title |
| 8. | 646.144 | 1,184.387 | ↑ 1.1 | 516,421 | 1 | → Hash Join (cost=1.14..148,509.24 rows=5 (actual time=0.017..1,184.387 Hash Cond: (t.kind_id = k.kind_ |
| 9. | 538.236 | 538.236 | ↑ 1.0 | 5,825,667 | 1 | → Seq Scan on titles t (cost=0.00..126,734.67 row (actual time=0.005..538.236 |

# explain.depesz.com

- Stats show a breakdown by node and table

- Easy to focus on problem areas




| node type | count | sum of times | % of query |
|---|---|---|---|
| GroupAggregate | 1 | 593.310 ms | 1.6 % |
| Hash | 3 | 28.570 ms | 0.1 % |
| Hash Join | 3 | 4,132.159 ms | 10.9 % |
| Index Scan | 1 | 17,750.048 ms | 46.9 % |
| Nested Loop | 1 | 1,607.529 ms | 4.2 % |
| Seq Scan | 4 | 2,091.045 ms | 5.5 % |
| Sort | 1 | 11,661.744 ms | 30.8 % |

| Table name | Scan count | Total time | % of query |
|---|---|---|---|
| scan type | count | sum of times | % of table |
| **kind** | 1 | 0.005 ms | 0.0 % |
| Seq Scan | 1 | 0.005 ms | 100.0 % |
| **people** | 1 | 17,750.048 ms | 46.9 % |
| Index Scan | 1 | 17,750.048 ms | 100.0 % |
| **people_titles** | 1 | 1,398.222 ms | 3.7 % |
| Seq Scan | 1 | 1,398.222 ms | 100.0 % |
| **ratings** | 1 | 154.582 ms | 0.4 % |
| Seq Scan | 1 | 154.582 ms | 100.0 % |
| **titles** | 1 | 538.236 ms | 1.4 % |
| Seq Scan | 1 | 538.236 ms | 100.0 % |

# Summary

- There is a learning curve to understanding EXPLAIN output
- EXPLAIN does not give the tuning answer, only where to start looking
- Remember that session state affects EXPLAIN results

# Planner Statistics

# Planner Statistics

- Statistics are used by the planner to to estimate the cost of a query plan
- Collected by
  - ANALYZE
  - VACUUM
  - CREATE INDEX
  - REINDEX
  - CLUSTER
- Stored in pg_class and pg_statistic

# Analyze

- Analyze is used to collect statistics about a table
- Results are stored in pg_statistic
- Query planner uses these statistics to increase efficiency
- Analyze should be rerun against tables with many changes

```
tac=# analyze verbose aws.cells;
INFO:  analyzing "aws.cells"
INFO:  "cells": scanned 30000 of 3761114 pages,
       containing 2729773 live rows and 51181 dead rows;
       30000 rows in sample, 310978905 estimated total rows
ANALYZE
```

# Looking at pg_class

- Most columns are structural, but some are statistics

| Column | Type |
| --- | --- |
| relname | name |
| relpages | integer |
| reltuples | real |

```
      relname        | reltuples | relpages
--------------------+-----------+----------
 pgbench_tellers    |       100 |        1
 pgbench_branches   |        10 |        1
 pgbench_accounts   |     1e+06 |    16394
 pgbench_history    |         0 |        0
```

# Understanding pg_statistic

- Contains all of the statistics of all relations

| Column | Type |
| --- | --- |
| starelid | oid |
| staattnum | smallint |
| stainherit | boolean |
| stanullfrac | real |
| stawidth | integer |
| stadistinct | real |
| stakind1 | smallint |
| stakind2 | smallint |
| stakind3 | smallint |
| stakind4 | smallint |
| stakind5 | smallint |
| staop1 | oid |
| staop2 | oid |

| Column | Type |
| --- | --- |
| staop3 | oid |
| staop4 | oid |
| staop5 | oid |
| stanumbers1 | real[] |
| stanumbers2 | real[] |
| stanumbers3 | real[] |
| stanumbers4 | real[] |
| stanumbers5 | real[] |
| stavalues1 | anyarray |
| stavalues2 | anyarray |
| stavalues3 | anyarray |
| stavalues4 | anyarray |
| stavalues5 | anyarray |

