

CS 301 - Introduction to Database Systems
COURSE PROJECT PHASE B



Aarushi - 2019CSB1059
Urvee Gupta - 2019CSB1128

Under the guidance of: Dr. Viswanath Gunturi
Academic year 2021-22

Part A: Experimenting with Query Selectivity

SUBPART i and ii

Q1: SELECT name FROM movie WHERE imdb score < 2;

QUERY PLAN	
1	Bitmap Heap Scan on movie (cost=4638.64..16066.54 rows=247512 width=11) (actual time=71.149..247.788 rows=249847 loops=1)
2	Recheck Cond: (imdb_score < '2'::double precision)
3	Heap Blocks: exact=8334
4	-> Bitmap Index Scan on movie_imdb_score_idx (cost=0.00..4576.77 rows=247512 width=0) (actual time=68.970..68.971 rows=249847 loops=1)
5	Index Cond: (imdb_score < '2'::double precision)
6	Planning Time: 8.852 ms
7	Execution Time: 262.325 ms

Explanation -

Query selectivity = $249847/1000000 \sim 0.2498$ **(24.98%)**

Approximately 25% of all entries are returned in the relation. The number of entries are large enough to not implement a simple index scan but not too large to implement an entire sequence scan on the table. Hence, the query optimizer implements bitmap heap scan on the movie table. Bitmap index scan filters out the tuples which do not satisfy the required condition using imdb_score as the index. Bitmap heap scan then scans through the potential matching blocks.

Q2: SELECT name FROM movie WHERE imdb score between 1.5 and 4.5;

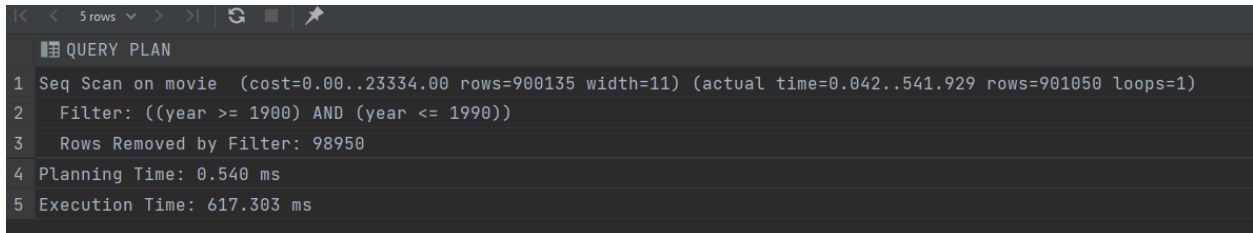
QUERY PLAN	
1	Seq Scan on movie (cost=0.00..23334.00 rows=750664 width=11) (actual time=0.026..542.180 rows=750184 loops=1)
2	Filter: ((imdb_score >= '1.5'::double precision) AND (imdb_score <= '4.5'::double precision))
3	Rows Removed by Filter: 249816
4	Planning Time: 0.193 ms
5	Execution Time: 594.993 ms

Explanation -

Query selectivity = $750184/1000000 \sim 0.75$ **(75%)**

Approximately 75% of all the entries are returned in the relation. The number of entries here is too large to implement an index scan as there would be large block access overhead. Similarly, the number of entries is too large to implement a bitmap index scan since most of the entries would be mapped anyway making it inefficient and redundant. Since 75% of total is a large number of entries (almost covers the entire original table), the query optimizer implements a seq scan as that is the best option. Also, the query optimizer doesn't use imdb_score as the index as it is not beneficial in this case.

Q3: SELECT name FROM movie WHERE year between 1900 and 1990;



The screenshot shows a query plan for the query 'SELECT name FROM movie WHERE year between 1900 and 1990;'. The plan consists of five steps: 1. Seq Scan on movie (cost=0.00..23334.00 rows=900135 width=11) (actual time=0.042..541.929 rows=901050 loops=1); 2. Filter: ((year >= 1900) AND (year <= 1990)); 3. Rows Removed by Filter: 98950; 4. Planning Time: 0.540 ms; 5. Execution Time: 617.303 ms.

