Contents

1 Part A: Experimenting with Query Selectivity

1.1 Queries to analyze Selectivity

The following range queries are used to experiment and analyze the behaviour of the query optimizer for queries with different selectivity:

Q1: SELECT name FROM movie WHERE imdb_score < 2;
Q2: SELECT name FROM movie WHERE imdb_score between 1.5 and 4.5;
Q3: SELECT name FROM movie WHERE year between 1900 and 1990;
Q4: SELECT name FROM movie WHERE year between 1990 and 1995;
Q5: SELECT * FROM movie WHERE pc_id < 50;
Q6: SELECT * FROM movie WHERE pc_id > 20000;

1.2 Query Outputs and Analysis

The following section contains the output of the queries in section 1.1 and brief explanation about the query strategy adopted by the query optimizer in each of them.

1. SELECT name FROM movie WHERE imdb_score < 2;

```
postgres=# explain analyze SELECT name FROM movie WHERE imdb_score < 2;
                                                        QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on movie  (cost=6860.64..24510.48 rows=234867 width=11) (actual time=28.458..140.251 rows=237910 loops=1)
   Recheck Cond: (imdb_score < '2'::numeric)
   Heap Blocks: exact=7354
   ->  Bitmap Index Scan on idx_movie_imdb  (cost=0.00..6801.93 rows=234867 width=0) (actual time=27.017..27.017 rows=237910 loops=1)
         Index Cond: (imdb_score < '2'::numeric)
 Planning Time: 0.146 ms
 Execution Time: 161.130 ms
(7 rows)
```

Explanation: At first, the query plan consists of performing a Bitmap Index Scan on the *idx_movie_imdb* index. For this, the bitmap index scan constructs a bitmap of potential row locations, using the Index Condition(*imdb_score* < 2).
The output of the Index Scan on *idx_movie_imdb* index is passed up to a parent Bitmap Heap Scan on *movie*. Using the received data, the Bitmap Heap Scan looks up for the required rows. A bitmap can have either exact chunks(pointing directly to rows) or lossy chunks(pointing to a page containing at least one row satisfying the given predicate). In case the bitmap becomes too large, it is converted to a "lossy" style, where only the pages

containing the matching tuples are remembered instead of remembering each individual tuple. Consequently, during the table-visiting phase, each tuple on the page has to be rechecked against the scan condition($imdb\_score < 2$) to determine the final tuples to be returned as query output.

2. SELECT name FROM movie WHERE imdb_score between 1.5 and 4.5;

```
postgres=# explain analyze SELECT name FROM movie WHERE imdb_score between 1.5 and 4.5;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Seq Scan on movie  (cost=0.00..29714.00 rows=771633 width=11) (actual time=116.623..708.967 rows=775469 loops=1)
   Filter: ((imdb_score >= 1.5) AND (imdb_score <= 4.5))
   Rows Removed by Filter: 224531
 Planning Time: 1.127 ms
 Execution Time: 756.478 ms
(5 rows)
```

Explanation: Here, the optimizer uses sequential scan on *movie*, with a filter condition of (($imdb\_score >= 1.5$) AND ($imdb\_score <= 4.5$)). The rows removed by the filter are 224531 and the result rows are 775469.

3. SELECT name FROM movie WHERE year between 1900 and 1990;

```
postgres=# explain analyze SELECT name FROM movie WHERE year between 1900 and 1990;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on movie  (cost=3157.88..20625.24 rows=183557 width=11) (actual time=15.249..72.308 rows=181730 loops=1)
   Recheck Cond: ((year >= 1900) AND (year <= 1990))
   Heap Blocks: exact=7354
   ->  Bitmap Index Scan on idx_movie_year  (cost=0.00..3111.99 rows=183557 width=0) (actual time=13.796..13.796 rows=181730 loops=1)
         Index Cond: ((year >= 1900) AND (year <= 1990))
 Planning Time: 1.709 ms
 Execution Time: 82.223 ms
(7 rows)
```

Explanation: At first, the query plan consists of performing a Bitmap Index Scan on the *idx_movie_year* index. For this, the bitmap index scan constructs a bitmap of potential row locations, using the Index Condition(($year >= 1900$) AND ($year <= 1990$)).
The output of the Index Scan on *idx_movie_year* index is passed up to a parent Bitmap Heap Scan on *movie*. Using the received data, the Bitmap Heap Scan looks up for the required rows, with the recheck condition same as the index scan condition.

