Explaining EXPLAIN
EXPLAIN

tpcc=> EXPLAIN
SELECT *
FROM oorder
JOIN order_line ON (ol_w_id = o_w_id
  AND ol_d_id = o_w_id
  AND ol_o_id = o_id)
JOIN item ON (i_id = ol_i_id)
JOIN stock ON (s_w_id = o_w_id AND s_i_id = i_id)
JOIN warehouse ON (w_id = o_w_id)
JOIN district ON (d_w_id = w_id AND d_id = o_d_id)
JOIN customer ON (c_w_id = w_id
  AND c_d_id = d_id
  AND c_id = o_c_id)
WHERE o_w_id = 1

•EXPLAIN works on any DML not just SELECT (ie UPDATE, DELETE, and INSERT)
EXPLAIN

Hash Join (cost=22896.89..54208.53 rows=330801 width=1239)
  Hash Cond: (order_line.ol_o_id = oorder.o_id)
  ->  Nested Loop (cost=8853.68..27149.42 rows=32734 width=542)
  ->  Seq Scan on warehouse (cost=0.00..1.01 rows=1 width=85)
      Filter: (w_id = 1)
  ->  Merge Join (cost=8853.68..26821.07 rows=32734 width=457)
      Merge Cond: (order_line.ol_i_id = item.i_id)
  ->  Merge Join (cost=8852.66..22503.03 rows=32734 width=385)
      Merge Cond: (stock.s_i_id = order_line.ol_i_id)
      ->  Index Scan using pk_stock on stock (cost=0.00..12910.70 rows=100000 width=315)
          Index Cond: (s_w_id = 1)
      ->  Materialize (cost=8852.63..9261.81 rows=32734 width=70)
          ->  Sort (cost=8852.63..8934.47 rows=32734 width=70)
              Sort Key: order_line.ol_i_id
              ->  Bitmap Heap Scan on order_line (cost=843.82..5053.83 rows=32734 width=70)
                  Recheck Cond: ((ol_w_id = 1) AND (ol_d_id = 1))
              ->  Bitmap Index Scan on pk_order_line (cost=0.00..835.64 rows=32734 width=0)
                  Index Cond: ((ol_w_id = 1) AND (ol_d_id = 1))
  ->  Hash (cost=11040.12..11040.12 rows=29767 width=697)
  ->  Hash Join (cost=3743.15..11040.12 rows=29767 width=697)
      Hash Cond: (oorder.o_d_id = district.d_id)
      ->  Merge Join (cost=3741.90..10629.58 rows=29767 width=606)
          Merge Cond: ((customer.c_d_id = oorder.o_d_id) AND (customer.c_id = oorder.o_c_id))
          ->  Index Scan using pk_customer on customer (cost=0.00..6215.00 rows=30000 width=564)
              Index Cond: (c_w_id = 1)
          ->  Materialize (cost=3741.90..4116.90 rows=30000 width=42)
              ->  Sort (cost=3741.90..3816.90 rows=30000 width=42)
                  Sort Key: oorder.o_d_id, oorder.o_c_id
                  ->  Seq Scan on oorder (cost=0.00..636.00 rows=30000 width=42)
                      Filter: (o_w_id = 1)
      ->  Hash (cost=1.12..1.12 rows=10 width=91)
          ->  Seq Scan on district (cost=0.00..1.12 rows=10 width=91)
              Filter: (d_w_id = 1)
EXPLAIN in pgAdmin
Hash Join (cost=22896.89..54208.53 rows=330801 width=1239)
 Hash Cond: (order_line.ol_o_id = order.o_id)
  -> Nested Loop (cost=8853.68..27149.42 rows=32734 width=542)
    -> Seq Scan on warehouse (cost=0.00..1.01 rows=1 width=85)
      Filter: (w_id = 1)
    -> Merge Join (cost=8853.68..26821.07 rows=32734 width=457)
      Merge Cond: (order_line.ol_i_id = item.i_id)
    -> Merge Join (cost=8852.66..22503.03 rows=32734 width=385)
      Merge Cond: (stock.s_i_id = order_line.ol_i_id)
        -> Index Scan using pk_stock on stock (cost=0.00..12910.70 rows=100000)
          Index Cond: (s_w_id = 1)
        -> Materialize (cost=8852.63..9261.81 rows=32734 width=70)
          -> Sort (cost=8852.63..8934.47 rows=32734 width=70)
            Sort Key: order_line.ol_i_id
              -> Bitmap Heap Scan on order_line (cost=843.82..5053.83 rows=327)
                Recheck Cond: ((ol_w_id = 1) AND (ol_d_id = 1))
                  -> Bitmap Index Scan on pk_order_line (cost=0.00..835.64 rows=100000)
                    Index Cond: ((ol_w_id = 1) AND (ol_d_id = 1))
                  -> Index Scan using pk_item on item (cost=0.00..3659.26 rows=100000 width=1)
Hash Join (cost=22896.89..54208.53 rows=330801 width=1239)
  Hash Cond: (order_line.ol_o_id = oorder.o_id)
  -> Nested Loop (cost=8853.68..27149.42 rows=32734 width=542)
    -> Seq Scan on warehouse (cost=0.00..1.01 rows=1 width=85)
      Filter: (w_id = 1)
    -> Merge Join (cost=8853.68..26821.07 rows=32734 width=457)
      Merge Cond: (order_line.ol_i_id = item.i_id)
    -> Merge Join (cost=8852.66..22503.03 rows=32734 width=385)
      Merge Cond: (stock.s_i_id = order_line.ol_i_id)
      -> Index Scan using pk_stock on stock (cost=0.00..12910.70 rows=100000)
        Index Cond: (s_w_id = 1)
      -> Materialize (cost=8852.63..9261.81 rows=32734 width=70)
        -> Sort (cost=8852.63..8934.47 rows=32734 width=70)
          Sort Key: order_line.ol_i_id
          -> Bitmap Heap Scan on order_line (cost=843.82..5053.83)
            Recheck Cond: ((ol_w_id = 1) AND (ol_d_id = 1))
            -> Bitmap Index Scan on pk_order_line (cost=0.00..835.64 rows=100000)
              Index Cond: ((ol_w_id = 1) AND (ol_d_id = 1))
    -> Index Scan using pk_item on item (cost=0.00..3659.26 rows=100000 width=
Costs add up

