6.814 - Problem Set 2

Long Nguyen; Collaborator: Joseph Lin

October 16, 2017

Part I - Query Plans & Cost Models

1. Using the command `\d movies`, I retrieved the following indexes for `movies`.

```
Indexes:
"movies_pkey" PRIMARY KEY, btree (id)
"movies_title" btree (title)
```

2. Postgres chooses the following sequential scan query for `select title from movies`.

```
QUERY PLAN
----------------------------------------------------------------
Seq Scan on movies  (cost=0.00..8418.65 rows=464065 width=18)
(1 row)
```

   This plan is not any different from the plan on the previous page, but given the `btree` index defined on `movies`, we could have chosen another possible query plan, one based on the B-tree.

3. I ran `explain select title from movies order by title;` and got the following output.

```
QUERY PLAN
-----------------------------------------------------------------------------------------
Index Only Scan using movies_title on movies  (cost=0.42..15677.40 rows=464065 width=18)
(1 row)
```

   The previous plan retrieves all movie titles sequentially, but the plan in this query must retrieve movies in order, so it accesses *only* on the B-tree index to get sorted titles.

4. The command `explain select title, year from movies order by title` yielded the following output.

```
QUERY PLAN
------------------------------------------------------------------------------------
Index Scan using movies_title on movies  (cost=0.42..30786.82 rows=464065 width=22)
(1 row)
```

   In this query, we don't have an index-*only* scan. This is because while we are still sorting by `title`, we need to also retrieve `year` from the heap. The previous question made use of a scan that would only refer to the B-tree, conveniently because `title` was the only criterion we had for selection.

5. Because `people_reduced` does not have the extra column with type `character(20000)`, a sequential scan is the best option, as there is no I/O overhead in reading so much more data, and sequentially scanning only 1000 rows is going to be more efficient than using the index (random access overhead).

   On the other hand, `people_wide` has the extra column, so performing a sequential scan, while sensible to get all the names without sorting, makes little sense if I/O operations are going to be wasted on the extra field, especially if each can be up to 20000 characters long. Thus, using only the B-tree index to scan for the names is going to be cheaper in this case.

6. First, I ran sequential and index-only scans on the query for `people_reduced`. Their outputs are below.

```
QUERY PLAN
----------------------------------------------------------------
Seq Scan on people_reduced  (cost=0.00..18.00 rows=1000 width=14)
(actual time=0.050..0.271 rows=1000 loops=1)
Planning time: 0.029 ms
Execution time: 0.338 ms

QUERY PLAN
----------------------------------------------------------------
Index Only Scan using people_reduced_name on people_reduced  (cost=0.28..43.27 rows=1000 width=14)
(actual time=0.099..0.294 rows=1000 loops=1)
Heap Fetches: 0
Planning time: 0.060 ms
Execution time: 0.369 ms
```

As we see, indeed, the sequential scan on `people_reduced` was the right plan, and the query optimizer was correct. Now, I run sequential and index-only scans on the query for `people_wide`.

```
QUERY PLAN
----------------------------------------------------------------
Seq Scan on people_wide  (cost=0.00..50.00 rows=1000 width=14)
(actual time=0.043..0.562 rows=1000 loops=1)
Planning time: 0.024 ms
Execution time: 0.630 ms

QUERY PLAN
----------------------------------------------------------------
Index Only Scan using people_wide_name on people_wide  (cost=0.28..43.27 rows=1000 width=14)
(actual time=0.087..0.260 rows=1000 loops=1)
Heap Fetches: 0
Planning time: 0.037 ms
Execution time: 0.326 ms
```