# Understanding pg_statistic

```
SELECT staattnum, stanullfrac, stawidth,
       stadistinct, stanumbers1, stavalues1
  FROM pg_statistic WHERE starelid = 'pg_class'::regclass;

...
-[ RECORD 2 ]--------------------------------------------------
staattnum    | 1
stanullfrac  | 0
stawidth     | 64
stadistinct  | -1
stanumbers1  | (null)
stavalues1   | {_pg_foreign_data_wrappers,_pg_foreign_tables,...
```

# Understanding pg_stats

- Contains all of the statistics of all relations

| Column | Type |
| --- | --- |
| schemaname | name |
| tablename | name |
| attname | name |
| inherited | boolean |
| null_frac | real |
| avg_width | integer |
| n_distinct | real |
| most_common_vals | anyarray |
| most_common_freqs | real[] |
| histogram_bounds | anyarray |
| correlation | real |
| most_common_elems | anyarray |
| most_common_elem_freqs | real[] |
| elem_count_histogram | real[] |

# Understanding pg_stats

```
SELECT attname, null_frac, avg_width, n_distinct,
       most_common_vals, most_common_freqs, histogram_bounds
  FROM pg_stats
 WHERE schemaname = 'pg_catalog' AND tablename = 'pg_class';

...
-[ RECORD 2 ]-----+------------------------------------------
attname           | relname
null_frac         | 0
avg_width         | 64
n_distinct        | -1
most_common_vals  | (null)
most_common_freqs | (null)
histogram_bounds  | {_pg_foreign_data_wrappers,_pg_foreign_tables,...
```

# Understanding pg_stats

```
...
-[ RECORD 10 ]----+---------------------------------------------
attname           | relpages
null_frac         | 0
avg_width         | 4
n_distinct        | 14
most_common_vals  | {0,1,2,5}
most_common_freqs | {0.53481,0.268987,0.129747,0.0221519}
histogram_bounds  | {3,3,4,4,6,9,10,15,31,72}
```

# Helping the Optimizer

- Understand and react to uneven distributions of data

- default_statistics_target
  - Effects collection of most frequent values and histogram statistics

- ALTER TABLE SET STATISTICS
  - Set low on columns with simple data distributions
  - Set higher on frequently accessed search or join columns with known complex data

# Indexes

# Indexes

- Indexes are a common way to enhance performance

- PostgreSQL supports several index types:
  - B-tree (default)
  - Hash – Use when a simple comparison using the “=“ operator is needed
  - Index on Expressions – Use when an often used expression is queried. Inserts and updates will be slower.
  - Partial Index – Indexes only rows that satisfy the WHERE clause. A query must include the same WHERE clause.

```
CREATE INDEX <name> on <table> (<column>);
CREATE INDEX <name> ON <table> USING HASH (<column>);
CREATE INDEX <name> on <table>(expression(<column(s)>));
CREATE INDEX <name> ON <table> (<column>) WHERE <where clause>;
```

# B-Trees Index

- DELETEs don’t remove links – improves concurrency
- INSERTs on Rightmost page most efficient
- Suitable for both unique and highly non-unique data, or very skewed data
- Can use index-only scans

# B-Trees Index

```
EXPLAIN SELECT original_title FROM titles WHERE tid = 'tt0076759';
                               QUERY PLAN
-------------------------------------------------------------------
 Index Scan using titles_pkey on titles
     (cost=0.43..8.45 rows=1 width=20)
   Index Cond: ((tid)::text = 'tt0076759'::text)
(2 rows)

EXPLAIN SELECT primary_title FROM titles WHERE tid = 'tt0076759';
                               QUERY PLAN
-------------------------------------------------------------------
 Index Only Scan using titles_id_title_idx on titles
     (cost=0.56..4.58 rows=1 width=20)
   Index Cond: (tid = 'tt0076759'::text)
(2 rows)
```

# Hash Index

- Allows equality search only
- Ideal for lookups of large strings like UUIDs
- Can not be used as unique constraints

