CS301: Databases

Course project (Phase B) report

Group 7

Team members:

1. Shivam Pandey (2019CHB1055)
2. Deepan Maitra (2019CSB1044)

Table of contents:

1. Part A
   - ➢ Question 2 (page 3)
   - ➢ Question 3 (page 5)
   - ➢ Question 4 (page 5)
   - ➢ Question 5 (page 6)
2. Part B
   - ➢ Question 2 (page 7)
   - ➢ Question 3 (page 11)

Part A: Experimenting with query selectivity

As far as the interpretations of the query plan is concerned, there were several factors we needed to consider: query selectivity, number of rows accessed, scanning strategies, number of block accesses, amount of disk space utilized, I/O accesses, cost overhead, join strategies, range queries, B+ tree indexing, searching on key and non-key attributes and several usages of B+ tree indexes.

1) Queries were run on the generated database.

2) The following are the query plans for each of the given commands and our interpretation of the same:

   1. **SELECT name FROM movie WHERE imdb_score < 2 ;**

```
                                                       QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
Bitmap Heap Scan on movie  (cost=4684.89..22514.21 rows=249866 width=11) (actual time=81.161..28532.082 rows=249308 loops=1)
  Recheck Cond: (imdb_score < '2'::numeric)
  Heap Blocks: exact=7354
  ->  Bitmap Index Scan on movie_imdb  (cost=0.00..4622.42 rows=249866 width=0) (actual time=79.004..79.005 rows=249308 loops=1)
        Index Cond: (imdb_score < '2'::numeric)
Planning Time: 3.414 ms
Execution Time: 28561.880 ms
```

The SQL query here is a scan across the movie table, to find rows which match the range condition: the imdb_score less than 2. This can be done by a normal *sequential scan*, which goes to each and every row of the table and outputs the ones which pass the condition (this is the normal '*filter*' condition in the WHERE clause). But the query optimizer actually decides to *Bitmap Heap scan*. For queries whose outputs have very high query selectivity (most rows pass the condition) and we need the entire row data (and not just the key value) then sequential scan is chosen. However when there is a lower selectivity (here almost 2.5 lakh rows passed the condition out of 10 lakh movies, which is a 25% selectivity), Bitmap scan actually reduces the cost overhead since not all the rows of the table have to be fetched and read repeatedly, regardless of the condition.

Here, firstly there is a bitmap index scan going on the movie_imdb index (index defined on the 'imdb_score' attribute of the movie table) which finds all the locations of rows matching the index condition. After that, the upper plan (*Bitmap heap scan* on movie table) does a sorting of those row locations using the bitmap data structure, so as to minimize the overhead of random and individual fetching. This also preserves the locality of fetching. Fetches from locations close to one another are done together, reducing access time. After this, the row data is fetched from the passed locations (with a '*Recheck*' condition to verify). Thus, a two-level query plan is chosen: an *index-scan* and then a *bitmap heap scan*, in contrast to a direct filter *sequential scan*.

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

```
                                         QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
Seq Scan on movie  (cost=0.00..29706.00 rows=751649 width=11) (actual time=7237.262..21806.000 rows=751599 loops=1)
  Filter: ((imdb_score >= 1.5) AND (imdb_score <= 4.5))
  Rows Removed by Filter: 248401
Planning Time: 25.247 ms
```

This query plan is very highly selective (almost 7.5 lakh rows are passed out of 10 lakh movies, which is 75% selectivity) and we require the name attribute of the row, so *sequential scan* works best. The '*filter*' checks for the given range of imdb_score, scans through all the rows sequentially and then displays the rows which pass the condition. Doing an *index scan* would have resulted in too many I/O operations for each row (look up the row from the index, and then retrieve the row from the heap), but the sequential scan does only one I/O operation for all the rows which can be accommodated in a single block. Also in the case of bitmap index scan, the leaf nodes have to store the corresponding records having that particular imdb_score and subsequently later on have to scan the corresponding blocks in order to fetch the tuples which satisfy the given condition. This adds up quite significantly to the total cost (a type of non-key attribute searching).

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

```
                                         QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
Seq Scan on movie  (cost=0.00..29706.00 rows=900000 width=11) (actual time=6795.956..14846.265 rows=900879 loops=1)
  Filter: ((year >= 1900) AND (year <= 1990))
  Rows Removed by Filter: 99121
Planning Time: 395.338 ms
Execution Time: 14944.227 ms
```

Here also the query selectivity for the range query is very high (9 lakh rows passed out of 10 lakh, which is 90% selectivity) and hence to minimise the I/O overhead, sequential scan is chosen. Since most of the rows pass through the filter, whenever a block is fetched, most of the rows present in the block will pass the condition--hence will give much less fetch/access overhead.

