

CS301, DBMS

Database Management and Systems



Indian Institute of Technology, Ropar

Topic: Analysis of Query Planning

Instructor: Dr. Vishwanath Gunturi

Date of Submission: 17-11-2021

Team Members

Vasu Bansal (2019csb1130@iitrpr.ac.in)

Pradeep Kumar (2019csb1107@iitrpr.ac.in)

Anant Dhakad (2019csb1070@iitrpr.ac.in)

Table of Contents

- 1) [Distribution of Data & Assumptions:](#)
- 2) [Indexes](#)
- 3) [Part A: Experimenting with Query Selectivity](#)
 - a) [Queries](#)
 - b) [Running the EXPLAIN ANALYZE command \(Briefly explain query strategy\)](#)
 - i) [Query1](#)
 - ii) [Query2](#)
 - iii) [Query3](#)
 - iv) [Query4](#)
 - v) [Query5](#)
 - vi) [Query6](#)
 - c) [Is the index on imdbscore used for Query1 and Query2?](#)
 - i) [Query1](#)
 - ii) [Query2](#)
 - d) [Is the index on the year used for both Query3 and Query4?](#)
 - i) [Query3](#)
 - ii) [Query4](#)
 - e) [Is the index on pc id used for both Query5 and Query6?](#)
 - i) [Query5](#)
 - ii) [Query6](#)
- 4) [Part B: Join Strategies](#)
 - a) [Queries](#)
 - b) [Running the EXPLAIN ANALYZE command \(Briefly explain query strategy\)](#)
 - i) [Query1](#)
 - ii) [Query2](#)
 - iii) [Query3](#)
 - iv) [Query4](#)
 - c) [Briefly justify the query optimiser choices \(choice of query processing algorithms\)](#)
 - i) [Query1](#)
 - ii) [Query2](#)
 - iii) [Query3](#)
 - iv) [Query4](#)

Distribution of Data & Assumptions:

actor: Uniform Random.

productioncompany: Uniform Random.

movie: imdb_score for 95% tuples between 1-2, remaining uniform random (assuming that imdbscore is float value rounded up to one decimal point).

casting: 95% of actor ids within 1-1000.

Indexes

"All the Indices Present in the Database"		
tablename	indexname	indexdef
actor	actor_pkey	CREATE UNIQUE INDEX actor_pkey ON public.actor USING btree (a_id)
casting	casting_m_id	CREATE INDEX casting_m_id ON public.casting USING btree (m_id)
casting	casting_pkey	CREATE UNIQUE INDEX casting_pkey ON public.casting USING btree (m_id, a_id)
movie	movie_fk_pc_id	CREATE INDEX movie_fk_pc_id ON public.movie USING btree (pc_id)
movie	movie_imdbscore	CREATE INDEX movie_imdbscore ON public.movie USING btree (imdbscore)
movie	movie_pkey	CREATE UNIQUE INDEX movie_pkey ON public.movie USING btree (m_id)
movie	movie_year	CREATE INDEX movie_year ON public.movie USING btree (year)
productioncompany	productioncompany_pkey	CREATE UNIQUE INDEX productioncompany_pkey ON public.productioncompany USING btree (pc_id)
(8 rows)		

Part A: Experimenting with Query Selectivity

Queries

```

Query1: SELECT name FROM movie WHERE imdbscore < 2;
Query2: SELECT name FROM movie WHERE imdbscore between 1.5 and 4.5;
Query3: SELECT name FROM movie WHERE year between 1900 and 1990;
Query4: SELECT name FROM movie WHERE year between 1990 and 1995;
Query5: SELECT * FROM movie WHERE pc_id < 50;
Query6: SELECT * FROM movie WHERE pc_id > 20000;

```

Running the *EXPLAIN ANALYZE* command (Briefly explain query strategy)

Query1

```
bollywood=# EXPLAIN ANALYZE SELECT name FROM movie WHERE imdbscore < 2;
               QUERY PLAN
-----
Seq Scan on movie  (cost=0.00..19853.00 rows=901867 width=11) (actual time=0.065..396.149 rows=902704 loops=1)
  Filter: (imdbscore < '2'::numeric)
  Rows Removed by Filter: 97296
Planning Time: 0.280 ms
Execution Time: 440.639 ms
(5 rows)
```

Sequential Scan is performed on the movie table. For each row from the movie table, the condition (*imdbscore* < 2) is checked and the row satisfying the condition will be returned.