# Hash Index

```
CREATE INDEX titles_guid_hidx ON titles USING HASH (guid);

 EXPLAIN ANALYZE SELECT primary_title FROM titles
        WHERE guid = 'e6524bb6-761e-11e8-a734-c73f598fdce4';
                              QUERY PLAN
-------------------------------------------------------------------
Index Scan using titles_guid_hidx on titles
  (cost=0.00..8.02 rows=1 width=20)
  (actual time=0.010..0.011 rows=1 loops=1)
  Index Cond: (guid = 'e6524bb6-761e-11e8-a734-c73f598fdce4'::uuid)
Planning time: 0.056 ms
Execution time: 0.024 ms
(4 rows)
```

# GIST Index

- Generalized Search Tree
  - Allows application/custom data type specific indexes

- Used by
  - TSEARCH2 – Full text indexing
  - POSTGIS – R-Trees for spatial searching

# GIST Index

```
CREATE INDEX titles_title_gist_idx ON titles
    USING GIST (primary_title gist_trgm_ops);

EXPLAIN ANALYZE SELECT tid, primary_title FROM titles
             WHERE primary_title ILIKE '%star wars%';
                                   QUERY PLAN
-------------------------------------------------------------------
Bitmap Heap Scan on titles  (cost=31.94..1746.26 rows=455 width=30)
                   (actual time=789.989..800.880 rows=1668 loops=1)
  Recheck Cond: ((primary_title)::text ~~* '%star wars%'::text)
  Rows Removed by Index Recheck: 20
  Heap Blocks: exact=1413
  ->  Bitmap Index Scan on titles_title_gist_idx
            (cost=0.00..31.83 rows=455 width=0)
            (actual time=789.719..789.719 rows=1688 loops=1)
        Index Cond: ((primary_title)::text ~~* '%star wars%'::text)
Planning time: 0.568 ms
Execution time: 801.639 ms
(8 rows)
```

# GIN Index

- Generalized Inverted Index
  - Allows application/custom data type specific indexes

- Used by
  - TSEARCH2 – Full text indexing
  - POSTGIS – R-Trees for spatial searching

# GIN Index

```
CREATE INDEX titles_title_gin_idx ON titles
     USING GIN (primary_title gin_trgm_ops);

EXPLAIN ANALYZE SELECT tid, primary_title FROM titles
                WHERE primary_title ILIKE '%star wars%';
                           QUERY PLAN
-------------------------------------------------------------------
Bitmap Heap Scan on titles  (cost=91.54..1809.51 rows=456 width=30)
                         (actual time=41.174..45.603 rows=1668 loops=1)
  Recheck Cond: ((primary_title)::text ~~* '%star wars%'::text)
  Rows Removed by Index Recheck: 20
  Heap Blocks: exact=1413
  ->  Bitmap Index Scan on titles_title_gin_idx
           (cost=0.00..91.42 rows=456 width=0)
           (actual time=40.905..40.905 rows=1688 loops=1)
        Index Cond: ((primary_title)::text ~~* '%star wars%'::text)
Planning time: 0.543 ms
Execution time: 45.751 ms
(8 rows)
```

# GIST vs GIN

- GIN is about 3X faster for lookups than GIST
- GIN takes about 3X longer to build than GIST
- GIN is about 10X slower to update than GIST
- GIN is about 2-3X smaller on disk than GIST

# Partial Index

- Allows you to index just some of the rows of a table
  - e.g. Column is unique except when NULL
  - e.g. Fast access required only to Employee.CPRTrained = ‘Y’
- Defined by WHERE clause on CREATE INDEX
- Must be defined using constants
  - Cannot build index WHERE log_date > current_date – 30 since current_date value changes as it is executed

# Partial Index

```
CREATE INDEX titles_movies_id_idx ON titles(tid)
  WHERE title_type = 'movie';

 EXPLAIN ANALYZE SELECT * FROM titles
 WHERE tid = 'tt0076759' AND title_type = 'movie';
                         QUERY PLAN
-------------------------------------------------------------------
 Index Scan using titles_movies_id_idx on titles
   (cost=0.42..8.44 rows=1 width=79)
   (actual time=0.047..0.049 rows=1 loops=1)
   Index Cond: ((tid)::text = 'tt0076759'::text)
 Planning time: 0.400 ms
 Execution time: 0.159 ms
(4 rows)
```