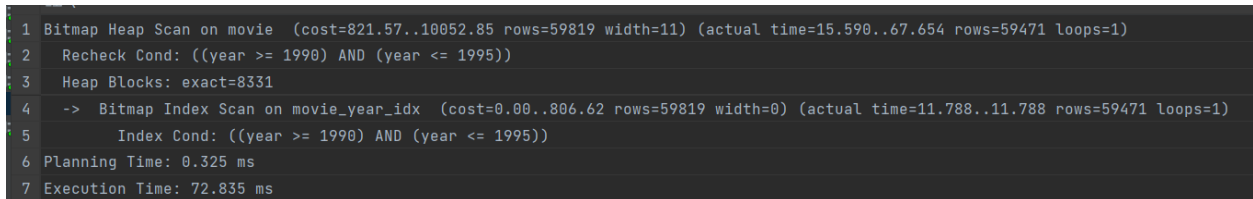
Step	Operation	Cost	Rows	Width	Actual Time	Actual Rows	Loops
1	Seq Scan on movie	0.00..23334.00	900135	11	0.042..541.929	901050	1
2	Filter: ((year >= 1900) AND (year <= 1990))						
3	Rows Removed by Filter		98950				
4	Planning Time				0.540 ms		
5	Execution Time				617.303 ms		

Explanation -

Query selectivity = $901050/1000000 \sim 0.901050$ **(90.1%)**

Approximately 90.1% of all the entries are returned in the relation. The number of entries here is again too large to implement an index scan as there would be large block access overhead. Similarly, the number of entries is too large to implement a bitmap index scan since most of the entries would be mapped anyway making it inefficient and redundant. Since 90.1% of total is a large number of entries (almost covers the entire original table), the query optimizer implements a seq scan as that is the best option.

Q4: SELECT name FROM movie WHERE year between 1990 and 1995;



The screenshot shows a query plan for the query 'SELECT name FROM movie WHERE year between 1990 and 1995;'. The plan consists of seven steps: 1. Bitmap Heap Scan on movie (cost=821.57..10052.85 rows=59819 width=11) (actual time=15.590..67.654 rows=59471 loops=1); 2. Recheck Cond: ((year >= 1990) AND (year <= 1995)); 3. Heap Blocks: exact=8331; 4. -> Bitmap Index Scan on movie_year_idx (cost=0.00..806.62 rows=59819 width=0) (actual time=11.788..11.788 rows=59471 loops=1); 5. Index Cond: ((year >= 1990) AND (year <= 1995)); 6. Planning Time: 0.325 ms; 7. Execution Time: 72.835 ms.

Step	Operation	Cost	Rows	Width	Actual Time	Actual Rows	Loops
1	Bitmap Heap Scan on movie	821.57..10052.85	59819	11	15.590..67.654	59471	1
2	Recheck Cond: ((year >= 1990) AND (year <= 1995))						
3	Heap Blocks: exact=8331						
4	-> Bitmap Index Scan on movie_year_idx	0.00..806.62	59819	0	11.788..11.788	59471	1
5	Index Cond: ((year >= 1990) AND (year <= 1995))						
6	Planning Time				0.325 ms		
7	Execution Time				72.835 ms		

Explanation -

Query selectivity = $59471/1000000 \sim 0.059471$ **(5.947%)**

Approximately 5.947% of all entries are returned in the relation. The number of entries are large enough to not implement a simple index scan but not too large to implement an entire sequence scan on the table. Hence, the query optimizer implements bitmap heap scan on the movie table. Bitmap index scan filters out the tuples which do not satisfy the required condition using year as the index. Bitmap heap scan then scans through the potential matching blocks.