The *Sequential Scan* will work better than *Index Scan* or *Bitmap Scan* because the number of expected rows is considerable (about 90%).

Query2

```
bollywood=# EXPLAIN ANALYZE SELECT name FROM movie WHERE imdbscore between 1.5 and 4.5;
               QUERY PLAN
-----
Seq Scan on movie  (cost=0.00..22353.00 rows=563000 width=11) (actual time=0.041..435.046 rows=564389 loops=1)
  Filter: ((imdbscore >= 1.5) AND (imdbscore <= 4.5))
  Rows Removed by Filter: 435611
Planning Time: 0.256 ms
Execution Time: 463.848 ms
(5 rows)
```

Sequential Scan is performed on the movie table. For each row from the movie table, the condition ((*imdbscore* >= 1.5) and (*imdbscore* <= 4.5)) is checked and the row satisfying the condition will be returned.

The *Sequential Scan* will work better than *Index Scan* or *Bitmap Scan* because the number of expected rows is considerable (about 56%).

Query3

```
bollywood=# EXPLAIN ANALYZE SELECT name FROM movie WHERE year between 1900 and 1990;
               QUERY PLAN
-----
Seq Scan on movie (cost=0.00..22353.00 rows=890269 width=11) (actual time=0.026..266.028 rows=900367 loops=1)
  Filter: ((year >= 1900) AND (year <= 1990))
  Rows Removed by Filter: 99633
Planning Time: 2.011 ms
Execution Time: 311.812 ms
(5 rows)
```

Sequential Scan is performed on the movie table. For each row from the movie table, the condition $((year \geq 1900) \text{ AND } (year \leq 1990))$ is checked and the row satisfying the condition will be returned.

The *Sequential Scan* will work better than *Index Scan* or *Bitmap Scan* because the number of expected rows is considerable (about 89%).

Query4

```
bollywood=# EXPLAIN ANALYZE SELECT name FROM movie WHERE year between 1990 and 1995;
               QUERY PLAN
-----
Bitmap Heap Scan on movie (cost=955.14..9354.06 rows=69728 width=11) (actual time=20.138..49.103 rows=59699 loops=1)
  Recheck Cond: ((year >= 1990) AND (year <= 1995))
  Heap Blocks: exact=7352
  -> Bitmap Index Scan on movie_year (cost=0.00..937.70 rows=69728 width=0) (actual time=18.260..18.261 rows=59699 loops=1)
       Index Cond: ((year >= 1990) AND (year <= 1995))
Planning Time: 0.264 ms
Execution Time: 52.239 ms
(7 rows)
```

Bitmap Heap Scan is performed on the movie table. Because the number of expected rows (6.9%) isn't particularly huge or low, a *Bitmap Heap Scan* is preferable.

Bitmap Index Scan fetches all the tuple-pointers from the *movie_year* index with index condition $((year \geq 1990) \text{ AND } (year \leq 1995))$ in one go, sorts them using an in-memory "bitmap" data structure, and then visits the table rows in physical row-location order and returns them.