# Multiple Indexes

```
EXPLAIN ANALYZE SELECT * FROM people WHERE first_name = 'Bruce' AND last_name =
'Lee';
                                              QUERY PLAN
----------------------------------------------------------------------------------
----
Bitmap Heap Scan on people  (cost=452.18..456.19 rows=1 width=42)
                            (actual time=4.824..4.844 rows=18 loops=1)
 Recheck Cond: (((first_name)::text = 'Bruce') AND ((last_name)::text = 'Lee'))
 Heap Blocks: exact=18
 ->  BitmapAnd  (cost=452.18..452.18 rows=1 width=0)
                (actual time=4.776..4.776 rows=0 loops=1)
   ->  Bitmap Index Scan on people_first_name_idx (cost=0.00..12.88 rows=592
width=0)
                                          (actual time=1.322..1.322 rows=7956
loops=1)
         Index Cond: ((first_name)::text = 'Bruce'::text)
   ->  Bitmap Index Scan on people_last_name_idx (cost=0.00..439.05 rows=21682
width=0)
                                           (actual time=3.098..3.098 rows=21947
loops=1)
         Index Cond: ((last_name)::text = 'Lee'::text)
Planning Time: 0.101 ms
Execution Time: 4.873 ms
```

# Compound Indexes

- Multiple columns can be used in a single index
- Order of the columns matter

```
CREATE INDEX people_name_idx ON people (last_name, first_name);

EXPLAIN ANALYZE SELECT * FROM people WHERE first_name = 'Bruce' AND
last_name = 'Lee';
                                              QUERY PLAN
-------------------------------------------------------------------------
--------------
 Index Scan using people_name_idx on people  (cost=0.56..8.58 rows=1
width=42)
                                            (actual time=0.038..0.056 rows=18
loops=1)
   Index Cond: (((last_name)::text = 'Lee') AND ((first_name)::text =
'Bruce'))
 Planning Time: 0.304 ms
 Execution Time: 0.073 ms
```

# Summary

- Indexes can help queries perform much better
- The correct type and scope of the index matter
- Indexes come at a cost. Write performance is affected

