CENG352 Written Assignment 2

Mustafa Ozan Alpay

23/05/2021

1 Query Processing

1.1 Part 1

1.1.1 Question a

This query aims to find the maximum net worth per age for actors that are 30 or younger, if there are more than 2 actors for that age group.

If we have the search key (age, net_worth), that would mean we have all the information we would need to evaluate this query through indexes. If we had the key on age only, we would need to access the data to calculate the net_worth anyway. Therefore the minimal search key for B+ tree index that would allow this query to be evaluated using index-only plan is (age, net_worth).

1.1.2 Question b

```sql
SELECT year, COUNT(distinct *)
FROM Movie
WHERE budget > 1000000
GROUP BY year;
```

Query 1

Query 1 can be evaluated with index-only plan if there is an index on (year, budget), since the query compares budget values and groups by year, and finally counts distinct values. Since each of these comparisons are based on the index, it can be achieved without actually reading the data, but only referring to the indexes.

```sql
SELECT year, AVG(budget)
FROM Movie
WHERE year > 2010 AND genre = 'Horror'
GROUP BY year;
```

Query 2

However,Query 2 needs access to the genre field which is not stored as a key. That would cause the DBMS to read the actual data while evaluating the query, therefore cannot be executed through the index-only plan on the composite index of (year, budget).