4. SELECT name FROM movie WHERE year between 1990 and 1995;

```
postgres=# explain analyze SELECT name FROM movie WHERE year between 1990 and 1995;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Seq Scan on movie  (cost=0.00..29714.00 rows=492658 width=11) (actual time=82.658..321.832 rows=490598 loops=1)
   Filter: ((year >= 1990) AND (year <= 1995))
   Rows Removed by Filter: 509402
 Planning Time: 1.436 ms
 Execution Time: 349.318 ms
(5 rows)
```

Explanation: Here, the optimizer again uses the sequential scan technique on *movie*, with a filter condition of ((*year* >= 1990) AND (*year* <= 1995)). The rows removed by the filter are 509402 and the returned rows are 490598.

5. SELECT * FROM movie WHERE pc_id < 50;

```
postgres=# explain analyze SELECT * FROM movie WHERE pc_id < 50;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on movie  (cost=12.86..1938.50 rows=572 width=29) (actual time=2.251..2.829 rows=604 loops=1)
   Recheck Cond: (pc_id < 50)
   Heap Blocks: exact=588
   ->  Bitmap Index Scan on idx_movie_pcid  (cost=0.00..12.71 rows=572 width=0) (actual time=2.160..2.160 rows=604 loops=1)
         Index Cond: (pc_id < 50)
 Planning Time: 1.559 ms
 Execution Time: 2.890 ms
(7 rows)
```

Explanation: At first, the query plan consists of performing a Bitmap Index Scan on the *idx_movie_pcid* index. For this, the bitmap index scan constructs a bitmap of potential row locations, using the Index Condition(*pc_id* < 50). The output of the Index Scan on *idx_movie_pcid* index is passed up to a parent Bitmap Heap Scan on *movie*. Using the received data, the Bitmap Heap Scan looks up for the required rows, with the recheck condition same as the index scan condition.

6. SELECT * FROM movie WHERE pc_id > 20000;

```
postgres=# explain analyze SELECT * FROM movie WHERE pc_id > 20000;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Seq Scan on movie  (cost=0.00..27214.00 rows=753183 width=29) (actual time=75.295..378.572 rows=750165 loops=1)
   Filter: (pc_id > 20000)
   Rows Removed by Filter: 249835
 Planning Time: 1.001 ms
 Execution Time: 426.079 ms
(5 rows)
```

Explanation: Here, the optimizer uses sequential scan on *movie*, with a filter condition of (*pc_id* > 20000). The rows removed by the filter are 249835 and the result rows are 750165.

1.3 Index-based comparison: Q1 and Q2

In this section, we compare the decision of query optimizer to use the index on *imdb_score* for Q1 and Q2. In Q1, the query optimizer employed the bitmap index scan on *imdb_score* followed by the bitmap heap scan on *movie* to determine the result of the query, whereas in Q2, a sequential scan on movie was employed by the query optimizer. The rationale for this choice can be explained on the basis of query selectivity for the two queries. The estimated number of tuples to be returned by the Q1 were 234867, which is significantly lower than the number estimated by the query optimizer for Q2, which is 771633.

Hence, the query optimizer selected an index-based scan for Q1, as the number of tuples estimated to be touched by the range query was much lower than the total number of tuples in the *movie* relation. On the other hand, the query optimizer preferred a sequential scan for Q2, as index-based scan would not have been advantageous to perform due to the estimated number of tuples being close to the total number of tuples in the *movie* relation.