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
<th>vs page read</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq_page_cost</td>
<td>cost of a sequentially fetched disk page.</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>random_page_cost</td>
<td>cost of a nonsequentially fetched disk page.</td>
<td>4.00</td>
<td>4x slower</td>
</tr>
<tr>
<td>cpu_tuple_cost</td>
<td>cost of processing each tuple (row).</td>
<td>0.01</td>
<td>100x faster</td>
</tr>
<tr>
<td>cpu_operator_cost</td>
<td>cost of processing each operator or function call.</td>
<td>0.0025</td>
<td>400x faster</td>
</tr>
<tr>
<td>cpu_index_tuple_cost</td>
<td>cost of processing each index entry during an index scan.</td>
<td>0.005</td>
<td>1000x faster</td>
</tr>
</tbody>
</table>

- Costs are estimates of the time a node is expected to take
- By default costs are in units of “time a sequential 8kb block read takes”
- Each node has two costs, “startup” cost and “total” cost
- Costs cumulative – parents assume their children's costs
- Optimizer selects plans based on overall lowest startup and total cost
tpcc=> EXPLAIN ANALYSE
SELECT  *
  FROM oorder
  JOIN order_line   ON (ol_w_id = o_w_id
                        AND ol_d_id = o_w_id
                        AND ol_o_id = o_id)
  JOIN item         ON (i_id = ol_i_id)
  JOIN stock        ON (s_w_id = o_w_id AND s_i_id = i_id)
  JOIN warehouse    ON (w_id = o_w_id)
  JOIN district     ON (d_w_id = w_id AND d_id = o_d_id)
  JOIN customer     ON (c_w_id = w_id
                        AND c_d_id = d_id
                        AND c_id = o_c_id)
WHERE o_w_id = 1 ;
Large discrepancies between row estimates and reality are a prime suspect for poorly performing queries.
Cost Versus Actual Time