1.2 Part 2

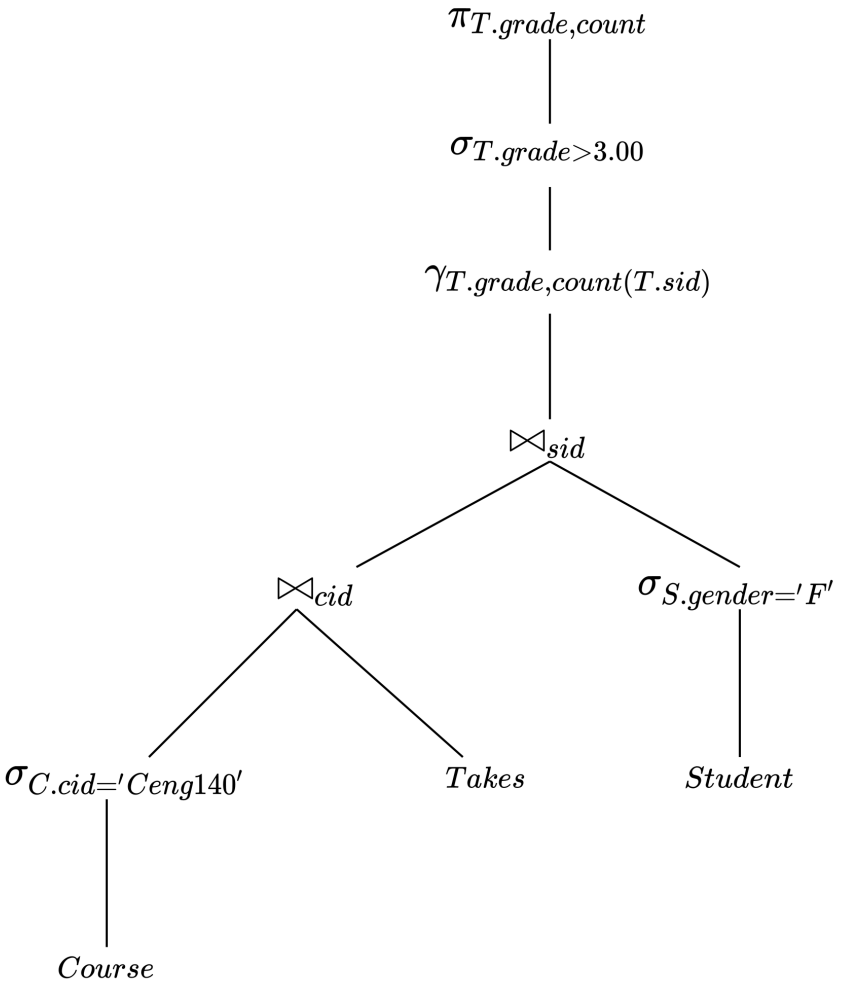

Figure 1: Logical Plan for Query 1.2.1

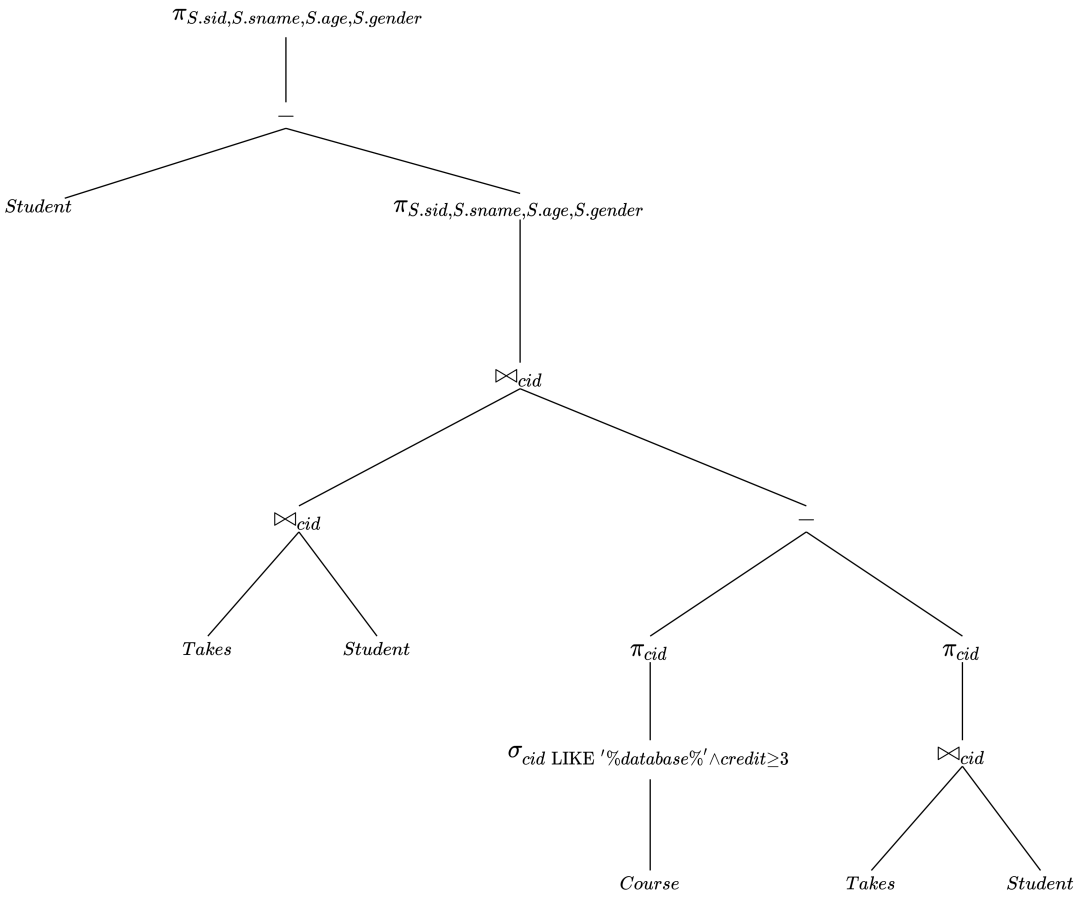

Figure 2: Logical Plan for Query 1.2.2

2 Join Algorithms

- $T(R) = 40000$
- $T(S) = 8000$
- $B(R) = 4000$
- $B(S) = \lceil 8000/15 \rceil = 534$
- Since S.y is a primary key, $V(S, y) = 8000$
- $M = 62$

2.1 Question a

$$\begin{aligned} \text{Cost} &= B(R) + \frac{B(R) \cdot B(S)}{M-2} \\ &= 4000 + \frac{4000 \cdot 534}{60} \\ &= 39600 \end{aligned}$$