1.4 Index-based comparison: Q3 and Q4

In this section, we compare the decision of query optimizer to use the index on *year* for Q3 and Q4. In Q3, the query optimizer employed the bitmap index scan on *year* followed by the bitmap heap scan on *movie* to determine the result of the query, whereas in Q4, a sequential scan on movie was employed by the query optimizer. The difference in the decisions of the query optimizer can be explained on the basis of query selectivity for the two queries. The estimated number of tuples to be returned by the Q3 were 183557, which is significantly lower than the number estimated by the query optimizer for Q4, which is 492658.

Hence, an index-based scan was employed by the query optimizer for Q3, as the number of tuples estimated to be touched by the range query was much lower than the total number of tuples in the *movie* relation. On the other hand, the query optimizer decided to perform a sequential scan for Q4, as performing an index-based scan would not have been advantageous due to the relatively large number of estimated tuples.

1.5 Index-based comparison: Q5 and Q6

In this section, we compare the decision of query optimizer to use the index on *pc_id* for Q5 and Q6. In Q5, the query optimizer employed the bitmap index scan on *pc_id* followed by the bitmap heap scan on *movie* to determine the result of the query, whereas in Q6, a sequential scan on movie was employed by the query optimizer. The rationale for this choice can be explained on the basis of query selectivity for the two queries. The estimated number of tuples to be returned by the Q5 were 572, which is significantly lower than the number estimated by the query optimizer for Q6, which is 753183.

Hence, the query optimizer selected an index-based scan for Q5, as the number of tuples estimated to be touched by the range query is much lower than the total number of tuples in the *movie* relation. On the other hand, the query optimizer selected a sequential scan for Q6, as index-based scan would not have been advantageous to perform due to the estimated number of tuples being quite close to the total number of tuples in the *movie* relation.

2 Part B: Join Strategies

2.1 Queries to analyze Join Strategies

The following range queries are used to experiment and analyze the behaviour of the query optimizer for queries with different selectivity:

Q1: Join Actor, Movie and Casting; Where a_id < 50. Finally, the query outputs actor name and movie name.
Q2: Join Actor and Casting; Where m_id < 50. Finally, the query outputs actor name.
Q3: Join Movie and Production Company; where imdb_score is less than 1.5. Finally, the query outputs the movie name and production company.
Q4: Join Movie and Production Company; where year is between 1950 and 2000. Finally, the query outputs the movie name and production company.

2.2 Query Outputs and Analysis

The following section contains the output of the queries in section 2.1 and brief explanation about the query strategy adopted by the query optimizer in each of them.

1. SELECT actor.name, movie.name
   FROM actor, movie, casting
   WHERE casting.m_id = movie.m_id
   AND casting.a_id = actor.a_id
   AND actor.a_id < 50;

```
postgres=# explain analyze select actor.name, movie.name from actor, movie, casting where casting.m_id = movie.m_id and casting.a_id = actor.a_id and actor.a_id < 50;
                                                             QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=1.28..2935.20 rows=584 width=27) (actual time=0.695..18.905 rows=644 loops=1)
   ->  Nested Loop  (cost=0.85..2665.01 rows=584 width=20) (actual time=0.306..1.928 rows=644 loops=1)
         ->  Index Scan using idx_actor_aid on actor  (cost=0.42..9.19 rows=44 width=20) (actual time=0.270..0.290 rows=49 loops=1)
               Index Cond: (a_id < 50)
         ->  Index Scan using idx_casting_aid on casting  (cost=0.43..60.22 rows=14 width=8) (actual time=0.005..0.030 rows=13 loops=49)
               Index Cond: (a_id = actor.a_id)
   ->  Index Scan using movie_pkey on movie  (cost=0.42..0.46 rows=1 width=15) (actual time=0.026..0.026 rows=1 loops=644)
         Index Cond: (m_id = casting.m_id)
 Planning Time: 13.393 ms
 Execution Time: 20.144 ms
(10 rows)
```