Query5

```
bollywood=# EXPLAIN ANALYZE SELECT * FROM movie WHERE pc_id < 50;
              QUERY PLAN
-----
Bitmap Heap Scan on movie  (cost=9.35..1927.49 rows=635 width=29) (actual time=0.249..1.043 rows=589 loops=1)
  Recheck Cond: (pc_id < 50)
  Heap Blocks: exact=566
  -> Bitmap Index Scan on movie_fk_pc_id  (cost=0.00..9.19 rows=635 width=0) (actual time=0.142..0.142 rows=589 loops=1)
       Index Cond: (pc_id < 50)
Planning Time: 9.183 ms
Execution Time: 1.125 ms
(7 rows)
```

Bitmap Heap Scan is performed on the movie table. *Bitmap Index Scan* fetches all the tuple-pointers from the *movie_fk_pc_id* index with index condition(*pc_id*<50) in one go, sorts them using an in-memory “bitmap” data structure, and then visits the table rows in physical row-location order and returns them.

Query6

```
bollywood=# EXPLAIN ANALYZE SELECT * FROM movie WHERE pc_id > 20000;
              QUERY PLAN
-----
Seq Scan on movie  (cost=0.00..19853.00 rows=750168 width=29) (actual time=0.022..231.618 rows=750429 loops=1)
  Filter: (pc_id > 20000)
  Rows Removed by Filter: 249571
Planning Time: 0.164 ms
Execution Time: 270.222 ms
(5 rows)
```

Sequential Scan is performed on the movie table. For each row from the movie table, the condition (*pc_id* > 20000) is checked and the row satisfying the condition will be returned.

The *Sequential Scan* will work better than *Index Scan* or *Bitmap Scan* because the number of expected rows is considerable (about 75%).

Is the *index on imdbscore* used for Query1 and Query2?

No, the index on *imdbscore* is not used for Query1 and Query2.

The total number of rows in the movie table is 10,00,000.

In our scenario, the *imdbscore* for 95% of the rows should be in the range [1,2], and the *imdbscore* for the remaining records should be distributed evenly.

Query1

The expected number of rows is around 90% (901867 to be precise) of the total rows.

Query2

The number of expected rows is around 56.3% (56300 to be precise) of total rows.

The query optimizer selects to run a sequential scan over the records because this proportion is too high for bitmap index scan and index-only scan. Therefore the query planner chose to sequentially scan all tuples instead of going for an index scan. This is due to the fact that an index scan necessitates multiple IO operations for each row in order to search for the row in the index and then retrieve the row from the heap. A sequential scan, on the other hand, takes one or fewer IOs for each row because a single block on the disc can contain several rows, allowing all of the rows to be fetched in a single IO operation.

Besides *movie_imdbscore* index only contains 41 leaf nodes (since there are 41 unique values of *imdbscore*), and each of its nodes points to pages\blocks containing around 10000 rows. So when fetching these rows from the table, the cost of random I/O access would have proved more expensive than a simple sequential scan.

Is the *index* on the *year* used for both Query3 and Query4?

Index is used in Query4 but not in Query3. (Here “index” does not imply the *INDEX ONLY SCAN*, but rather it implies the data structure index that is being used by *Bitmap Index Scan*).

Query3

The expected number of rows is around 89% (890269 to be precise), which is too large for using *Bitmap Heap scan* or *Index Only Scan*. Because fetching rows from resulting leaf nodes in *Index Only Scan* (or fetching blocks from row location bitmap returned by *Bitmap Index Scan* and rechecking over the block to fetch rows in case of *Bitmap Heap Scan*) will be more expensive than simple scanning over all actual rows and checking the *Where clause* condition.

So the query planner decided to go with *Sequential Scan* on the table.

Query4

The expected number of rows is around 6.97% (69728 to be precise), which is too much for *Index Only Scan* and too little for a *Sequential Scan*. As the expected rows are ~7%, there will be too much random I/O access while fetching actual rows which makes *Index Only Scan* expensive. And since expected rows are only ~7% of total rows it is not necessary to scan through all the rows as that would increase the cost. Therefore *Bitmap Heap Scan* is used by the optimizer.

Is the *index* on *pc_id* used for both Query5 and Query6?

Index is used in Query5 but not in Query6. (Here “index” does not imply the *INDEX ONLY SCAN*, but rather it implies the data structure index that is being used by *Bitmap Index Scan*).