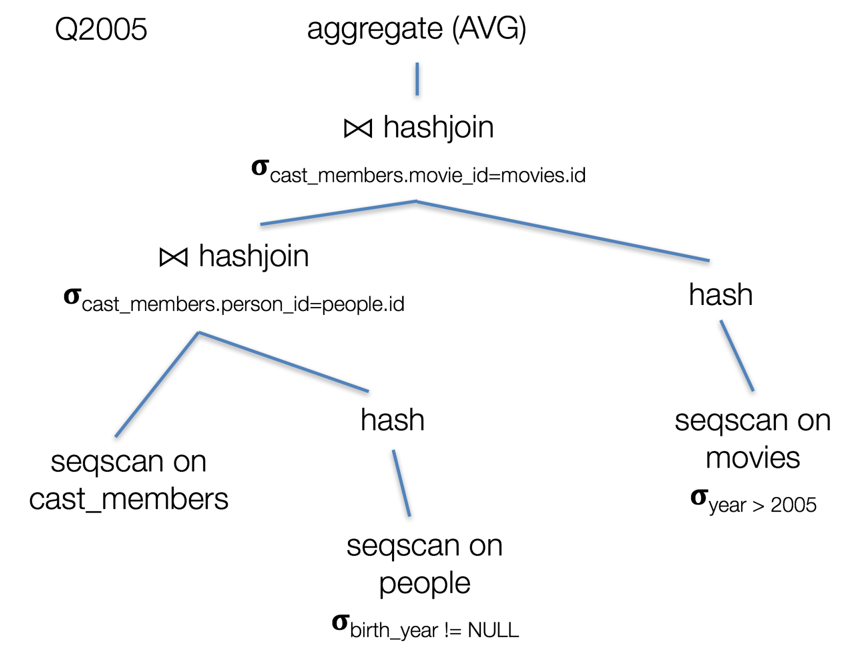
We see that the query optimizer was right again, and by more than its estimated costs. There is only a $50 - 43.27$ difference in estimated cost, but the runtime of a sequential scan on `people_wide` is almost double that of an index-only scan.

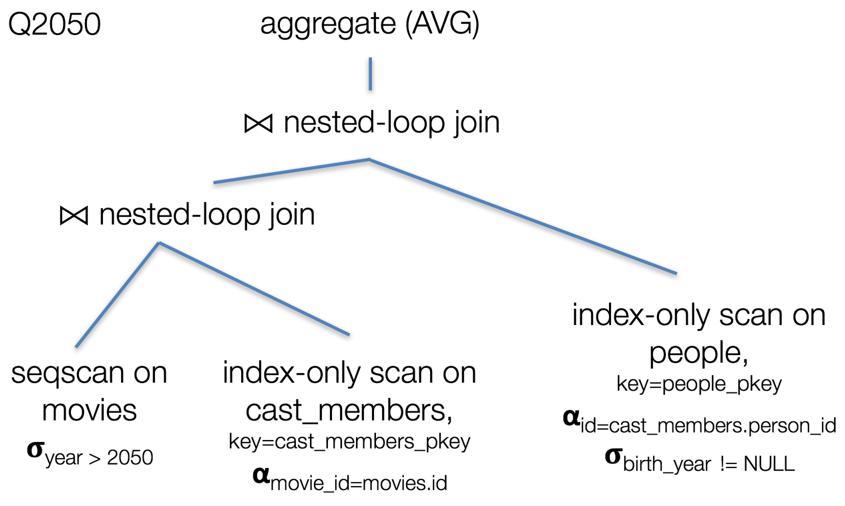
7. We have to keep in mind that Postgres's query planner/optimizer isn't a perfect proxy for how our query will actually run. The optimizer makes certain assumptions about the hardware (in this case, the Athena cluster) and the row estimates are only based off of selectivity functions run on the samples chosen. Thus, the huge discrepancy between the estimated costs of the two queries is large and valid, and they don't necessarily correlate completely to actual query execution.

   As for the actual costs, it makes sense when considering how many predicate operations we have to perform respectively for each one. We have to filter out 45 rows for the second query, whereas we only have to filter out 10 for the first. Flushing these out to the user is also dependent on how many results we got.

   Verbatim from the `EXPLAIN` documentation provided in the link given in the problem set document: "In particular, the cost does not consider the time spent transmitting result rows to the client, which could be an important factor in the real elapsed time."

8. Below are the full diagrams for both Q2005 and Q2050.





9. Refer to 8.
10. Refer to 8.
11. On the hashjoin between `cast_members` and `people` in Q2005, I notice that PSQL expected 548809 result rows from the join, but the actual amount turned out to be 1391484, more than double that.