Explanation: In the query plan, there are two Nested Loop nodes. The process can be translated as follows:

(a) **Inner Nested Loop Node:**

- First, determine all the records in *actor* table which satisfy the condition *a_id < 50*, using an Index Scan on index *idx_actor_aid*.
- Then, for each of the record obtained from the *actor* table in the previous step, iterate through *casting* table to find out the records that satisfy the condition *casting.a_id = actor.a_id* using Index Scan on index *idx_casting_aid*. *a_id* in casting table is a foreign key referencing *a_id* in the *actor* table.
- This joins *actor* and *casting* relations with the join condition *actor.a_id = casting.a_id* and the filter condition *actor.a_id < 50*.

(b) **Outer Nested Loop Node:**

- For each tuple obtained as an output of the **Inner Nested Loop Node**, iterate through *movie* table to determine the tuples satisfying the condition *movie.m_id = casting.m_id* using an Index Scan on index *movie_pkey*. *m_id* is the primary key of *movie* relation. *m_id* in casting table is a foreign key referencing *m_id* in the *movie* table.
- Here, the previously joined tables(*in (a)*) are joined with the *movie* relation under the join condition *movie.m_id = casting.m_id*.

2. SELECT actor.name
   FROM actor, casting
   WHERE actor.a_id = casting.a_id
   AND casting.m_id < 50;

```
postgres=# explain analyze select actor.name from actor, casting where actor.a_id = casting.a_id and casting.m_id < 50;
                                                     QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.85..1601.76 rows=204 width=16) (actual time=0.099..2.579 rows=196 loops=1)
   ->  Index Only Scan using casting_pkey on casting  (cost=0.43..8.00 rows=204 width=4) (actual time=0.020..0.062 rows=196 loops=1)
         Index Cond: (m_id < 50)
         Heap Fetches: 0
   ->  Index Scan using idx_actor_aid on actor  (cost=0.42..7.81 rows=1 width=20) (actual time=0.012..0.012 rows=1 loops=196)
         Index Cond: (a_id = casting.a_id)
 Planning Time: 4.283 ms
 Execution Time: 2.620 ms
(8 rows)
```

Explanation: The query plan consists of a Nested Loop Join, wherein the following happens:

- First, search for all the records in the *casting* table using Index Only Scan on index *casting_pkey* with the index condition of *casting.m_id <*

50. In Index Only Scan, there is no table access needed because the index has all the columns required to satisfy the query.

- Then, for each of the record obtained from *casting* table in the former step, iterate through *actor* table to determine the tuples which satisfy the condition *actor.a_id = casting.a_id* using Index Scan on index *idx_actor_aid*. Here, *a_id* in the casting table is a foreign key referencing *a_id* in the *actor* table.
- Here, *actor* and *casting* table are joined under the join condition *actor.a_id = casting.a_id* and the filter condition *casting.m_id < 50*.

3. SELECT movie.name, production_company.name
   FROM movie, production_company
   WHERE movie.pc_id = production_company.pc_id
   AND movie.imdb_score < 1.5;

```
postgres=# explain analyze select movie.name, production_company.name from movie, production_company where movie.pc_id = production_company.pc_id and movie.imdb_score < 1.5;
                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=6216.20..24112.72 rows=112100 width=22) (actual time=49.907..147.092 rows=112419 loops=1)
   Hash Cond: (movie.pc_id = production_company.pc_id)
   ->  Bitmap Heap Scan on movie  (cost=3277.20..19392.45 rows=112100 width=15) (actual time=17.428..50.842 rows=112419 loops=1)
         Recheck Cond: (imdb_score < 1.5)
         Heap Blocks: exact=7354
         ->  Bitmap Index Scan on idx_movie_imdb  (cost=0.00..3249.18 rows=112100 width=0) (actual time=16.012..16.012 rows=112419 loops=1)
               Index Cond: (imdb_score < 1.5)
   ->  Hash  (cost=1548.00..1548.00 rows=80000 width=15) (actual time=30.700..30.701 rows=80000 loops=1)
         Buckets: 131072  Batches: 2  Memory Usage: 2900kB
         ->  Seq Scan on production_company  (cost=0.00..1548.00 rows=80000 width=15) (actual time=0.028..10.800 rows=80000 loops=1)
 Planning Time: 2.713 ms
 Execution Time: 152.059 ms
(12 rows)
```