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

```
                                         QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
Bitmap Heap Scan on movie  (cost=751.16..16280.68 rows=54901 width=11) (actual time=524.800..576.425 rows=59437 loops=1)
  Recheck Cond: ((year >= 1990) AND (year <= 1995))
  Heap Blocks: exact=7353
  ->  Bitmap Index Scan on movie_year  (cost=0.00..737.43 rows=54901 width=0) (actual time=521.212..521.213 rows=59437 loops=1)
        Index Cond: ((year >= 1990) AND (year <= 1995))
Planning Time: 15.893 ms
Execution Time: 610.852 ms
```

For this query, with significant low selectivity (54k rows out of 10 lakhs passed, which is 5.4% selectivity) doing a sequential scan is a waste, since there is no use of iterating through all the rows, most of which won't be passed. Hene, Bitmap scan is opted for. First, there is an index scan on the movie_year index (index of 'year' attribute of movie table) which returns all the row locations passing the index condition. Then the Bitmap heap scan actually fetches the rows from the heap locations (after sorting them and arranging them to reduce the number of fetches). Although for each row passed we have to first do index scanning and then do heap fetches, the number of rows passed overall is very less and so this seems to be the better option.

5. **SELECT * FROM movie WHERE prod_company < 50;**

```
                                                         QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
Bitmap Heap Scan on movie  (cost=1039.55..16840.91 rows=87629 width=29) (actual time=343.090..390.743 rows=87922 loops=1)
  Recheck Cond: (prod_company < 50)
  Heap Blocks: exact=6618
  ->  Bitmap Index Scan on movie_pcid  (cost=0.00..1017.64 rows=87629 width=0) (actual time=341.572..341.572 rows=87922 loops=1)
        Index Cond: (prod_company < 50)
Planning Time: 47.360 ms
Execution Time: 396.455 ms
```

Here also with 8.7% query selectivity, index scan on the movie_pcid index is opted for, followed by the Bitmap heap scan. It must be noted that the index scanning and bit-map scanning is possible only because B+ trees exist for the attributes being searched for/used as ranges. If the 'pc_id' attribute here wouldn't have had a B+ tree indexing, then such an index scanning would have been impossible.

6. **SELECT * FROM movie WHERE prod_company > 20000;**

```
                                                         QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
Bitmap Heap Scan on movie  (cost=878.64..16510.79 rows=74092 width=29) (actual time=7.642..22.067 rows=75605 loops=1)
  Recheck Cond: (prod_company > 20000)
  Heap Blocks: exact=737
  ->  Bitmap Index Scan on movie_pcid  (cost=0.00..860.11 rows=74092 width=0) (actual time=7.508..7.514 rows=75605 loops=1)
        Index Cond: (prod_company > 20000)
Planning Time: 0.202 ms
Execution Time: 25.915 ms
```

Here also with **7.4% query selectivity**, index scan on the *movie_pcid index* is opted for, followed by the Bitmap heap scan. The total cost of fetching the required leaf nodes and later on fetching the corresponding records from their respective heaps is quite less in comparison to sequentially scanning all the records and fetching the data satisfying the condition. Also if the indexes are in sorted fashion, then the process turns out to be quite efficient as the leaf nodes are connected to each other via pointers and accessing the required nodes will turn out to be faster in the case of index scan.

3) For **Q1** (*imdb_score* < *2*) the index on the *imdb_score* attribute (*movie_imdb*) is used since there is a lower query selectivity (**25%**) and the query optimiser is opting for a Bitmap index scan followed by Bitmap Heap scan.