    Also, on all the scans and nested-loop joins in Q2050, I notice that PSQL overshot its estimates of output cardinality by way more than factors of 2. For example, the nested-loop join between `movies` and `cast_members` produced *only 6 rows* when PSQL predicted 165.
12. The query planner changes from hash joins to nested-loop joins starting at the year 2018. This is the only point at which it changes plans. It does so because our filter becomes increasingly selective. When querying at the very beginning, `WHERE year > 1893`, we are looking over the *entire dataset*, so hashing and then joining is much more efficient than using nested loops.

    More quantitatively, let us consider how the selectivity changes. We use . . . for brevity.

    Consider the filter `WHERE year > 1893`, where we expect to have 508297 output rows, the *entire dataset.*

```
                                      QUERY PLAN
-------------------------------------------------------------------------------------
 Aggregate  (cost=141886.36..141886.37 rows=1 width=4)
   ->  Hash Join  (cost=35967.04..140615.62 rows=508297 width=4)
         Hash Cond: (cast_members.movie_id = movies.id)
         ->  Hash Join  (cost=21015.61..114407.12 rows=548809 width=14)
               Hash Cond: (cast_members.person_id = people.id)
               ->  Seq Scan on cast_members  (cost=0.00..54568.21 rows=3333521 width=20)
         ...
```

Now consider 1955, between 2018 and 1893. We still have 418333 rows of expected output, so the hash join is still valid.

```
                                      QUERY PLAN
-------------------------------------------------------------------------------------
 Aggregate  (cost=138438.88..138438.89 rows=1 width=4)
   ->  Hash Join  (cost=35016.12..137393.04 rows=418333 width=4)
         Hash Cond: (cast_members.movie_id = movies.id)
         ->  Hash Join  (cost=21015.61..114407.12 rows=548809 width=14)
               Hash Cond: (cast_members.person_id = people.id)
               ->  Seq Scan on cast_members  (cost=0.00..54568.21 rows=3333521 width=20)
               ->  Hash  (cost=18744.75..18744.75 rows=181669 width=14)
         ...
```

Now, the breakpoint. 2017 expects 7955 output rows.

```
                                      QUERY PLAN
-------------------------------------------------------------------------------------
 Aggregate  (cost=58245.99..58246.00 rows=1 width=4)
   ->  Hash Join  (cost=21016.04..58226.10 rows=7955 width=4)
         Hash Cond: (cast_members.person_id = people.id)
         ->  Nested Loop  (cost=0.43..36647.71 rows=48322 width=10)
            ...
```

Now transition to 2018, where the expected output cardinality has decreased *significantly* to 496.

```
                                      QUERY PLAN
-------------------------------------------------------------------------------------
 Aggregate  (cost=12949.63..12949.64 rows=1 width=4)
   ->  Nested Loop  (cost=0.86..12948.39 rows=496 width=4)
         ->  Nested Loop  (cost=0.43..11528.11 rows=3010 width=10)
               ->  Seq Scan on movies  (cost=0.00..9578.81 rows=419 width=10)
                     Filter: (year > 2018)
         ...
```

At this point, is it really worth it to hash and then join anymore, when we have such little expected cardinality? It's better now to brute-force our way through with nested-loop joins, which may even prove more efficient than the former.

Let's see if the query planner is actually switching correctly. We run `EXPLAIN ANALYZE` on this switch point.