Q5: SELECT * FROM movie WHERE pc_id < 50;

```
1  Bitmap Heap Scan on movie  (cost=1038.39..10465.87 rows=87479 width=31) (actual time=17.227..71.538 rows=88085 loops=1)
2    Recheck Cond: (pc_id < 50)
3    Heap Blocks: exact=8334
4    -> Bitmap Index Scan on movie_pc_id_idx  (cost=0.00..1016.52 rows=87479 width=0) (actual time=14.016..14.017 rows=88085 loops=1)
5        Index Cond: (pc_id < 50)
6  Planning Time: 0.279 ms
7  Execution Time: 78.510 ms
```

Explanation -

Query selectivity = $88085/1000000 \sim 0.088$ **(8.8%)**

Approximately 8.8% of all entries are returned in the relation. The number of entries are large enough to not implement a simple index scan but not too large to implement an entire sequence scan on the table. Hence, the query optimizer implements bitmap heap scan on the movie table. Bitmap index scan filters out the tuples which do not satisfy the required condition using pc_id as the index. Bitmap heap scan then scans through the potential matching blocks.

Q6: SELECT * FROM movie WHERE pc_id > 20000;

```
1  Bitmap Heap Scan on movie  (cost=910.89..10203.70 rows=76705 width=31) (actual time=20.607..61.444 rows=75337 loops=1)
2    Recheck Cond: (pc_id > 20000)
3    Heap Blocks: exact=8334
4    -> Bitmap Index Scan on movie_pc_id_idx  (cost=0.00..891.71 rows=76705 width=0) (actual time=18.275..18.275 rows=75337 loops=1)
5        Index Cond: (pc_id > 20000)
6  Planning Time: 0.779 ms
7  Execution Time: 66.557 ms
```

Explanation -

Query selectivity = $75337/1000000 \sim 0.075$ **(7.5%)**

Approximately 8.8% of all entries are returned in the relation. The number of entries are large enough to not implement a simple index scan but not too large to implement an entire sequence scan on the table. Hence, the query optimizer implements bitmap heap scan on the movie table. Bitmap index scan filters out the tuples which do not satisfy the required condition using pc_id as the index. Bitmap heap scan then scans through the potential matching blocks.

SUBPART iii

Query 1 uses imdb_score as an index but query 2 doesn't.

In query 1, Bitmap index scan is used by the optimizer because 25% (query selectivity is approximately 25%) of total entries are returned by the relation. As the number of entries to be processed is reasonably low, it's a more selective query. The bitmap index scan creates a bitmap of the expected row locations, and the bitmap heap scan accesses those locations to find the required rows hence making the overall process efficient. Since the B+ tree was not built on the index name, using index scan is not the best choice here as it would have to read separate blocks for imdb scores to access the name, while the bitmap index scan does not require the same.

In query 2, Sequence scan is used by the optimizer since 75% (query selectivity is approximately 75%) of total entries are returned by the relation. Sequential scan is the best option here as index scan would require a lot more I/O operations because of the extra number of blocks it would have to read to reach the leaf nodes of the B+ trees. The overhead of choosing which blocks to read in index scan would weigh out the minimal optimization achieved by using it. Sequential scan reads blocks in a sequential manner hence, avoiding multiple block reads of the same block. As almost the entire table is returned by the relation, sequential scanning would be most efficient.

SUBPART iv

Query 3 does not use year as index but query 4 does.

In query 3, the query selectivity is approximately 90%. Since almost the entire table is returned by the relation, sequential scan is preferred over other methods since it yields less overhead and minimum multiple block reads.

In query 4, the query selectivity is approximately 6% which is significantly less. The optimizer uses bitmap index scan followed by bitmap heap scan. Bitmap index scan is preferred here over index scan to avoid reading blocks for each year to find the names. Index only scan would probably be the optimizer's first choice if the query output just required the year.

SUBPART v

Both query 5 and query 6 use pc_id as the index.

In query 5 and query 6, the query selectivity is 8.8% and 7.5% respectively. As the number of entries are comparable, both queries use the same optimization methods. The number of entries are not large enough to implement a sequence scan and not too low to implement index scan either. Hence, the optimizer uses bitmap index scan followed by bitmap heap scan. [In our case 90% of the pc_ids in the table are a random integer between 1 and 500 and others (remaining 10%) are distributed over 501 and 80000.]