Hash Join (cost=22896.89..54208.53 rows=330801 width=1239)
  (actual time=1232.035..5577.446 rows=299020 loops=1)
Hash Cond: (order_line.ol_o_id = oorder.o_id)
  ->  Nested Loop (cost=8853.68..27149.42 rows=32734 width=542)
      (actual time=170.958..1545.218 rows=29902 loops=1)
  ->  Seq Scan on warehouse (cost=0.00..1.01 rows=1 width=85)
      (actual time=0.019..0.023 rows=1 loops=1)
      Filter: (w_id = 1)
  ->  Merge Join (cost=8852.66..26821.07 rows=32734 width=457)
      (actual time=170.928..1424.466 rows=29902 loops=1)
      Merge Cond: (order_line.ol_i_id = item.i_id)
  ->  Merge Join (cost=8852.66..22503.03 rows=32734 width=385)
      (actual time=170.830..913.964 rows=29902 loops=1)
      Merge Cond: (stock.s_i_id = order_line.ol_i_id)
...

• Costs are not (normally) in ms!
• But they should be roughly proportional to time
Plan Nodes

- **Scans**
  - Table scans (Sequential, Index, Bitmap, tid)
  - Other scans (Function, Values, Result)
- **Joins**
  - Nested Loop, Merge, Hash
- **Set Operations, Partitioned Tables, and Inheritance**
  - Append
  - SetOp Except, Intersect
- **Miscellaneous**
  - Sort, Aggregate, Unique, Limit
  - Materialize
  - SubPlan, Initplan
Table Scans – Sequential Scans

```
tpcc=> explain select * from stock;
QUERY PLAN
---------------------------------------------------------------
Seq Scan on stock  (cost=0.00..5348.00 rows=100000 width=315)
```

- Fast to start up
- Sequential I/O is **much** faster than random access
- Only has to read each block once
- Produces unordered output
**Table Scans – Index Scans**

```sql
tpcc=> explain select * from stock where s_w_id = 1 and s_i_id = 1;
QUERY PLAN
------------------------------------------------------------------------
Index Scan using pk_stock on stock  (cost=0.00..8.28 rows=1 width=315)
  Index Cond: ((s_w_id = 1) AND (s_i_id = 1))
------------------------------------------------------------------------
```

- Random access is **much** slower than sequential I/O
- Also requires additional I/O to access index
- Worse, potentially has to read blocks multiple times
- Only scan which produces ordered output
Table Scans – Bitmap Index/Heap Scans

tpcc=# explain select * from stock where s_i_id in (1,3,5) or s_i_id in (2,4);

QUERY PLAN

<table>
<thead>
<tr>
<th>QUERY PLAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitmap Heap Scan on stock (cost=5959.28..5978.85 rows=5 width=315)</td>
</tr>
<tr>
<td>Recheck Cond: ((s_i_id = ANY ('{1,3,5}'::integer[])) OR (s_i_id = ANY ('{2,4}'::integer[])))</td>
</tr>
<tr>
<td>-&gt; BitmapOr (cost=5959.28..5959.28 rows=5 width=0)</td>
</tr>
<tr>
<td>-&gt; Bitmap Index Scan on pk_stock (cost=0.00..3354.76 rows=3 width=0)</td>
</tr>
<tr>
<td>Index Cond: (s_i_id = ANY ('{1,3,5}'::integer[]))</td>
</tr>
<tr>
<td>-&gt; Bitmap Index Scan on pk_stock (cost=0.00..2604.51 rows=2 width=0)</td>
</tr>
<tr>
<td>Index Cond: (s_i_id = ANY ('{2,4}'::integer[]))</td>
</tr>
</tbody>
</table>

• Best of both worlds – sequential I/O with index selectivity
• But slow to start up due to having to read all the index tuples and sort them
• Often selected for IN and =ANY(array) operators
• Can combine multiple indexes
• But optimizer can choose it for any indexable scan with low selectivity
• Often ideal for DSS queries
• Produces unordered output
More Scans (Function, Values, Result)

tpcc=> explain select * from generate_series(1,100);

QUERY PLAN
--------------------------------------
Function Scan on generate_series (cost=0.00..12.50 rows=1000 width=4)

tpcc=> explain values (1), (2),(3);

QUERY PLAN
--------------------------------------
Values Scan on "*VALUES*" (cost=0.00..0.04 rows=3 width=4)

tpcc=> explain select 1;