Explanation: The query plan consists of Hash Join, which can be described as follows:

(a) **Build Phase:**

- Perform a sequential scan through *production_company* table. For each record in this table, calculate a hash key based on *pc_id* attribute of *production_company*.
  Here, *production_company.pc_id* is used for the purpose of hashing since it is used in the JOIN condition.
- Simultaneously, insert each record of *production_company* relation in a hash table (shown in Hash node in the query plan) using the calculated hash key.

(b) **Probe Phase:**

- The query plan consists of performing a Bitmap Index Scan on the *idx_movie_imdb* index. For this, the bitmap index scan constructs a bitmap of potential row locations, using the Index Condition($imdb\_score < 1.5$).
  The output of the Index Scan on *idx_movie_imdb* index is passed up to a parent Bitmap Heap Scan on *movie*. Using the received data, the Bitmap Heap Scan looks up for the required rows, with the recheck condition same as the index scan condition.
- Next, for each record obtained as an output of the Bitmap Heap Scan, a hash key is calculated using the value of *movie.pc_id*, since the JOIN condition contains the attribute *movie.pc_id*. If the calculated hash key is present in the hash table, the records of *production_company* in the corresponding hash table bucket are mapped to this *movie* record, and returned as output.
- This joins *movie* and *production_company* relations with the join condition *movie.pc_id = production_company.pc_id* and the filter condition *movie.imdb_score < 1.5*.

4. SELECT movie.name, production_company.name
   FROM movie, production_company
   WHERE movie.pc_id = production_company.pc_id
   AND movie.year BETWEEN 1950 AND 2000;

```
postgres=# explain analyze select movie.name, production_company.name from movie, production_company where movie.pc_id = production_company.pc_id and movie.year between 1950 and 2000;
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=2939.00..44785.29 rows=947490 width=22) (actual time=94.274..736.893 rows=944575 loops=1)
   Hash Cond: (movie.pc_id = production_company.pc_id)
   ->  Seq Scan on movie  (cost=0.00..29714.00 rows=947490 width=15) (actual time=63.776..256.680 rows=944575 loops=1)
         Filter: ((year >= 1950) AND (year <= 2000))
         Rows Removed by Filter: 55425
   ->  Hash  (cost=1548.00..1548.00 rows=80000 width=15) (actual time=29.680..29.682 rows=80000 loops=1)
         Buckets: 131072  Batches: 2  Memory Usage: 2900kB
         ->  Seq Scan on production_company  (cost=0.00..1548.00 rows=80000 width=15) (actual time=0.023..10.662 rows=80000 loops=1)
 Planning Time: 2.691 ms
 Execution Time: 768.300 ms
(10 rows)
```

Explanation: The query plan consists of Hash Join wherein, the following happens:

(a) **Build Phase:**

- Perform a sequential scan through *production_company* table. For each record in this table, calculate a hash key based on *pc_id* attribute of *production_company*.
  production_company.pc_id is used for hash key here because it is used in the JOIN condition.

- Simultaneously, insert each record of *production_company* relation in a hash table (shown in Hash node in the query plan) using the calculated hash key.

(b) **Probe Phase:**

- Do a sequential scan through *movie* relation under the filter condition *((year >= 1950) AND (year <= 2000))*.
- For each filtered record, a hash key is calculated based on the *movie.pc_id* because JOIN condition contains the attribute *movie.pc_id*. If the calculated hash key is present in the hash table, the records of *production_company* in the corresponding hash table bucket are mapped to this *movie* record, and returned as output.
- This joins *movie* and *production_company* relations with the join condition *movie.pc_id = production_company.pc_id* and the filter condition *((year >= 1950) AND (year <= 2000))*.