Part B: Join Strategies

SUBPART i and ii

Q1: Join Actor, Movie and Casting; Where a_id < 50; finally, the query outputs actor name and movie name

```
Sql query - EXPLAIN ANALYZE SELECT actor.name, movie.name
          FROM actor, movie, casting
          Where actor.a_id < 50 and
                casting.a_id = actor.a_id and
                casting.m_id = movie.m_id;
```

QUERY PLAN	
1	Gather (cost=1178.91..40108.87 rows=653 width=27) (actual time=6.112..847.424 rows=18701 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Nested Loop (cost=178.91..39043.57 rows=272 width=27) (actual time=2.037..651.406 rows=6234 loops=3)
5	-> Hash Join (cost=178.49..38920.20 rows=272 width=20) (actual time=1.935..542.701 rows=6234 loops=3)
6	Hash Cond: (casting.a_id = actor.a_id)
7	-> Parallel Seq Scan on casting (cost=0.00..34366.65 rows=1666665 width=8) (actual time=0.712..296.226 rows=1333332 loops=3)
8	-> Hash (cost=177.88..177.88 rows=49 width=20) (actual time=0.737..0.738 rows=49 loops=3)
9	Buckets: 1024 Batches: 1 Memory Usage: 11kB
10	-> Bitmap Heap Scan on actor (cost=4.80..177.88 rows=49 width=20) (actual time=0.104..0.695 rows=49 loops=3)
11	Recheck Cond: (a_id < 50)
12	Heap Blocks: exact=49
13	-> Bitmap Index Scan on actor_a_id_idx (cost=0.00..4.79 rows=49 width=0) (actual time=0.069..0.069 rows=49 loops=3)
14	Index Cond: (a_id < 50)
15	-> Index Scan using movie_m_id_idx on movie (cost=0.42..0.45 rows=1 width=15) (actual time=0.016..0.016 rows=1 loops=18701)
16	Index Cond: (m_id = casting.m_id)
17	Planning Time: 25.184 ms
18	Execution Time: 850.256 ms

Explanation -

In this query the optimiser decides to involve 2 workers which work parallelly to complete the task. The query also uses a nested loop.

The 'Gather' node is included in the query plan which has the responsibility to work like other workers and compile the work.

In the outer loop, bitmap index scan is performed on the actor table as the number of rows are very low. A bitmap index scan, instead of fetching the actual rows, constructs a bitmap of potential row locations. This bitmap is then used by bitmap heap scan(line 10) to look up the relevant data. A hash table is then constructed on the actors table(line 8). Parallelly, for the casting table, a sequential scan is performed. The outputs are then hash joined with a hash condition: casting.a_id = actor.a_id.

For the inner loop(line 15), index scan is performed on the movie table with a condition m_id = casting.m_id and the output is returned.

Q2: Join Actor and Casting; Where m_id < 50; finally, the query outputs actor name and movie name

```
Sql query - EXPLAIN ANALYZE SELECT a.name
                        FROM actor a, casting c
                        WHERE a.a_id = c.a_id AND c.m_id < 50;
```

```
1 Nested Loop (cost=0.85..1507.81 rows=191 width=16) (actual time=0.013..0.449 rows=196 loops=1)
2   -> Index Only Scan using casting_pk on casting c (cost=0.43..7.77 rows=191 width=4) (actual time=0.005..0.027 rows=196 loops=1)
3       Index Cond: (m_id < 50)
4       Heap Fetches: 0
5   -> Index Scan using actor_a_id_idx on actor a (cost=0.42..7.85 rows=1 width=20) (actual time=0.002..0.002 rows=1 loops=196)
6       Index Cond: (a_id = c.a_id)
7 Planning Time: 0.248 ms
8 Execution Time: 0.473 ms
```

Explanation -

In this query, first an index only scan is performed on the casting table with the condition m_id<50. Index scan was chosen because of the selective nature of the query. Index only scan is chosen because it performs the minimum number of disk block reads. Next, for every tuple returned in the above step, in which an index scan was run on the actor table with condition actor.a_id equals the a_id as returned by the concerned casting tuple. This was done as the query is expected to return only one tuple.