QUERY PLAN
--------------------------------------
Result (cost=0.00..0.01 rows=1 width=0)
Nested Loop Joins

tpcc=> explain select * from order_line join item on (i_id = ol_i_id) where ol_o_id = 1;

QUERY PLAN

--------------------------------------------------------------------------------------------
<table>
<thead>
<tr>
<th>Nested Loop (cost=0.00..7849.57 rows=102 width=142)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-&gt; Index Scan using pk_o_line on order_line (cost=0.00..7092.11 rows=102 width=70)</td>
</tr>
<tr>
<td>Index Cond: (ol_o_id = 1)</td>
</tr>
<tr>
<td>-&gt; Index Scan using pk_item on item (cost=0.00..7.41 rows=1 width=72)</td>
</tr>
<tr>
<td>Index Cond: (item.i_id = order_line.ol_i_id)</td>
</tr>
</tbody>
</table>

• Slowest form of join in theory
• But fast to produce first record
• In practice it's usually desirable for OLTP queries
• Performs very poorly if second child is slow
• Only join capable of executing CROSS JOIN
• Only join capable of inequality join conditions
Merge Joins

tpcc=> explain select * from order_line join item on (i_id = ol_i_id) ;

QUERY PLAN

-------------------------------------------------------------------------------------
Merge Join  (cost=58862.12..67288.43 rows=301231 width=142)
Merge Cond: (item.i_id = order_line.ol_i_id)
  ->  Index Scan using pk_item on item  (cost=0.00..3659.26 rows=100000 width=72)
  ->  Materialize  (cost=58861.10..62626.49 rows=301231 width=70)
       ->  Sort  (cost=58861.10..59614.18 rows=301231 width=70)
          Sort Key: order_line.ol_i_id
       ->  Seq Scan on order_line  (cost=0.00..6731.31 rows=301231 width=70)

• Can only be used for equality join conditions – and only for ordered data types
• Fastest join in theory, especially for large data sets
• Requires ordered inputs – which can require slow sorts or index scans
• Often ideal for data warehouse queries
• Startup can be slow, not desirable for OLTP queries
Hash Joins

```sql
tpcc=> explain select * from warehouse join district on (d_w_id = w_id);

QUERY PLAN

---------------------------------------------------------------------
Hash Join  (cost=1.02..2.26 rows=10 width=176)
  Hash Cond: (district.d_w_id = warehouse.w_id)
  ->  Seq Scan on district  (cost=0.00..1.10 rows=10 width=91)
  ->  Hash  (cost=1.01..1.01 rows=1 width=85)
    ->  Seq Scan on warehouse  (cost=0.00..1.01 rows=1 width=85)
```

- Can only be used for equality join conditions – and only for hashable data types
- Often ideal when joining a large table against a small table
- Slow to start due to hashing the second (usually smaller) table
- Can be especially slow if the estimate of the size of the tables is wrong
tpcc=> explain select * from s_partitions;

QUERY PLAN

Result  (cost=0.00..5362.00 rows=100200 width=361)
  ->  Append  (cost=0.00..5362.00 rows=100200 width=361)
    ->  Seq Scan on stock_partitions  (cost=0.00..12.00 rows=200 width=361)
    ->  Seq Scan on stock_0 s_partitions  (cost=0.00..535.00 rows=10000 width=315)
    ->  Seq Scan on stock_1 s_partitions  (cost=0.00..535.00 rows=10000 width=315)
    ->  Seq Scan on stock_2 s_partitions  (cost=0.00..535.00 rows=10000 width=315)
    ->  Seq Scan on stock_3 s_partitions  (cost=0.00..535.00 rows=10000 width=315)
    ->  Seq Scan on stock_4 s_partitions  (cost=0.00..535.00 rows=10000 width=315)
    ->  Seq Scan on stock_5 s_partitions  (cost=0.00..535.00 rows=10000 width=315)
    ->  Seq Scan on stock_6 s_partitions  (cost=0.00..535.00 rows=10000 width=315)
    ->  Seq Scan on stock_7 s_partitions  (cost=0.00..535.00 rows=10000 width=315)
    ->  Seq Scan on stock_8 s_partitions  (cost=0.00..535.00 rows=10000 width=315)
    ->  Seq Scan on stock_9 s_partitions  (cost=0.00..535.00 rows=10000 width=315)