Query5

The expected number of rows is 635, which is too large using *Index Only Scan* and too less for using *Sequential Scan*. Therefore the optimizer chooses *Bitmap Heap Scan*.

Query6

The expected number of rows is around 75% (750168 to be precise), which is too much for *Index Only Scan* or *Bitmap Heap Scan*. Therefore the optimizer chooses *Sequential Scan*.

Part B: Join Strategies

Queries

```
-- Query1
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;

-- Query2
select actor.name
from actor, casting
where casting.m_id < 50 and
      casting.a_id = actor.a_id;

-- Query3
select M.name, PC.name
from Movie as M, ProductionCompany as PC
where imdbscore < 1.5 and
      M.pc_id = PC.pc_id;

-- Query4
select M.name, PC.name
from Movie as M, ProductionCompany as PC
where year between 1950 and 2000 and
      M.pc_id = PC.pc_id;
```

Running the *EXPLAIN ANALYZE* command (Briefly explain query strategy)

Query1

```

                                QUERY PLAN
-----
Gather  (cost=1010.26..39932.39 rows=627 width=27) (actual time=1.823..1707.660 rows=186158 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  -> Nested Loop  (cost=10.26..38869.69 rows=261 width=27) (actual time=1.479..1594.440 rows=62053 loops=3)
    -> Hash Join  (cost=9.83..38751.57 rows=261 width=20) (actual time=1.391..1276.574 rows=62053 loops=3)
      Hash Cond: (casting.a_id = actor.a_id)
      -> Parallel Seq Scan on casting  (cost=0.00..34366.67 rows=1666667 width=8) (actual time=0.826..1056.623 rows=1333333 loops=3)
      -> Hash  (cost=9.24..9.24 rows=47 width=20) (actual time=0.126..0.126 rows=49 loops=3)
        Buckets: 1024  Batches: 1  Memory Usage: 11kB
        -> Index Scan using actor_pkey on actor  (cost=0.42..9.24 rows=47 width=20) (actual time=0.070..0.089 rows=49 loops=3)
          Index Cond: (a_id < 50)
    -> Index Scan using movie_pkey on movie  (cost=0.42..0.45 rows=1 width=15) (actual time=0.004..0.004 rows=1 loops=186158)
      Index Cond: (m_id = casting.m_id)
Planning Time: 1.000 ms
Execution Time: 1718.763 ms
(15 rows)

```

With the criterion `actor.a_id < 50`, we need to combine three tables in this query: actor, movie, and casting.

To combine the actor and casting tables, the query optimizer used Hash Join. The required actor rows are first identified using *Index Scan* on the condition that `a_id < 50`. This table is then utilised as a hash table's construct relation.

On the casting table, the *parallel sequential scan* is employed, and this relation serves as a probe for the hash join.

The generated table is then combined with the movie table via a Nested Loop, with the movie table serving as the inner relation. The *Index scan* selects the corresponding value from the movie table.

Query2

```
bollywood=# EXPLAIN ANALYZE
bollywood=# select actor.name
bollywood=# from actor, casting
bollywood=# where casting.m_id < 50 and
bollywood=#     casting.a_id = actor.a_id;

                                QUERY PLAN
-----
Nested Loop (cost=0.85..1456.61 rows=184 width=16) (actual time=1.567..2.788 rows=196 loops=1)
  -> Index Only Scan using casting_pkey on casting (cost=0.43..7.65 rows=184 width=4) (actual time=0.153..0.192 rows=196 loops=1)
      Index Cond: (m_id < 50)
      Heap Fetches: 0
  -> Index Scan using actor_pkey on actor (cost=0.42..7.87 rows=1 width=20) (actual time=0.009..0.009 rows=1 loops=196)
      Index Cond: (a_id = casting.a_id)
Planning Time: 25.311 ms
Execution Time: 5.781 ms
```

Initially, an *Index Only Scan* is performed on the casting table using *casting_pkey* index with condition *m_id < 50* (reason for performing Index Only Scan and using *casting_pkey* index & not *casting_m_id* is stated in [next section](#)).