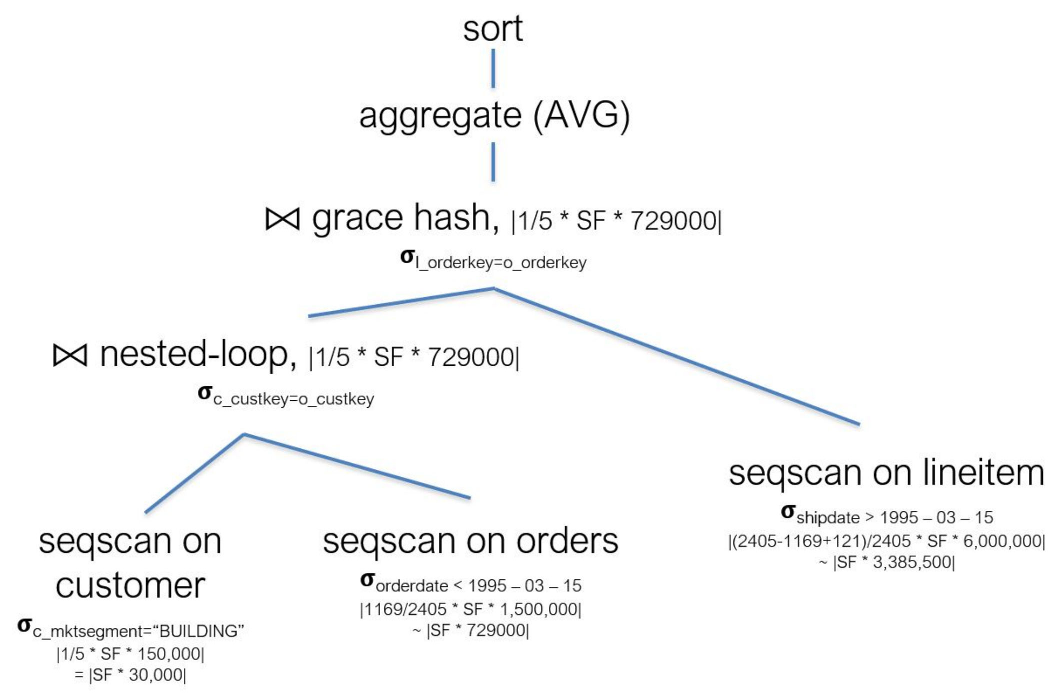
```
> 2017
 Aggregate  (cost=58245.99..58246.00 rows=1 width=4) (actual time=423.223..423.223 rows=1 loops=1)
   ->  Hash Join  (cost=21016.04..58226.10 rows=7955 width=4) (actual time=199.922..421.803 rows=9251 loops=1)
         Hash Cond: (cast_members.person_id = people.id)

> 2018
 Aggregate  (cost=12949.63..12949.64 rows=1 width=4) (actual time=127.325..127.325 rows=1 loops=1)
   ->  Nested Loop  (cost=0.86..12948.39 rows=496 width=4) (actual time=26.743..127.086 rows=851 loops=1)
```

Indeed, the actual output cardinality decreases significantly by more than a factor of 10 within just this one transition. The query planner was right in switching to nested loops.

Part II - Query Plans & Access Methods

13. If we have no indexes, then using a nested-loop join on `customer` and `orders` will benefit us greatly, since the inner table `customer` is only of cardinality $SF \cdot 150,000$. Then, once we've done this join, we'll use the grace hash join between it and `lineitem`, which is far too large to fit into memory.

    The image of my proposed query plan is below.


sort
aggregate (AVG)
⋈ grace hash, |1/5 * SF * 729000|
σ l_orderkey=o_orderkey
⋈ nested-loop, |1/5 * SF * 729000|
σ c_custkey=o_custkey
seqscan on customer
σ c_mktsegment="BUILDING"
|1/5 * SF * 150,000|
= |SF * 30,000|
seqscan on orders
σ orderdate < 1995 – 03 – 15
|1169/2405 * SF * 1,500,000|
~ |SF * 729000|
seqscan on lineitem
σ shipdate > 1995 – 03 – 15
|(2405-1169+121)/2405 * SF * 6,000,000|
~ |SF * 3,385,500|

    Assuming uniform distributions, we should expect to have 1/5th of `customer` filtered from our scan with all the market segments. Likewise, there are 1169 days from the beginning of the range until March 15, 1995 (out of 2405 possible days in the range), and calculations were made accordingly. Then, we assume again uniformity in the sense that 1/5th of `orders` belongs to the 1/5th customers we filtered out for. Thus, we are bounded by $1/5 \cdot SF \cdot 729000$.