• Produces unordered output which can reduce the available plans
• Adds overhead, especially if you have many partitions which can’t be eliminated
• Also used for UNION ALL and UNION
  • Warning: UNION must eliminate duplicates which requires a sort!
### Set Operations – Inheritance and Partitioning

```sql
tpcc=> explain select * from stock_partitions where s_w_id = 1 and s_i_id = 1;
```

**QUERY PLAN**

<table>
<thead>
<tr>
<th>Cost</th>
<th>Rows</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>11</td>
<td>361</td>
</tr>
</tbody>
</table>

**Result**

- **Append** (cost=0.00..95.71 rows=11 width=361)
  - **Seq Scan on stock_partitions** (cost=0.00..13.00 rows=1 width=361)
    - **Filter**: ((s_w_id = 1) AND (s_i_id = 1))
  - **Index Scan using pk_stock_0 on stock_partitions** (cost=0.00..8.27 rows=1 width=315)
    - **Index Cond**: ((s_w_id = 1) AND (s_i_id = 1))
  - **Index Scan using pk_stock_1 on stock_partitions** (cost=0.00..8.27 rows=1 width=315)
    - **Index Cond**: ((s_w_id = 1) AND (s_i_id = 1))
  - **Index Scan using pk_stock_2 on stock_partitions** (cost=0.00..8.27 rows=1 width=315)
    - **Index Cond**: ((s_w_id = 1) AND (s_i_id = 1))
  - **Index Scan using pk_stock_3 on stock_partitions** (cost=0.00..8.27 rows=1 width=315)
    - **Index Cond**: ((s_w_id = 1) AND (s_i_id = 1))
  - **Index Scan using pk_stock_4 on stock_partitions** (cost=0.00..8.27 rows=1 width=315)
    - **Index Cond**: ((s_w_id = 1) AND (s_i_id = 1))
  - **Index Scan using pk_stock_5 on stock_partitions** (cost=0.00..8.27 rows=1 width=315)
    - **Index Cond**: ((s_w_id = 1) AND (s_i_id = 1))
  - **Index Scan using pk_stock_6 on stock_partitions** (cost=0.00..8.27 rows=1 width=315)
    - **Index Cond**: ((s_w_id = 1) AND (s_i_id = 1))
  - **Index Scan using pk_stock_7 on stock_partitions** (cost=0.00..8.27 rows=1 width=315)
    - **Index Cond**: ((s_w_id = 1) AND (s_i_id = 1))
  - **Index Scan using pk_stock_8 on stock_partitions** (cost=0.00..8.27 rows=1 width=315)
    - **Index Cond**: ((s_w_id = 1) AND (s_i_id = 1))
  - **Index Scan using pk_stock_9 on stock_partitions** (cost=0.00..8.27 rows=1 width=315)
    - **Index Cond**: ((s_w_id = 1) AND (s_i_id = 1))
```
tpcc=> set constraint_exclusion = on;
SET

tpcc=> explain select * from stock_partitions where s_w_id = 1 and s_i_id = 1;

QUERY PLAN

Result  (cost=0.00..21.27 rows=2 width=361)
  ->  Append  (cost=0.00..21.27 rows=2 width=361)
      ->  Seq Scan on stock_partitions  (cost=0.00..13.00 rows=1 width=361)
          Filter: ((s_w_id = 1) AND (s_i_id = 1))
      ->  Index Scan using pk_stock_0 on stock_0 stock_partitions
          (cost=0.00..8.27 rows=1 width=315)
          Index Cond: ((s_w_id = 1) AND (s_i_id = 1))

• Make sure to set constraint_exclusion on if you have partition constraints set up
Not just for ORDER BY – also DISTINCT, GROUP BY, UNION, and merge joins
Sorts always have large startup times – bad for OLTP
If sort fits in \texttt{work\_mem} then it will use faster in-memory quicksort
Otherwise it will use slower external disk sort using temporary files
tpcc=> explain select s_i_id, count(*) from stock group by s_i_id;

GroupAggregate (cost=13652.32..15652.22 rows=99995 width=4)
  -> Sort (cost=13652.32..13902.31 rows=99995 width=4)
    Sort Key: s_i_id
  -> Seq Scan on stock (cost=0.00..5347.95 rows=99995 width=4)

(tpcc=> explain select s_i_id, count(*) from stock group by s_i_id;)

HashAggregate (cost=5847.93..7097.86 rows=99995 width=4)