2.3 Justification of the Query Optimizer's Choices

2.3.1 Use cases for different JOIN strategies in PostgreSQL

1. **Nested Loop Join**: Nested loop joins are particularly used in the following scenario:-

   - If the outer relation is small; resulting in lesser number of loops on the inner relation.
     In case the outer relation is large, nested loop joins are generally very inefficient, though they may be supported by an index on the inner relation.
   - Only join strategy can be used if the join condition does not use the "=" operator, thereby serving as a fall-back strategy, in situations when no other strategy can be used.
   - As the outer relation is scanned sequentially, therefore an index on the outer relation will not help in improving the query performance. However, an index on the join attribute of the inner relation can improve the performance of a nested loop join considerably(**Single Loop Join**).

2. **Hash Join**: Hash Joins are particularly useful in the following scenario:-

   - If neither of the involved relations is small enough, however the hash table for the smaller table can be accommodated in the working memory.

- At least one join condition has "=" operator.

3. **Merge Join:** Merge Joins are generally used in the following use case:-
   - If the involved relations are both so big that a hash table generated on either of them would not fit into the working memory.
     This strategy is particularly used for joining really large tables.
   - Similar to hash join, at least one join condition should contain the "=" operator.

2.3.2 Explanation for the queries

1. In Q1,
   (a) For the **Inner Join Node**:
   - The cost model for Nested Loop Join with Index Scan on a non-primary key(of inner relation) is given by $b_R + |R| * (height + s_B + \lceil \frac{s_B}{BFR\ s_B} \rceil)$ *block accesses*, where
     b_R = #Blocks containing records from outer relation R.
     $|R|$ = Number of tuples in outer relation R.
     height = Height of the *B+ tree* index on the required non-primary attribute of relation S(inner relation).
     s_B = The expected #records for a unique value.
     BFR = Blocking Factor(number of records per block)
   - The choice of query optimizer to employ the nested loop join algorithm can be justified on the basis of the query selectivity of the condition *actor.a_id < 50*.
     The query optimizer estimated the number of rows as an output of the query to be 44, resulting in a quite small outer relation, thereby favouring the selection of a Nested loop join which is particularly efficient if the outer relation is small, because then the inner loop won't be executed too often, thereby, reducing the number of block accesses. This is also supported by the cost model expression.
   - An index scan on *idx_actor_aid* is used to determine the tuples of the outer relation actor, owing to the low selectivity of the range query *actor.a_id < 50*.
   - An index scan on *idx_casting_aid* is used on the join key *a_id* of the inner relation *casting*, because it can speed up a nested loop join considerably.

- From the cost model, it is evident that the outer relation should be the smaller one. This justifies the choice of the query optimizer for the selection of outer and inner relations in our case.
  In our case, the table returned by the index scan on the *actor* table has an estimated size of 44 rows and is taken as the outer relation. The *casting* relation with 4000000 tuples, is taken as the inner relation.

(b) For the **Outer Join Node**:

- The cost model for Nested Loop Join with Index Scan on primary key(of inner relation) is given by $b_R + |R| * (height + 1)$ *block accesses*, where
  b_R = #Blocks containing records from outer relation R.
  $|R|$ = Number of tuples in outer relation R.
  height = Height of the *B+ tree* index on the required primary attribute of relation S(inner relation).
- Again a Nested loop join is used by the query optimizer for performing the join between the result of the Inner Nested Loop Join(outer relation) and the *movie* relation(inner relation). This is due to the small size of the outer relation(= 584 rows) and the performance benefits of selecting Nested Loop Join when the outer relation is small, also supported by the cost model expression.
- Similar to (a), an index scan on *movie_pkey* is used on the join key *m_id* of the inner relation *movie*, because it can speed up a nested loop join considerably.
- From the cost model, we can discern that the outer relation should be the smaller one. This justifies the choice of the query optimizer for the selection of outer and inner relations in our case.
  In our case, the table returned by the inner nested loop join has an estimated size of 584 rows and is taken as the outer relation. The *movie* relation with 1000000 tuples, is taken as the inner relation.