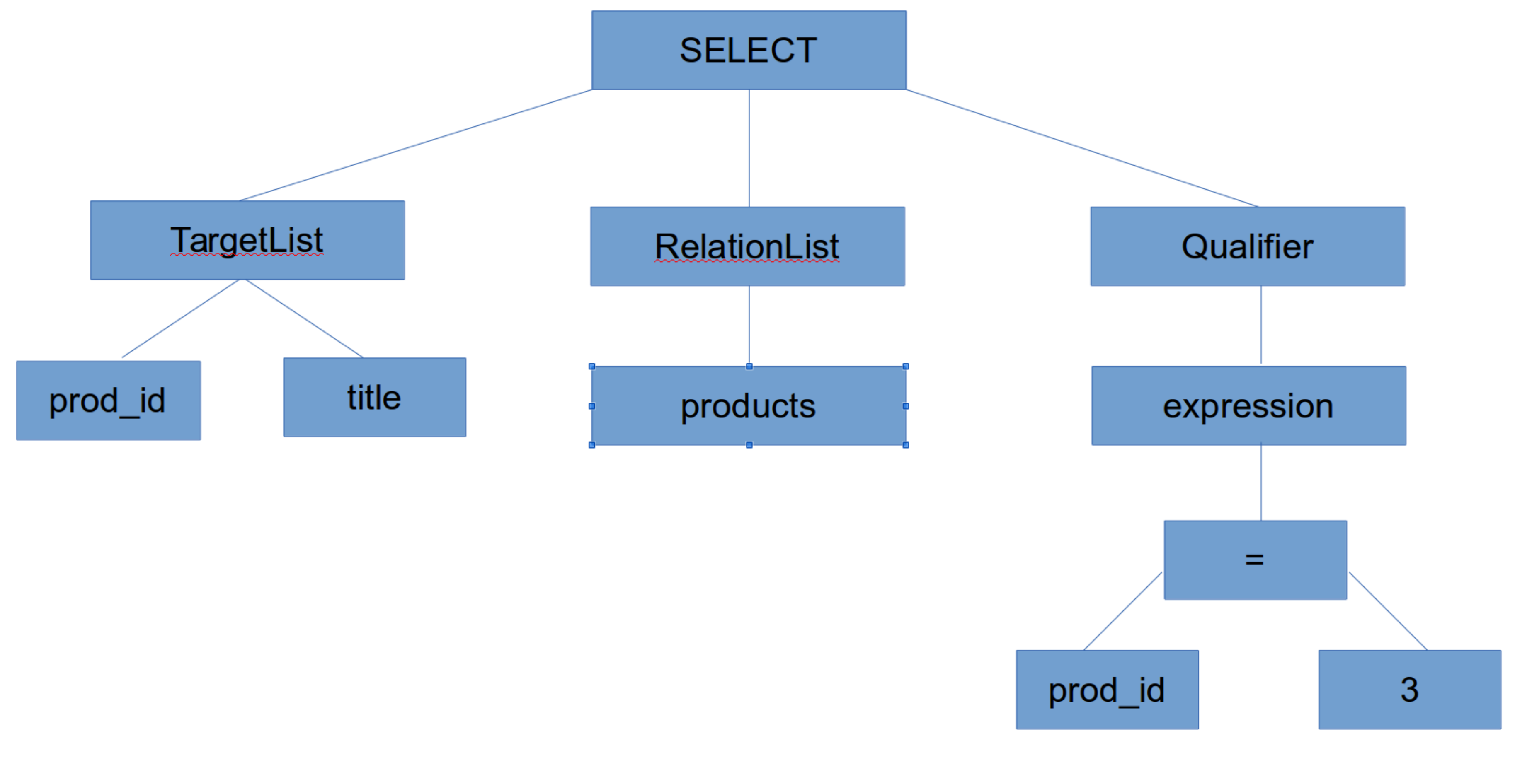
# Understanding the Optimizer

# What is the Optimizer?

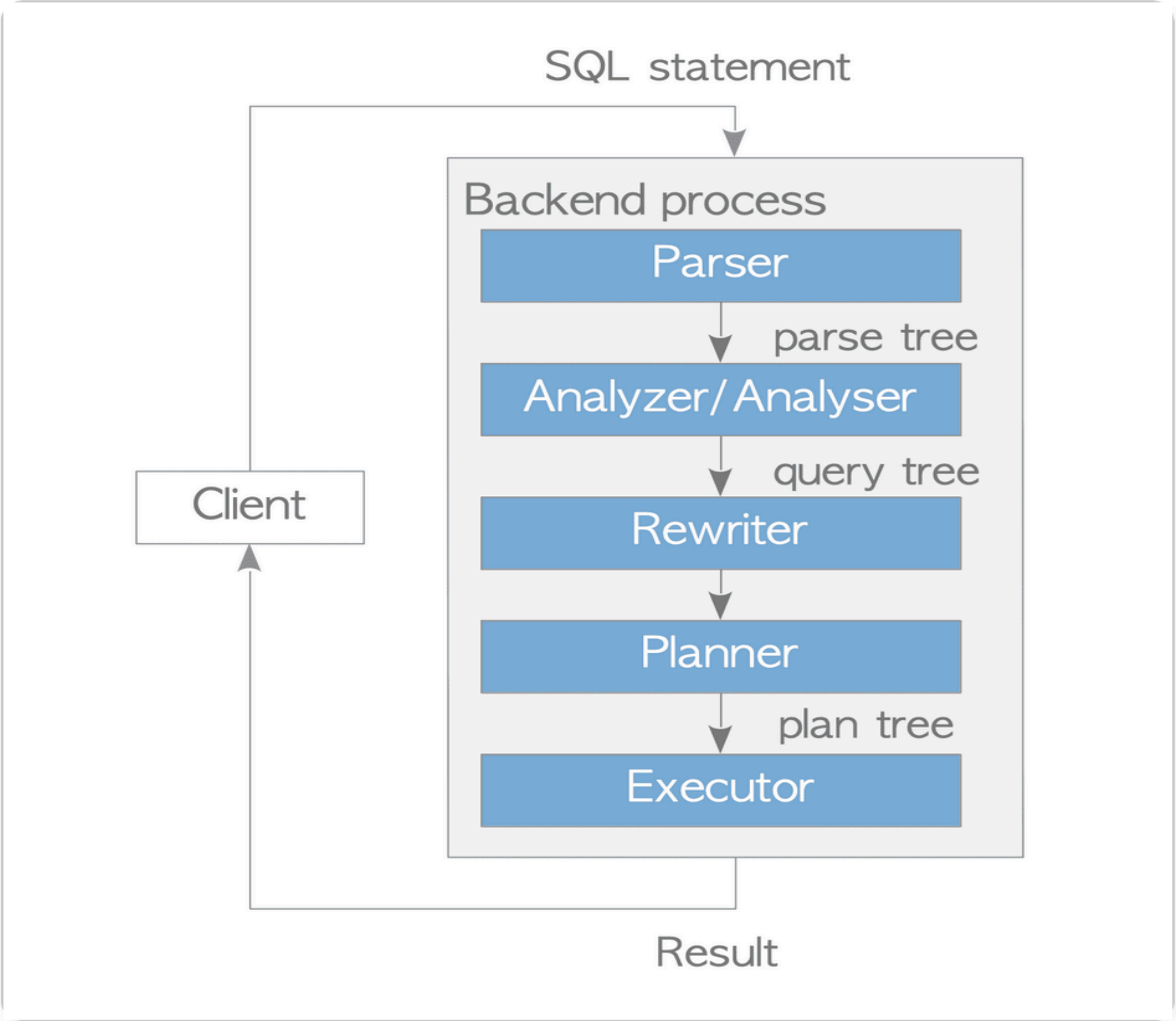
- Creates an optimal execution plan for a query
- Examines a SQL query in a variety of different ways
  - If feasible, it will evaluate each one
- The number of possible plans grows exponentially as the number of joined tables increases
  - Uses the algorithm introduced by the System R database (became DB2)
  - For more than 12 joins, switches to a Genetic Query Optimization
- Costs are measured relative to the cost of a sequential page fetch