  -> Seq Scan on stock (cost=0.00..5347.95 rows=99995 width=4)

(tpcc=> explain select s_i_id, count(*) from stock group by s_i_id;)

tpcc=> explain select distinct s_i_id from stock;

Unique (cost=13652.32..14152.29 rows=99995 width=4)
  -> Sort (cost=13652.32..13902.31 rows=99995 width=4)
    Sort Key: s_i_id
  -> Seq Scan on stock (cost=0.00..5347.95 rows=99995 width=4)
tpcc=> explain analyse select * from stock limit 10 offset 10;
QUERY PLAN

Limit (cost=0.53..1.07 rows=10 width=315) (actual time=0.061..0.113 rows=...)
  ->  Seq Scan on stock (cost=0.00..5347.95 rows=99995 width=315)
       (actual time=0.018..0.054 rows=20 loops=1)

• Limit handles both LIMIT and OFFSET
• Limit can also be used for min() and max() if there's no where clause
• Records skipped for OFFSET must still be generated and then thrown out!

• Note that the cost of child scan is still the full cost
• However the actual time spent reflects the time saved due to the limit

• Sort combined with Limit can use an optimized form of sort
Subplans for Subqueries

tpcc=# explain select (select count(*) from item where i_id = empty.i_id) from empty;

QUERY PLAN
--------------------------------------------------------------------------
Seq Scan on empty (cost=0.00..19933.22 rows=2400 width=4)
SubPlan
  -> Aggregate (cost=8.28..8.29 rows=1 width=0)
    -> Index Scan using pk_item on item (cost=0.00..8.28 rows=1 width=0)
       Index Cond: (i_id = $0)

----------------------------------------------------------------------------

tpcc=# explain select (select count(*) from item) from empty;

QUERY PLAN
--------------------------------------------------------------------------
Seq Scan on empty (cost=2522.01..2556.01 rows=2400 width=0)
InitPlan
  -> Aggregate (cost=2522.00..2522.01 rows=1 width=0)
    -> Seq Scan on item (cost=0.00..2272.00 rows=100000 width=0)
tpcc=> explain analyze select * from empty join item using (i_id);

QUERY PLAN

Hash Join (cost=4694.00..5957.00 rows=2400 width=72)
  (actual time=0.008..0.008 rows=0 loops=1)
  Hash Cond: (empty.i_id = item.i_id)
  ->  Seq Scan on empty (cost=0.00..34.00 rows=2400 width=4)
      (actual time=0.003..0.003 rows=0 loops=1)
  ->  Hash (cost=2272.00..2272.00 rows=100000 width=72) (never executed)
      ->  Seq Scan on item (cost=0.00..2272.00 rows=100000 width=72) (never executed)
Total runtime: 0.136 ms

Postgres only generates the hash if it's needed to match any records
Also often happens in Nested Loop joins, Merge Joins
Can also happen for subqueries used in select target lists
Real World Problems

- **Estimates are inaccurate**
  - Have you analysed recently?
  - Are your tables empty? Postgres falls back to a heuristic.
  - Are your columns strongly correlated?
  - Are your clauses written like
    WHERE i+0 = val or lower(t) = 'foo'?

- **Not using an available index**
  - Are you sure using the index would actually be helpful?
  - Are you using LIKE? Is your index using text_pattern_ops?

- **Mysterious time sinks**
  - Triggers? Do you have indexes on foreign keys?
  - Dead tuples? Have you vacuumed recently?
Asking for Help

- State your PostgreSQL version
- Make sure you have vacuumed and analysed appropriately
- Always include EXPLAIN ANALYSE output
- Include queries/tables/data when possible

pgsql-performance@postgresql.org

Thanks
Robert Treat, Greg Sabino Mullane, AndrewSN@postgresql, Magnifikus@postgresql, Bryan Encina, Neil Conway