2. In Q2,

- The cost model for Nested Loop Join with Index Scan on primary key(of inner relation) is given by $b_R + |R| * (height + 1)$ *block accesses*.
- The choice of query optimizer to employ the nested loop join algorithm can be justified on the basis of the query selectivity of the condition

casting.m_id < 50.
The query optimizer estimated the number of rows as an output of the query to be 204, resulting in a quite small outer relation, thereby favouring the selection of a Nested loop join which is particularly efficient if the outer relation is small, because then the inner loop won't be executed too often, also evident from the cost model expression.

- An index only scan on *casting_pkey* is used to determine the tuples of the outer relation casting, owing to the low selectivity of the range query *casting.m_id* < 50.
- An index scan on *idx_actor_aid* is used on the join key *a_id* of the inner relation *actor*, because it can speed up a nested loop join considerably.
- From the cost model, we can discern that the outer relation should be the smaller one. This justifies the choice of the query optimizer for the selection of outer and inner relations in our case.
  In our case, the table returned by the index only scan on the *casting* table has an estimated size of 204 rows and is taken as the outer relation. The *actor* relation with 300000 tuples, is taken as the inner relation.

3. In Q3,

   (a) **For Hash Join**

   - A plausible reason for the selection of Hash Join for this query is that none of the involved relations are small(estimated relation sizes: 112100 tuples and 80000 tuples), but the hash table of the build relation(smaller relation) could fit in the working memory.
   - The table returned by the sequential scan on the *production_company* relation is taken as the build relation for the Hash Join.
   - The table returned by the bitmap heap scan on the *movie* relation is taken as the probe relation for the Hash Join.
   - The rationale behind the selection of the build and the probe relation is the relation size (also dependent on the query selectivity).
   - Since, the estimated size of the output(= 80000 tuples) returned by the sequential scan on the *production_company* is less than the estimated size of the output(= 112100 tuples) returned by the bitmap heap scan on the *movie* relation, therefore, the former is taken as the build relation and the latter as the probe relation.

(b) **Other query optimizer choices**

- For creating a Hash table on the *production_company* table, a sequential scan is performed because all the tuples of the *production_company* have to be considered.
- A bitmap index scan followed by a bitmap heap scan is performed on the *movie* relation because the estimated number of tuples/rows(= 112100 rows) is very less as compared to the total number of rows in the *movie* relation(= 1000000 rows).

4. In Q4,

(a) **For Hash Join**

- A plausible reason for the selection of Hash Join for this query is that none of the involved relations are small(estimated relation sizes: 947490 tuples and 80000 tuples), but the hash table of the build relation(smaller relation) could fit in the working memory.
- The table returned by the sequential scan on the *production_company* relation is taken as the build relation for the Hash Join.
- The table returned by the sequential scan on the *movie* relation under the filter *((year >= 1950) AND (year <= 2000))* is taken as the probe relation for the Hash Join.
- The rationale behind the selection of the build and the probe relation is the relation size (also dependent on the query selectivity).
- Since, the estimated size of the output(= 80000 tuples) returned by the sequential scan on the *production_company* is much less than the estimated size of the output(= 947490 tuples) returned by the sequential scan (with filter) on the *movie* relation, therefore, the former is taken as the build relation and the latter as the probe relation.

(b) **Other query optimizer choices**

- For creating a Hash table on the *production_company* table, a sequential scan is performed because all the tuples of the *production_company* have to be considered.
- The query optimizer decided to perform a sequential scan on *movie* relation, as performing an index-based scan would not have been advantageous due to the relatively large number of estimated tuples(= 947490 tuples).