# Parse Trees

```sql
select prod_id, title
from products
where prod_id = 3
```


SELECT
TargetList
RelationList
Qualifier
prod_id
title
products
expression
=
prod_id
3

- Parser : Generates a parse tree from the statement in Plain Text
- Analyzer : Carries out a semantic analysis of the parse tree and generates a query tree
- Rewriter : Transforms a query tree using rules in the rules system
- Planner : Planner generates the tree that can be most effectively be executed from the query tree
- Executor : Executes the query by accessing the tables and indexes in the order that was created in the plan tree.



# Getting the Parse Tree

- Turn on parse debugging

```
SET debug_print_parse = on;
SET client_min_messages='LOG';
LOG:  parse tree:
DETAIL:     {QUERY
   :commandType 1
...
   :cteList <>
   :rtable (
      {RTE
      :alias <>
      :eref
         {ALIAS
         :aliasname foo
         :colnames ("row_owner" "col1" "col2")
         }
      :rtekind 0
      :relid 16386
      :relkind r
      :tablesample <>
...
      :selectedCols (b 10 11)
      :insertedCols (b)
      :updatedCols (b)
      :securityQuals <>
      }
   )
```

# Turning a Parse Tree into a Plan

- Uses a series of query operators to return a result set
- The planner generates many execution plans and the optimizer determines the least-expensive plan
- Uses tables statistics to determine

```
# EXPLAIN SELECT col1, col2 FROM foo WHERE col1 = 2;
                              QUERY PLAN
-------------------------------------------------------------------
 Index Scan using foo_idx on foo  (cost=0.28..8.29 rows=1 width=8)
   Index Cond: (col1 = 2)
(2 rows)
```

# Table Level Operators

- 6 methods to retrieve data from a table
  - Table Scan (seq scan)
    - Full scan of the table from beginning to end
  - Index Scan
    - Scans an index to find the appropriate row in the table
  - Bitmap Index Scan
    - Scans the full index and finds matching keys
  - Index Only Scan
    - Returns the target list out of the index
  - Remote Scan
    - Scan a foreign server
  - tuple-ID Scan (TID scan)
    - Only used ctid is used as a criteria