Then a nested loop is applied between the intermediate result from the casting table and actor table.

Basically, for every row of intermediate result from the casting table an *Index Scan* is performed on the Actor table with index condition *a_id = casting.a_id* using *actor_pkey* index (reason for why is *Index Scan* is performed on actor table and *Index Only Scan* is performed on casting table is stated in [next section](#)) to find the final query result.

Query3

```
bollywood=# EXPLAIN ANALYZE
bollywood=# select M.name, PC.name
bollywood=# from Movie as M, ProductionCompany as PC
bollywood=# where imdbscore < 1.5 and
bollywood=#     M.pc_id = PC.pc_id;

                                QUERY PLAN
-----
Hash Join (cost=2939.00..28494.22 rows=428633 width=22) (actual time=64.546..893.705 rows=428042 loops=1)
  Hash Cond: (m.pc_id = pc.pc_id)
  -> Seq Scan on movie m (cost=0.00..19853.00 rows=428633 width=15) (actual time=0.405..478.547 rows=428042 loops=1)
      Filter: (imdbscore < 1.5)
      Rows Removed by Filter: 571958
  -> Hash (cost=1548.00..1548.00 rows=80000 width=15) (actual time=62.901..62.902 rows=80000 loops=1)
      Buckets: 131072 Batches: 2 Memory Usage: 2900kB
      -> Seq Scan on productioncompany pc (cost=0.00..1548.00 rows=80000 width=15) (actual time=0.071..30.795 rows=80000 loops=1)
Planning Time: 16.925 ms
Execution Time: 917.479 ms
(10 rows)
```

Initially, a sequential scan is done over the ProductionCompany table, a hash key based on *pc.pc_id* is used (because it is used in the JOIN condition) to insert each

row in a hash table. The data is divided into 131072 buckets naturally as it's the power of 2 just larger than 80,000 and there 80,000 unique pc_id's.

After the hash is built, a sequential scan is performed over the movie table. Here the movie table acts as outer relation and the productioncompany table acts as an inner relation. For each row in the movie table, filter condition (imdbscore < 1.5) is checked and the hash key is calculated based on the movie.pc_id as this is present in the JOIN condition. If the calculated hash key is present in the hash table, the productioncompany row in the hash bucket table and movie row will be returned.

(Reason for why productioncompany table is hashed and not movie table. And why *Sequential Scan* is used on the movie table instead of *Index Scan* or *Bitmap Scan* is given in the [next section](#)).

Query4

```
bollywood=# EXPLAIN ANALYZE
bollywood=# select M.name, PC.name
bollywood=# from Movie as M, ProductionCompany as PC
bollywood=# where M.pc_id = PC.pc_id and
bollywood=#     year between 1950 and 2000;

                                QUERY PLAN
-----
Hash Join (cost=9925.85..31669.46 rows=511066 width=22) (actual time=77.103..598.626 rows=505565 loops=1)
  Hash Cond: (m.pc_id = pc.pc_id)
    -> Bitmap Heap Scan on movie m (cost=6986.85..22005.84 rows=511066 width=15) (actual time=36.283..195.663 rows=505565 loops=1)
      Recheck Cond: ((year >= 1950) AND (year <= 2000))
      Heap Blocks: exact=7353
      -> Bitmap Index Scan on movie_year (cost=0.00..6859.09 rows=511066 width=0) (actual time=34.214..34.215 rows=505565 loops=1)
        Index Cond: ((year >= 1950) AND (year <= 2000))
    -> Hash (cost=1548.00..1548.00 rows=80000 width=15) (actual time=39.775..39.776 rows=80000 loops=1)
      Buckets: 131072 Batches: 2 Memory Usage: 2900kB
      -> Seq Scan on productioncompany pc (cost=0.00..1548.00 rows=80000 width=15) (actual time=0.021..14.591 rows=80000 loops=1)
Planning Time: 0.517 ms
Execution Time: 625.045 ms
```

Initially, a sequential scan is done over the ProductionCompany table, a hash key based on *pc.pc_id* is used (because it is used in the JOIN condition) to insert each row in a hash table. The data is divided into 131072 buckets naturally as it's the power of 2 just larger than 80,000 and there 80,000 unique pc_id's.