Q3: Join Movie and Production Company; where imdb score is less than 1.5. Finally, the query outputs the movie name and production company.

```
Sql query - EXPLAIN ANALYZE SELECT movie.name,
                        production_company.name
FROM movie, production_company
WHERE imdb_score < 1.5 AND
      movie.pc_id =
      production_company.pc_id;
```

```
QUERY PLAN
1 Hash Join (cost=5231.07..17001.53 rows=122277 width=22) (actual time=230.397..556.596 rows=124896 loops=1)
2   Hash Cond: (movie.pc_id = production_company.pc_id)
3   -> Bitmap Heap Scan on movie (cost=2292.07..12154.53 rows=122277 width=15) (actual time=28.239..135.536 rows=124896 loops=1)
4       Recheck Cond: (imdb_score < '1.5'::double precision)
5       Heap Blocks: exact=8334
6       -> Bitmap Index Scan on movie_imdb_score_idx (cost=0.00..2261.50 rows=122277 width=0) (actual time=25.907..25.908 rows=124896 loops=1)
9           Index Cond: (imdb_score < '1.5'::double precision)
8   -> Hash (cost=1548.00..1548.00 rows=80000 width=15) (actual time=200.522..200.525 rows=80000 loops=1)
9       Buckets: 131072 Batches: 2 Memory Usage: 2908kB
10      -> Seq Scan on production_company (cost=0.00..1548.00 rows=80000 width=15) (actual time=0.635..107.523 rows=80000 loops=1)
11 Planning Time: 13.258 ms
12 Execution Time: 571.047 ms
```

Explanation -

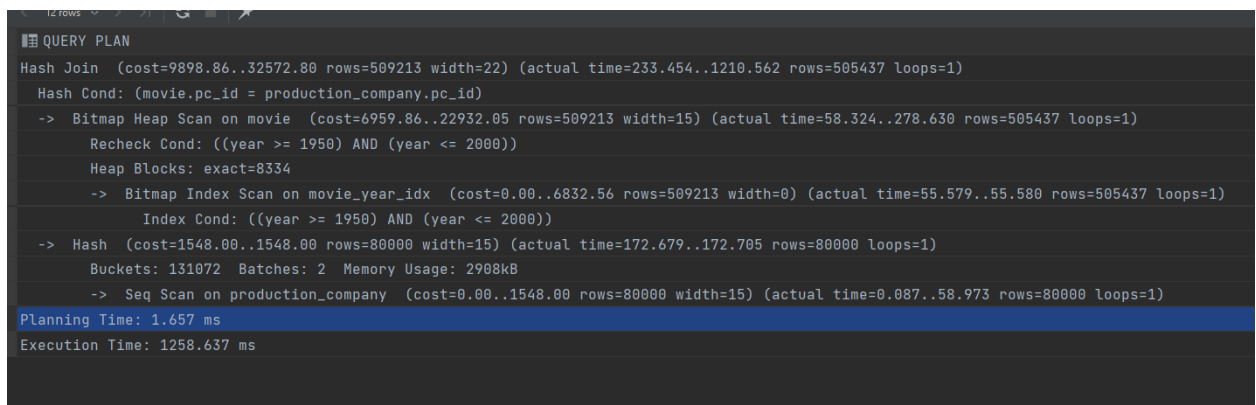
In this query, the movie table is scanned using bitmap index scan. This basically traverses through the entire B+ tree and marks the blocks as needed or not needed. Bitmap heap scan is then performed on the blocks marked as needed.

For the production company table, sequential scan is chosen by the query optimiser. This scan is performed on the production company table with an objective of making hash buckets out of it so that they can be transferred to the main memory and the execution time is minimized.

After this, 'Hash' is used to join the tuples retrieved from both scans based on the condition that `movies.pc_id = pc.pc_id`.