# Join Operators

- Only 3 possible join strategies
  - Nested Loop Join
    - The right relation scanned once for every relation on the left
  - Merge Sort Join
    - Each relation is sorted and scanned once in parallel
  - Hash Join
    - The right relation is loaded into a hash table and the left relation uses hash keys to find matching rows

# Other Operators

- Sort
  - Orders the result set either in-memory or on-disk
- Unique
  - Eliminates duplicate rows from an input set
- Limit
  - Limits the size of a result set
- Aggregate
  - Reads all of the rows in the input set and calculates the aggregate value
- Append
  - Implements a UNION
- Result
  - Used for a query that does not return data

# Other Operators (cont.)

- Group
  - Used to satisfy a GROUP BY clause
- Subquery Scan
  - Used to satisfy a UNION
- Subplan
  - Used to satisfy a subselect
- Materialize
  - Used to materialize a subselect
- Setop
  - Used for INTERSECT and EXCEPT operators

# Affecting the Optimizer

- Configuration settings
  - EFFECTIVE_CACHE_SIZE
    - The estimate of the cache available for an index scan. Higher values favor index scans
  - RANDOM_PAGE_COST
    - The cost of reading a non sequential disk page. Higher values favor sequential scans
  - CPU_INDEX_TUPLE_COST
    - The cost of processing an index row.
  - CPU_OPERATOR_COST
    - The cost of processing each operator in the WHERE clause
  - CPU_TUPLE_COST
    - The cost of processing each row

# Operators Costing Algorithms

- Sequential Scan
  - Cost = Number of Pages
- Index Scan
  - Uses the Mackert and Lohman approximation algorithm

# Optimizer Hints

# Hints (Caution)

- Hints are not magic
- They should be used with extreme care

# What are hints?

- Allows for the developer or operator to influence the optimizer
- It is a directive to the optimizer to be followed if possible

# Using Hints

RDS > Parameter groups > training-pg11

## training-pg11

**Parameters**

Cancel editing | Preview changes | Reset | Save changes

shared_pre

< 1 >

| | Name | Values | Allowed values | Modifiable | Source | Apply type | Data type |
|---|---|---|---|---|---|---|---|
| ☑ | shared_preload_libraries | ȷ_hint | auto_explain, orafce, pgaudit, pglogical, pg_similarity, pg_stat_statements, pg_hint_plan | true | system | static | list |

pg_hint_plan must be added to shared_preload_libraries

# Adding Hints

- Hints must be the first element of a statement
  - Will be treated as a comment otherwise

```
/*+
    BitmapScan(titles titles_id_brin_idx)
*/
EXPLAIN ANALYZE SELECT * FROM titles WHERE title_id =  'tt0076759';
```

# Hints Types

- **Scans** - How is data retrieved from a single table

- **Joins** - How are tables combined

- **Environment** - Parameters and statistics used by the query

# Hints Types - Scans

| Format | Description |
|---|---|
| SeqScan(table) | Forces sequential scan on the table |
| TidScan(table) | Forces TID scan |
| IndexScan(table[ index...]) | Forces index scan |
| IndexOnlyScan(table[ index...]) | Forces index only scan |
| BitmapScan(table[ index...]) | Forces bitmap scan |
| NoSeqScan(table) | Do not do sequential scan on the table |

# Hints Types - Joins

| Format | Description |
|---|---|
| NestLoop(table table[ table…]) | Forces nested loop for the joins |
| HashJoin(table table[ table…]) | Forces hash join |
| MergeJoin(table table[ table…]) | Forces merge join |
| NoNestLoop(table table[ table…]) | Do not do nested loop join |
| Leading(table table[ table…]) | Forces join order as specified |

# Hints Types - Environment

| Format | Description |
| --- | --- |
| Rows(table table[ table…] correction) | Corrects row number of a result of the joins |
| Set(param value) | Set the parameter while planner is running |

# Summary

- Hints can be a powerful tool to tune queries

- Use with caution

# Summary

# Summary

- Use EXPLAIN on all important queries to ensure it is executing the optimal way
- Knowing the data and the data usage helps determine the correct index type
- Just because an index scan is used does not mean it is the best index scan possible
- PostgreSQL does not limit the number of indexes on a table, but each index adds overhead to writes