After the hash table is built, a Bitmap Scan is performed over the movie table. Here the movie table acts as an outer relation and the productioncompany table acts as an inner relation.

Bitmap Index Scan fetches all the tuple-pointers from the *movie_year* index with index condition *((year >= 1950) and (year <= 2000))* in one go, sorts them using an in-memory "bitmap" data structure, and then visits the table rows in physical row-location order. For each row in the movie table, the hash key is calculated based on the movie.pc_id as this is present in the JOIN condition. If the calculated

hash key is present in the hash table, the *productioncompany* row in the hash bucket table and movie row will be returned.

(Why is the *productioncompany* table hashed but not the *movie* table? The following section explains why *Bitmap Scan* is used on the *movie* table rather than *Index Scan*).

Briefly justify the query optimiser choices (choice of query processing algorithms)

Query1

To begin, Hash Join is used to join the casting and actor tables. To begin, we must choose actors with an *a_id* < 50. Because the number of actors who meet this requirement is relatively small (47 rows), *Index Scan* is employed. Because this table is smaller, it can be used as a construct relation to produce hash buckets, reducing the amount of memory required for the hash table and allowing the complete hash table to fit in the main memory. Then, to efficiently join both tables, a *concurrent sequential scan* is performed.

Because the expected number of rows is less, a *nested loop* with *Index scan* is used to merge the movie table with the result of the *hash join* of the actor and casting tables (627).


Gather is used to combine the output (generated due to *parallel sequential scan*). Two workers assist in the gathering process.

Query2

Query optimizer decides to go with Nested Loop as the expected number of rows(184) are very less.

In the casting table, the expected number of rows (i.e rows that fulfil *m_id* < 50) are 184. The expected number of rows are too less to perform *Seq Scan* or *Bitmap Heap Scan* (because it would be expensive to scan all rows which are not at all required).

Casting.a_id is required to filter rows from the actor table. And we are using *m_id* in the first where condition. Since we can get both fields in the *casting_pkey* index nodes, without actually making table access from the leaf nodes, that is why *Index Only Scan* is done using *casting_pkey* instead of *Index Scan* using *casting_m_id*.



Now coming to the reason why is *Index Scan* used on the actor table and *Index Only Scan* on the casting table. This is done simply because the expected number of rows or result table size in the actor table (1 row) is less than the size of the casting table (184 rows).

Query3

Query optimizer decided to go with the Hash Join strategy as the expected number of rows(428633) is large.

Here productioncompany table is hashed instead of the movie table because the size of the productioncompany table is much smaller.

The expected number of rows in the movie table that satisfies (*imdbscore* < 1.5) are around 42% of the total rows. Therefore it is less costly to choose *sequential scan* over *index scan* (because traversing the index to find all matching entries and then fetching them from the table would prove to be more expensive). Besides, even if the query planner decided to go with the index scan or bitmap scan then it has to use the *movie_imdbscore* index which only contains 41 nodes (since there are 41 unique values of *imdbscore*), and each of its nodes contains around 10000 rows. So when fetching these rows from the table, the cost of random I/O access would have proved more expensive than a simple sequential scan.

Query4

Query optimizer decided to go with the Hash Join strategy as the expected number of rows(511066) is large.

Here productioncompany table is hashed instead of the movie table because the size of the productioncompany table is much smaller.

Bitmap Scan is used on movies because the expected number of rows is too large for an Index Scan. It is better than index scan in this case as it doesn't have to do random memory accesses (where we have to request row(s) (memory) for each index that satisfies the filter in a random way).