Q4: Join Movie and Production Company; where year is between 1950 and 2000. Finally, the query outputs the movie name and production company.

```
Sql query - EXPLAIN ANALYZE SELECT movie.name,  
                                     production_company.name  
FROM   movie, production_company  
WHERE  movie.year between 1950 and 2000  
       AND movie.pc_id =  
       Production_company.pc_id;
```



QUERY PLAN
Hash Join (cost=9898.86..32572.80 rows=509213 width=22) (actual time=233.454..1210.562 rows=505437 loops=1)
Hash Cond: (movie.pc_id = production_company.pc_id)
-> Bitmap Heap Scan on movie (cost=6959.86..22932.05 rows=509213 width=15) (actual time=58.324..278.630 rows=505437 loops=1)
Recheck Cond: ((year >= 1950) AND (year <= 2000))
Heap Blocks: exact=8334
-> Bitmap Index Scan on movie_year_idx (cost=0.00..6832.56 rows=509213 width=0) (actual time=55.579..55.580 rows=505437 loops=1)
Index Cond: ((year >= 1950) AND (year <= 2000))
-> Hash (cost=1548.00..1548.00 rows=80000 width=15) (actual time=172.679..172.705 rows=80000 loops=1)
Buckets: 131072 Batches: 2 Memory Usage: 2908kB
-> Seq Scan on production_company (cost=0.00..1548.00 rows=80000 width=15) (actual time=0.087..58.973 rows=80000 loops=1)
Planning Time: 1.657 ms
Execution Time: 1258.637 ms

Explanation -

In this query, the movie table is scanned using bitmap index scan. This basically traverses through the entire B+ tree and marks the blocks as needed or not needed. Bitmap heap scan is then performed on the blocks marked as needed.

For the production company table, sequential scan is chosen by the query optimiser. This scan is performed on the production company table with an objective of making hash buckets out of it so that they can be transferred to the main memory and the execution time is minimized.

After this, 'Hash' is used to join the tuples retrieved from both scans based on the condition that `movies.pc_id = pc.pc_id`.

SUBPART iii

Query 1 -

- Index scan - In the inner loop, the optimizer just needs to find one particular entry for each cycle, hence the best choice is to use bitmap index scan followed by bitmap heap scan to retrieve the actual entry. Another time index scan is used is on the actor table, using a_id as the index as the number of scans (rows) are less (about 50).
- Hash - hash operation is performed on the entries retrieved from the actor table because it is smaller in size (as compared to casting table) and hence occupies less space in the main memory.
- Sequence scan - a parallel sequence scan is performed on the casting table taking into consideration its large size. Hash join operation is then performed with the hash table formed on the actor table.
- Nested loops - using nested loops is a better choice here compared to hash join since the large size of the table would lead to slower processing. We can find m_id in the movie table with the available index.

Query 2-

- Nested loop - A nested loop is used by the query optimiser to perform the required operation.
- Index scan - It uses index only scan on the casting table to filter rows based on the condition m_id.50. Index scan is preferred in this case because the expected number of rows to be retrieved are $\leq 0.005\%$ which is very less.
For each row selected in this query, an index scan is performed on the actor table with the condition actor.a_id = casting.a_id. As the number of tuples for each m_id are also low here, traversing a B+ tree, i.e. using an index scan is much preferable.

Query 3-

- Bitmap index scan - This is the case of a relatively larger query. Hence, bitmap index scan is performed on the movie table to reduce the disk block reads. Because of the adjacency in extracted tuples, disk blocks would be overlapping. Bitmap index scan reduces disk block reads. It is also suitable for this case as it only retrieves the location of the required row as required.
- Bitmap heap scan - After the last operation, the bitmap heap scan is performed on the row extracted and the rows satisfying the relation condition are stored in the answer.
- Hash - hash join is then performed taking production_company and movie as the bound relation and probe relation respectively. The tuples from the movie table are probed one by one to compare with the pc_id.

Query 4-

- Bitmap index scan - considering bulk query, bitmap index scan is performed instead of normal index scan. This is done by the optimizer to reduce disk block reads since disk blocks will overlap in extracting adjacent tuples. It is also suitable for this case as it only retrieves the location of the required row as required.
- Bitmap heap scan - After the last operation, the bitmap heap scan is performed on the row extracted and the rows satisfying the relation condition are stored in the answer.
- Hash - hash join is then performed taking production_company and movie as the bound relation and probe relation respectively. The tuples from the movie table are probed one by one to compare with the pc_id.