2.2 Question b

$$\begin{aligned}\text{Cost} &= B(S) + \frac{B(S) \cdot B(R)}{M-2} \\ &= 534 + \frac{534 \cdot 4000}{60} \\ &= 36134\end{aligned}$$

2.3 Question c

Since $B(R) + B(S) \leq M$ does not hold, this join cannot be completed in one pass. If $B < M^2$ holds, the sorting is two-pass. Since $B(R) = 4000 < M^2 = 3844$ does not hold, the sorting can be completed in three passes. However, for S, $B(S) = 534 < M^2 = 3844$ holds, thus the sorting for S can be completed in two passes.

$$\begin{aligned}M_1 &= \lceil B(R)/M \rceil \\ &= \lceil 4000/62 \rceil \\ &= 65 \\ M_{11} &= \lceil M_1/62 \rceil \\ &= 2 \\ M_1 &= \lceil B(S)/M \rceil \\ &= \lceil 534/62 \rceil \\ &= 9\end{aligned}$$

Since $M_{11} + M_2 \leq M$we can sort-merge join.

- Step 1a: Generate initial runs for R
  Since R does not fit into two-passes, the cost would increase due to having three passes: Read + write + read + write + read + write + read = $7B(R)$
  In the first run, we sort $M - 1 = 61$ runs, which would give us a new $M = 62$ run. Later, by using this new run and the remaining runs ($\lceil (4000 - 61 * 62 = 218)/62 \rceil = 4$), we create a final run and complete the sorting process. All sorts happen in th memory, while the sorted pages get stored in the disk.
- Step 1b: Generate initial runs for S
  Since S fits into two-passes, the cost of the initial run: Read + write + read = $3B(S)$
  We sort $M - 1 = 61$ runs, which would give us a new $M = 62$ run.
- Step 2: Merge and join
  Total cost: $7B(R) + 3B(S)$
  The merge process uses $M - 1 = 61$ pages for input and it uses the remaining 1 page for output.

$$\begin{aligned}\text{Cost} &= 7 \cdot B(R) + 3 \cdot B(S) \\ &= 7 \cdot 4000 + 3 \cdot 534 \\ &= 28534\end{aligned}$$

2.4 Question d

To have a partitioned hash join, we need the following requirement: $min(B(R), B(S)) \leq M^2$.

1. Read relation R one at a time and hash into $M - 1 = 61$ buckets, and use 1 bucket as input buffer. When a bucket fills up, flush it to disk. In the end, we get relation R back on disk split into 61 buckets.

2. Read relation S one at a time and hash into $M - 1 = 61$ buckets, and use 1 bucket as input buffer. When a bucket fills up, flush it to disk. In the end, we get relation S back on disk split into 61 buckets.

3. Read one partition of S and create hash table in memory using a different hash function. Use 1 bucket as input buffer, 1 bucket as output buffer, and $M - 2 = 60$ buckets for hash.

4. Scan matching partition of R and probe the hash table.

5. Repeat for each buckets, and complete the join.

$$\begin{aligned}\text{Cost} &= 3 \cdot B(R) + 3 \cdot B(S) \\ &= 3 \cdot 4000 + 3 \cdot 534 \\ &= 13602\end{aligned}$$

2.5 Question e

The index-nested loop join works as the following:

- Considering the join, R $\bowtie$ S
- S has an index on the join attribute (y)
- Iterate over R, for each tuple fetch corresponding tuple(s) from S

$$\begin{aligned}\text{Clustered Cost} &= B(R) + \frac{T(R) \cdot B(S)}{V(S, y)} \\ &= 4000 + \frac{40000 \cdot 534}{8000} \\ &= 6670 \\ \text{Unclustered Cost} &= B(R) + \frac{T(R) \cdot T(S)}{V(S, y)} \\ &= 4000 + \frac{40000 \cdot 8000}{8000} \\ &= 44000\end{aligned}$$

2.6 Question f

- S has clustered index on z:

$$\begin{aligned}\text{Cost} &= \frac{B(R)}{V(R, z)} \\ &= \frac{4000}{20} \\ &= 200\end{aligned}$$