    A note: *a nested-loop join could've worked for the final join, because our expected output is only 412Mb and can still fit into memory, but only if our uniform distribution assumption actually holds in practice. For generality, I chose the grace hash join instead.*

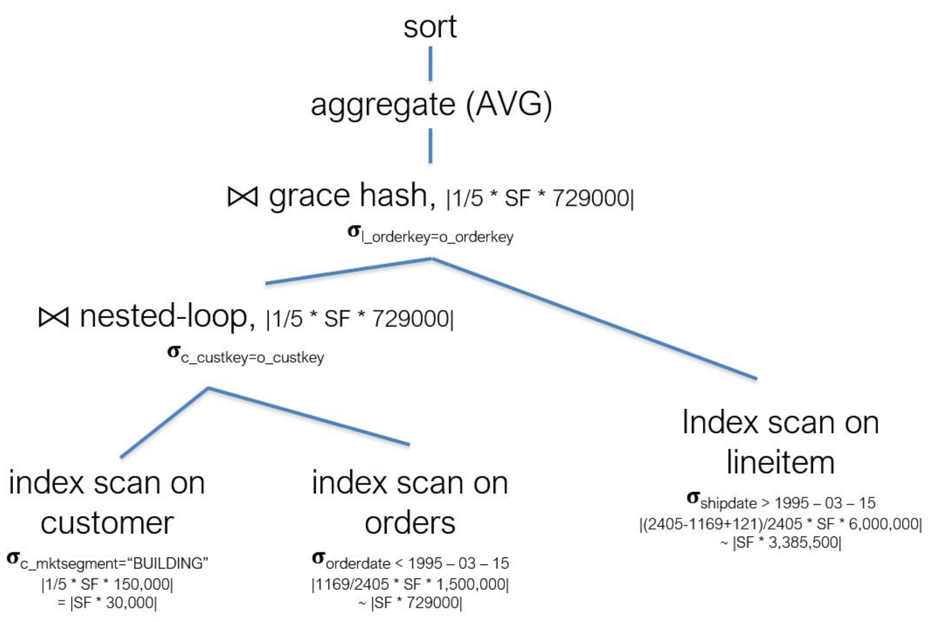
14. For this query plan, we first sequentially scan `customer`, so that's 268.5Mb. Then, we put `customer` in our 1Gb memory to make our nested-loop join quicker. Now, we read `orders` (with the predicate filtering too) for the nested-loop join, and this is 1560Mb. Finally, we perform grace hash on our intermediate join result (412Mb) and `lineitem` (6720Mb), so we must scan `lineitem` and our previous output of the nested-loop join (formulas as taken from lecture slides).

    Altogether, this is
$$\frac{268.5 + 1560 + 3(412 + 6720)}{100} = 232 \text{ seconds} = 3.9 \text{ minutes}.$$

15. Now that we are only concerned with efficiency and don't care about insert time, we can use clustered indexes to save ourselves the need of sequentially scanning each table in its entirety. We cluster each heap file based on the useful rangescan that we need to perform: thus, cluster `lineitem` by `shipdate`, `customer` by `mktsegment`, and `orders` by `orderdate`. This gets rid of the entire overhead that we have in needing to scan each entire table during our joins, and now, we only need to do a disk seek and then read sequentially all this data that meets our criteria. This will help with efficiency.

16. For this query, I chose not to modify anything related to the join algorithms. We now have the choice of using index-based nested-loop joins, but I felt that sticking with the initial nested-loop join was the better option, since we're still just reading a small amount of data and either would probably work. As for the grace hash join, we still have $> 3,000,000$ results expected from the scan on `lineitem`, so it's probably best to just stick to the old plan where it may not fit into memory.

The picture for this query is located below.


sort
aggregate (AVG)
⋈ grace hash, |1/5 * SF * 729000|
σ l_orderkey=o_orderkey
⋈ nested-loop, |1/5 * SF * 729000|
σ c_custkey=o_custkey
index scan on customer
σ c_mktsegment="BUILDING"
|1/5 * SF * 150,000|
= |SF * 30,000|
index scan on orders
σ orderdate < 1995 – 03 – 15
|1169/2405 * SF * 1,500,000|
~ |SF * 729000|
Index scan on lineitem
σ shipdate > 1995 – 03 – 15
|(2405-1169+121)/2405 * SF * 6,000,000|
~ |SF * 3,385,500|

