https://robots.thoughtbot.com/reading-an-explain-analyze-query-plan
See also:
https://wiki.postgresql.org/wiki/CTEReadme#Generating_a_plan
https://dba.stackexchange.com/questions/209138/whats-the-big-o-computational-complexity-of-a-postgresql-recursive-common-tab

Reading a Postgres EXPLAIN ANALYZE Query Plan

robots.thoughtbot.com/reading-an-explain-analyze-query-plan

The most powerful tool at our disposal for understanding and optimizing SQL queries is `EXPLAIN ANALYZE`, which is a Postgres command that accepts a statement such as `SELECT ...`, `UPDATE ...`, or `DELETE ...`, executes the statement, and instead of returning the data provides a query plan detailing what approach the planner took to executing the statement provided.

Here's a query pulled from the Postgres Using EXPLAIN page:

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2 ORDER BY t1.fivethous;

                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------
-------------------------------------------------
 Sort  (cost=717.34..717.59 rows=101 width=488) (actual time=7.761..7.774 rows=100 loops=1)
   Sort Key: t1.fivethous
   Sort Method: quicksort  Memory: 77kB
   ->  Hash Join  (cost=230.47..713.98 rows=101 width=488) (actual time=0.711..7.427 rows=100
loops=1)
         Hash Cond: (t2.unique2 = t1.unique2)
         ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244) (actual
time=0.007..2.583 rows=10000 loops=1)
         ->  Hash  (cost=229.20..229.20 rows=101 width=244) (actual time=0.659..0.659 rows=100
loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 28kB
               ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20 rows=101 width=244)
(actual time=0.080..0.526 rows=100 loops=1)
                     Recheck Cond: (unique1 < 100)
                     ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101
width=0) (actual time=0.049..0.049 rows=100 loops=1)
                           Index Cond: (unique1 < 100)
 Planning time: 0.194 ms
 Execution time: 8.008 ms
```

Postgres builds a tree structure of plan nodes representing the different actions taken, with the root and each `->` pointing to one of them. In some cases `EXPLAIN ANALYZE` provides additional execution statistics beyond the execution times and row counts, such as `Sort` and `Hash` above. Any line other than the first without an `->` is such information, so the structure of the query is:

```
Sort
└── Hash Join
    ├── Seq Scan
    └── Hash
        └── Bitmap Heap Scan
            └── Bitmap Index Scan
```

Each tree's branches represent sub-actions, and you'd work inside-out to determine what's happening "first" (though the order of nodes at the same level could be different).

The first thing done is a `Bitmap Index Scan` on the `tenk_unique1` index:

```
->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0) (actual
time=0.049..0.049 rows=100 loops=1)
      Index Cond: (unique1 < 100)
```

This corresponds to the SQL `WHERE t1.unique1 < 100`. Postgres is finding the locations of the rows matching the index condition `unique1 < 100`. The rows themselves aren't being returned here. The cost estimate `(cost=0.00..5.04 rows=101 width=0)` means that Postgres expects that it will "cost" 5.04 of an arbitrary unit of computation to find these values. The 0.00 is the cost at which this node can begin working (in this case, just startup time for the query). `rows` is the estimated number of rows this Index Scan will return, and `width` is the estimated size in bytes of the returned rows (0 because we only care about the location, not the content of the rows).

Because we ran `EXPLAIN` with the `ANALYZE` option, the query was actually executed and timing information was captured. `(actual time=0.049..0.049 rows=100 loops=1)` means that the index scan was executed 1 time (the `loops` value), that it returned 100 rows, and that the actual time was 0.. In the case of a node executed more than once, the actual time is an average of each iteration and you would multiply the value by the number of loops to get real time. The range values may also differ which gives an idea of min/max times spent. This establishes a ratio for the costs that each cost unit of 0.049ms / 5.04 units ≈ 0.01ms/unit for this query.

The results of the Index Scan are passed up to a `Bitmap Heap Scan` action. In this node, Postgres is taking the locations of the rows in the tenk1 table, aliased as t1, where `unique1 < 100` and fetching the rows from the table itself.

```
->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20 rows=101 width=244) (actual
time=0.080..0.526 rows=100 loops=1)
      Recheck Cond: (unique1 < 100)
```

We can see that the cost expectations, when multiplied by the 0.01 value we calculated, would mean a rough expected time of (229.20 - 5.07) * 0.01 ≈ 2.24ms , and we see an actual time of 0.526ms per row, which is off by a factor of 4. This may be because the cost estimate is an upper bound and not all rows needed to be read, or because the recheck condition is always true.

The combination of `Bitmap Index Scan` and `Bitmap Heap Scan` is much more expensive than reading the rows sequentially from the table (a `Seq Scan`), but because relatively few rows need to be visited in this case the two step process ends up being faster. This is further sped by sorting the rows into physical order before fetching them, which minimizes the cost of separate fetches. The "bitmap" in the node names does the sorting.

The results of the heap scan, those rows from tenk1 for which `unique1 < 100` is true, are inserted into an in-memory Hash table as they are read. As we can see by the costs, this takes no time at all.

```
->  Hash  (cost=229.20..229.20 rows=101 width=244) (actual time=0.659..0.659 rows=100 loops=1)
      Buckets: 1024  Batches: 1  Memory Usage: 28kB
```

The Hash node includes information about number of hash buckets and batches, as well as peak memory usage. If Batches > 1 there's also disk usage involved, but that is not shown. The memory usage makes sense at 100 rows * 244 bytes = 24.4 kB , which is close enough to the 28kB for our purposes that we can assume it's the memory taken by the Hash keys themselves.

Next, Postgres reads all 10000 rows from tenk2 (aliased as t2) and checks them against the Hash of tenk1 rows. Hash Join means that the rows of one table are entered into an in-memory hash (which we've built up to so far), after which the rows of another table is scanned and its values probed against the hash table for

matches. We see the conditions of the “match” on the second line, `Hash Cond: (t2.unique2 = t1.unique2)`. Note that because the query is selecting all values from both tenk1 and tenk2, the width of each row doubles during the Hash Join.

```
->  Hash Join  (cost=230.47..713.98 rows=101 width=488) (actual time=0.711..7.427 rows=100
loops=1)
      Hash Cond: (t2.unique2 = t1.unique2)
      ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244) (actual
time=0.007..2.583 rows=10000 loops=1)
```

Now that all rows that meet our conditions have been collected, we can sort the result set by the `Sort Key: t1.fivethous`.

```
Sort  (cost=717.34..717.59 rows=101 width=488) (actual time=7.761..7.774 rows=100 loops=1)
  Sort Key: t1.fivethous
  Sort Method: quicksort  Memory: 77kB
```

The Sort node includes information about the algorithm used to sort, `quicksort`, whether the sort was done in memory or on disk (which greatly effects speed), and the amount of memory/disk space needed.

Understanding how to read query plans is great for optimizing queries. For example, Seq Scan nodes often indicate an opportunity for an index to be added, which is much faster to read. Familiarizing yourself with these plans will make you a better database engineer. For more examples of query plans, read Using `EXPLAIN` from which this example was taken.

Now that you’re able to read a query plan, learn to optimize performance.