- S has unclustered index on z:

$$\begin{aligned}\text{Cost} &= \frac{T(R)}{V(R, z)} \\ &= \frac{40000}{20} \\ &= 2000\end{aligned}$$

- S has no index on z:
  Since we need to do a full sequential scan,

$$\begin{aligned}\text{Cost} &= B(R)\\ &= 4000\end{aligned}$$

I would choose the clustered index, since it provides the lowest cost, it might be the most beneficial one. However if the selectivity factor ($V(R, z)$) is susceptible to changes, I might rethink the strategy, since it severly affects the cost. For example a decrease in that would make the unclustered one perform the poorest. However, as long as it's greater than 1, it is guaranteed that the clustered index will perform better than the no index solution, so it is preferred.

3 Physical Query Plan

3.a Index Scan

Since Tweet has unclustered index on hashtag, the cost of index based selection on equality is

$$\begin{aligned}\text{Cost} &= \frac{T(T)}{V(T, hashtag)}\\ &= \frac{600000}{50000}\\ &= 12\\ \text{Result Size} &= \frac{T(T)}{V(T, hashtag)}\\ &= \frac{600000}{50000}\\ &= 12\end{aligned}$$

3.b Clustered Index Join

Joining $T \bowtie R$ using index nested loop join, since R has a clustered index on hashtag,

$$\begin{aligned}\text{Cost} &= B(T) + \frac{T(T) \cdot B(R)}{V(R, hashtag)}\\ &= 4000 + \frac{600000 \cdot 90000}{30000}\\ &= 1804000\\ \text{Result Size} &= \frac{T(T)}{V(T, hashtag)} + \frac{T(R)}{V(R, hashtag)}\\ &= \frac{600000}{50000} + \frac{3000000}{30000}\\ &= 12 + 100\\ &= 112\end{aligned}$$

3.c On-the-fly Selection

Since the selection is on-the-fly, there is no extra cost.

$$\begin{aligned}\text{Result Size} &= \frac{T(R)}{V(R, user)}\\ &= \frac{3000000}{1000}\\ &= 3000\end{aligned}$$

3.d Index Scan

Since Retweet has unclustered index on user, the cost of index based selection on equality is

$$\begin{aligned}\text{Cost} &= \frac{T(R)}{V(R, user)} \\ &= \frac{3000000}{1000} \\ &= 3000 \\ \text{Result Size} &= \frac{T(R)}{V(R, user)} \\ &= \frac{3000000}{1000} \\ &= 3000\end{aligned}$$

3.e In-memory Hash Join, Pipelined

Since we are looking for in-memory hash join for $T \bowtie R$, we need to assume that $B(T) \leq M$ holds.

$$\begin{aligned}\text{Cost} &= B(T) + B(R) \\ &= 4000 + 90000 \\ &= 94000 \\ \text{Result Size} &= \frac{T(T)}{V(T, hashtag)} + \frac{T(R)}{V(R, hashtag)} \\ &= \frac{600000}{50000} + \frac{3000000}{30000} \\ &= 12 + 100 \\ &= 112\end{aligned}$$

3.f Total IO Cost

$$\begin{aligned}\text{Cost for Plan 1} &= Cost(a) + Cost(b) + Cost(c) \\ &= 12 + 1804000 + 0 \\ &= 1804012 \\ \text{Cost for Plan 2} &= Cost(d) + Cost(e) + Cost(a) \\ &= 3000 + 94000 + 12 \\ &= 97012\end{aligned}$$

Since the total cost for Plan 2 is way lower than Plan 1, choosing Plan 2 seems the most reasonable one. Applying selects before doing the joins reduce the cost significantly, due to lowering the number of rows to join.