17. With our newly generated indexes, we don't have to scan the entire table sequentially to filter out what we want. Instead we can simply do a rangescan on all three tables with our B+ tree indexes. We look only at the `mktsegment='BUILDING'` part of `customer`, and so on regarding dates for the other two tables. Thus, our runtime is around

$$\frac{\frac{1}{5}\cdot 268.5+\frac{1169}{2405}\cdot 1560+3(412+\frac{1357}{2405}\cdot 6720)}{100} = 134 \text{ seconds} = 2.2 \text{ minutes}.$$

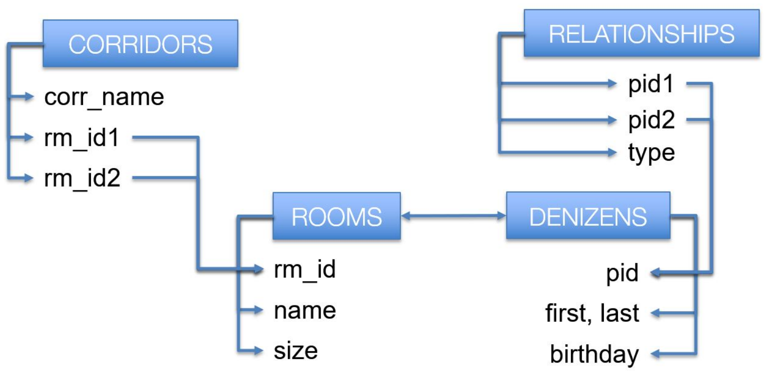
Part III - Schema Design and Query Execution

18. We make some assumptions. Assume that `roomname` is unique, that each `(firstname, lastname)` key is unique, that room pairs are connected by at most one corridor (it would be weird if a room had a backdoor that led to another corridor, etc.), and that relationships between pairs of people can only be one type (philosophically, you can be friends and enemies with a person, but practically, we assume you may only be one). Also, corridors are unique in that they have unique identifiers (like name, or ID, etc.).

    Now, we consider functional dependencies.

    - room → room size
    - (first, last) → birthday
    - relationship/pair → type of relationship

19. Consider the ER diagram below.



I drew it this way for a few reasons:

- Double-sided arrow between DENIZENS and ROOMS because the problem mentions that rooms may have 0 or more, and denizens may reside in 0 or more.
- Relationships consist of two people, which directly relate to the DENIZENS people.
- Corridors uniquely connect two rooms together, and this relates to the ROOMS table.

20. Following the ER diagram drawn previously, my BCNF schema is as follows: `rooms(rid, rname, rsize)`, `corridors(cname, rmid1, rmid2)`, `denizens(pid, first, last, bday)`, `relationships(pid1, pid2, type)`, and `residents(rid, pid)`.

    For `rooms`, primary key is `rid`, secondary key is `rname`.

    For `corridors`, primary key is `cname`.

    For `denizens`, primary key is `pid`, secondary key is `(first, last)`.

    For `relationships`, primary key is `(pid1, pid2)`.

    For `residents`, the primary key is `(rid, pid)`.

21. My schema is indeed redundancy and anomaly-free. For every FD mentioned previously, the left side is a key of its respective table. Also, by introducing the `residents` table, we capture the many-to-many relationship between `denizens` and `rooms` without being vulnerable to update anomalies. We can delete denizens and rooms from their respective tables while still maintaining accurate information regarding corridors and relationships between denizens.

22. If we wanted to ensure that each denizen has more friends (or be evicted otherwise), we can do a check on insertion. For example, we could query to check for the amount of friends and enemies a person has (easy queries on the `relationships` table), and if it's satisfied, insert. The constraint is thus enforced by construction, and proof by induction on the ith insertion is left beyond the scope of this problem.

23. If each denizen should live in exactly one room, then it's simple: we drop the `residents` table, and we add a new field `livingroom` field to the `denizens` table schema. Now, every denizen has a room associated with them.

    A view for this with backwards compatibility would be to generate the new table `denizens'` as part of the view, where the `livingroom` field is just one chosen arbitrarily from however many rooms a denizen lived in under the old schema.

    This works, but it obviously can't handle when we want to specify exactly what room the denizen could have lived in out of x rooms the denizen lived in under the old schema. `denizens'` can only take one of these.