4 Experiments

4.a

Due to the foreign key constraint, I would expect Postgres to apply a hash join.

```
EXPLAIN
SELECT *
FROM business
NATURAL JOIN tip;
```

```
[ozan@moa ~]$ psql -U ceng352
psql (13.2)
Type "help" for help.

ceng352=# EXPLAIN
SELECT *
FROM business
NATURAL JOIN tip;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=7510.16..91094.99 rows=1162119 width=175)
   Hash Cond: ((tip.business_id)::text = (business.business_id)::text)
   ->  Seq Scan on tip  (cost=0.00..35526.19 rows=1162119 width=128)
   ->  Hash  (cost=3620.85..3620.85 rows=160585 width=70)
         ->  Seq Scan on business  (cost=0.00..3620.85 rows=160585 width=70)
(5 rows)
```

Figure 3: Query Execution Plan for Q4.a

4.b

```
[ozan@moa ~]$ psql -U ceng352
psql (13.2)
Type "help" for help.

ceng352=# EXPLAIN ANALYSE
SELECT *
FROM business
NATURAL JOIN tip;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=7510.16..91094.99 rows=1162119 width=175) (actual time=356.625..2187.705 rows=1162119 loops=1)
   Hash Cond: ((tip.business_id)::text = (business.business_id)::text)
   ->  Seq Scan on tip  (cost=0.00..35526.19 rows=1162119 width=128) (actual time=0.986..415.008 rows=1162119 loops=1)
   ->  Hash  (cost=3620.85..3620.85 rows=160585 width=70) (actual time=355.057..355.058 rows=160585 loops=1)
         Buckets: 65536  Batches: 8  Memory Usage: 2549kB
         ->  Seq Scan on business  (cost=0.00..3620.85 rows=160585 width=70) (actual time=0.031..133.833 rows=160585 loops=1)
 Planning Time: 10.049 ms
 Execution Time: 2270.617 ms
(8 rows)

ceng352=#
```

Figure 4: Query Execution Plan and Execution Time for Q4.b before Indexes

After measuring the query execution times, I tried different indexes to determine the best performing one. Initially I was expecting B+ Tree indexes would perform better, and since I was merging tables using their business_id values, if I cluster using them, I would be able to get the best performance. I created indexes with the following queries:

```
CREATE INDEX U_bid ON Business(business_id);
CREATE INDEX U_bid_tip ON Tip(business_id);
CLUSTER U_bid ON Business;
CLUSTER U_bid_tip on Tip;
```

And the query execution time for the query became lower:

```
ceng352=# EXPLAIN ANALYSE
SELECT *
FROM business
NATURAL JOIN tip;
                                                                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=7511.16..91200.99 rows=1162119 width=175) (actual time=94.944..1761.463 rows=1162119 loops=1)
   Hash Cond: ((tip.business_id)::text = (business.business_id)::text)
   ->  Seq Scan on tip  (cost=0.00..35631.19 rows=1162119 width=128) (actual time=0.022..362.912 rows=1162119 loops=1)
   ->  Hash  (cost=3621.85..3621.85 rows=160585 width=70) (actual time=94.691..94.692 rows=160585 loops=1)
         Buckets: 65536  Batches: 8  Memory Usage: 2549kB
         ->  Seq Scan on business  (cost=0.00..3621.85 rows=160585 width=70) (actual time=0.011..33.798 rows=160585 loops=1)
 Planning Time: 10.835 ms
 Execution Time: 1840.302 ms
(8 rows)
```

Figure 5: Query Execution Plan and Execution Time for Q4.b with Clustered Indexes

Since B+ Trees are very efficient when answering point queries, I believe the join benefited from that. Since scan on a clustered index relies on number of blocks, and scan on an unclustered index relies on number of tuples, and since $X \cdot B(R) < X \cdot T(R)$ for this query, I was able to benefit from that.

4.c

Since there are no foreign key constraints in the query below, I would expect a nested loop join.

```sql
EXPLAIN
SELECT *
FROM business b
CROSS JOIN tip t
WHERE b.business_id < t.business_id;
```

```
[ozan@moa ~]$ psql -U ceng352
psql (13.2)
Type "help" for help.

ceng352=# EXPLAIN
SELECT *
FROM business b
CROSS JOIN tip t
WHERE b.business_id < t.business_id;
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.42..1711198358.55 rows=62206293205 width=198)
   ->  Seq Scan on tip t  (cost=0.00..35526.19 rows=1162119 width=128)
   ->  Index Scan using business_pkey on business b  (cost=0.42..937.17 rows=53528 width=70)
         Index Cond: ((business_id)::text < (t.business_id)::text)
 JIT:
   Functions: 5
   Options: Inlining true, Optimization true, Expressions true, Deforming true
(7 rows)

ceng352=#
```

Figure 6: Query Execution Plan for Q4.c

4.d

When I tried to disable Hash Join and Sequential Scan, Postgres was able to find an alternative way of calculating through Merge Join with Index Scan and Sort.

However, when I tried to disable Nested Loop, Postgres still used Nested Loop but with a higher cost. I could not find any reasoning behind that, but I believe since it is the simplest method, disabling it might not be possible for some queries. Also, when I tried disabling sequential scan, Postgres did not disable it either for this query. The reason behind that might be similar to nested loop, since disabling sequential scan would cause having no available method for this query. However, Index Scan was disabled effectively for the Business table, and Postgres used Bitmap Heap Scan and Bitmap Index Scan.

```
[ozan@moa ceng352-wa2]$ psql -U ceng352
psql (13.2)
Type "help" for help.

ceng352=# set enable_seqscan=off;
SET
ceng352=# set enable_hashjoin=off;
SET
ceng352=# EXPLAIN
SELECT *
FROM business
NATURAL JOIN tip;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Merge Join  (cost=10000303646.93..10000335151.65 rows=1162119 width=175)
   Merge Cond: ((tip.business_id)::text = (business.business_id)::text)
   ->  Sort  (cost=10000303645.95..10000306551.25 rows=1162119 width=128)
         Sort Key: tip.business_id
         ->  Seq Scan on tip  (cost=10000000000.00..10000035631.19 rows=1162119 width=128)
   ->  Index Scan using business_pkey on business  (cost=0.42..13673.00 rows=160585 width=70)
 JIT:
   Functions: 9
   Options: Inlining true, Optimization true, Expressions true, Deforming true
(9 rows)

ceng352=#
```

Figure 7: Query Execution Plan for Q4.a if Hash Join and Sequential Scan are disabled

```
[ozan@moa ceng352-wa2]$ psql -U ceng352
psql (13.2)
Type "help" for help.

ceng352=# set enable_seqscan=off;
SET
ceng352=# set enable_nestloop=off;
SET
ceng352=# set enable_indexscan=off;
SET
ceng352=# EXPLAIN
SELECT *
FROM business b
CROSS JOIN tip t
WHERE b.business_id < t.business_id;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Nested Loop  (cost=20000000415.26..21886904048.59 rows=62206293205 width=198)
   ->  Seq Scan on tip t  (cost=10000000000.00..10000035631.19 rows=1162119 width=128)
   ->  Bitmap Heap Scan on business b  (cost=415.26..1088.36 rows=53528 width=70)
         Recheck Cond: ((business_id)::text < (t.business_id)::text)
         ->  Bitmap Index Scan on business_pkey  (cost=0.00..401.88 rows=53528 width=0)
               Index Cond: ((business_id)::text < (t.business_id)::text)
 JIT:
   Functions: 5
   Options: Inlining true, Optimization true, Expressions true, Deforming true
(9 rows)

ceng352=#
```

Figure 8: Query Execution Plan for Q4.c if Nested Loop, Sequential Search and Index Scan are disabled

4.e

By creating a clustered index on Business.state, we can improve the query execution time, since the query calculates the average stars of each state, and having states in clusters might benefit the query execution time. We can create the index with the following query:

```
1 CREATE INDEX i_state ON Business(state);
2 CLUSTER i_state ON Business;
```

4.f

When we introduce new indexes to the database, if there are other queries that use ranges to generate results for the indexed field, we might actually worsen the query execution time. When I checked my queries from Project 1, I did not see any query with a range or "LIKE" statement for the field, state. Therefore I belive it might not affect it badly. On the other hand, there are some queries that check if the state is 'TX', but they should not be affected by that badly. In fact, it might improve the query execution time depending on the amount of data.