   Whereas for **Q2** since there is a higher selectivity (**75%** selectively), the query optimiser is opting for a sequential scan. For Q2, the other reason for opting sequential scan could be that we have to access the name of the *movie* rather than accessing only the *imdb_score*. Thus, if we had to go ahead with index scan there would have been additional scanning done to access the heap files (containing the whole information regarding the key) of the keys shortlisted after the first scan. So, it would have added to additional costs. Additionally, the resulting bitmap would have one bit for each heap page, where the bits are true only if they were true in all the individual bitmap index scans (the search condition would have matched for every index scan) and these are the heap pages which would have been needed to load and examine. Since each heap page could have contained multiple rows, the query optimizer would have to examine each row to see if it matches all the conditions ('recheck' part of the query plan). Therefore, for so many rows this would have resulted in a large I/O overhead. Also the random I/O operation in the bitmap index scan before sorting would also contribute significantly to the total costs. Thus, all these factors collectively led to opting for sequential scan by the query optimiser for Q2. In Q1, the lower query selectivity overrode these factors since the number of rows being considered was significantly less. So although the number of I/O operations (look up the row from the index, and then retrieve the row from the heap) were more for each row, the total estimated cost was much less.

4) For **Q3** (90% query selectivity) sequential scan is being used by the optimiser which is obvious since any other scan seems unnecessary for such a high selectivity. For **Q4** (5.4% selectivity) a bitmap heap scan is being done. We ran the following range queries to check the number of rows being passed using the count(*) commands:

Range of 'year'	No. of rows passed	Optimizer Query plan
1900 < year <1920	207651	Bitmap heap scan
1920 < year <1925	59359	Bitmap heap scan
year = 1925	10046	Bitmap heap scan
1930 < year < 1950	208418	Bitmap heap scan
1950 < year <2000	504252	Bitmap heap scan
1900 < year <1990	**900879**	**Sequential scan**
year = 1990	9906	Bitmap heap scan
1990 < year <1995	59437	Bitmap heap scan

From the above table, we can see that there is a direct correlation of *no. of rows being passed* out of the given range condition with the query plan. As the no. of rows increases beyond **90%** or so, the optimizer actually opts for the sequential scanning, instead of doing any sort of index scan or bitmap heap scan. *The reason: since the number of rows are excessively large, a large number of heap pages would have to be loaded after the bitmap generation, and within each of them, most rows would actually pass the condition*. This massive amount of frequent fetch and load operations would make bitmap scan very costly. A far better option would be the basic sequential iteration.

So most of the time, the index on the year column of *movie* table (*movie_year*) is being used for the index scanning purposes. But when normal sequential scanning is being done, a general filter for the range condition is being applied.

5) For **Q5** and **Q6** both, the query selectivities are very small. Hence, the query optimizer decides to opt for Bitmap heap scan again, and in both the cases the index on pc_id (*movie_pcid*) is used for the index scanning.

   This can be understood if we understand how ***Range queries*** are generally executed in PostgreSQL. The pc_id is a *non-key attribute* for the movie table, and if we consider the B+ tree (based on secondary index) where the search key is a non-key attribute like this, an index scan is always beneficial for a small number of indexes accessed. The advantage comes from the fact that the leaf nodes (which contain the pointers to the heap pages having the list of row addresses) are in sorted order of the *movie_pcid index*. So executing a range query involves:
   - Searching for the starting/end point of the query (50 for **Q5** and 20000 for **Q6**) in the B+ tree
   - Level-order traversal of the leaf nodes (to get the indexes matching with the given range)
   - Accessing each of those heap page addresses through the leaf pointers

   For small, number of rows this fetching and accessing is really beneficial, since all the heap page addresses corresponding to the matched indexes are generated in one go (the lower plan: *index scanning* on *movie_pcid* index) and then those addresses/locations are conveniently sorted by the *Bitmap heap scanning* technique, to minimise the fetch and access overhead costs (the upper plan).

   We ran the following range queries on the *prod_company* attribute. Most of them gave Bitmap heap scan as the plan chosen by the query optimiser. But, for each of them we also ran sequential scans (by dropping the indexes first).

Range of ‘prod_coompany’	**No. of rows passed**	**Bitmap heap scan** execution time	**Sequential scan** execution time
< 50	87922	1736.661 ms	7419.124 ms
< 20	34233	21.355 ms	304.180 ms
50 to 10000	**824109**	NA	328.133 ms
10000 to 20000	12365	28.556 ms	191.671 ms
100 to 500	**722511**	NA	351.790 ms
> 20000	75605	180.090 ms	268.386 ms

The above table shows that when query selectivity is huge (70% +) the optimiser chooses to go for a normal sequential scan. For other cases, when the selectivity is substantially low, Bitmap heap scan is chosen. The highlighted pink entries show the query plan actually chosen by the optimiser in each case. As it can be seen when query selectivity is low, using sequential scan would have lead to a *much larger execution time* (intuitively *larger cost*)

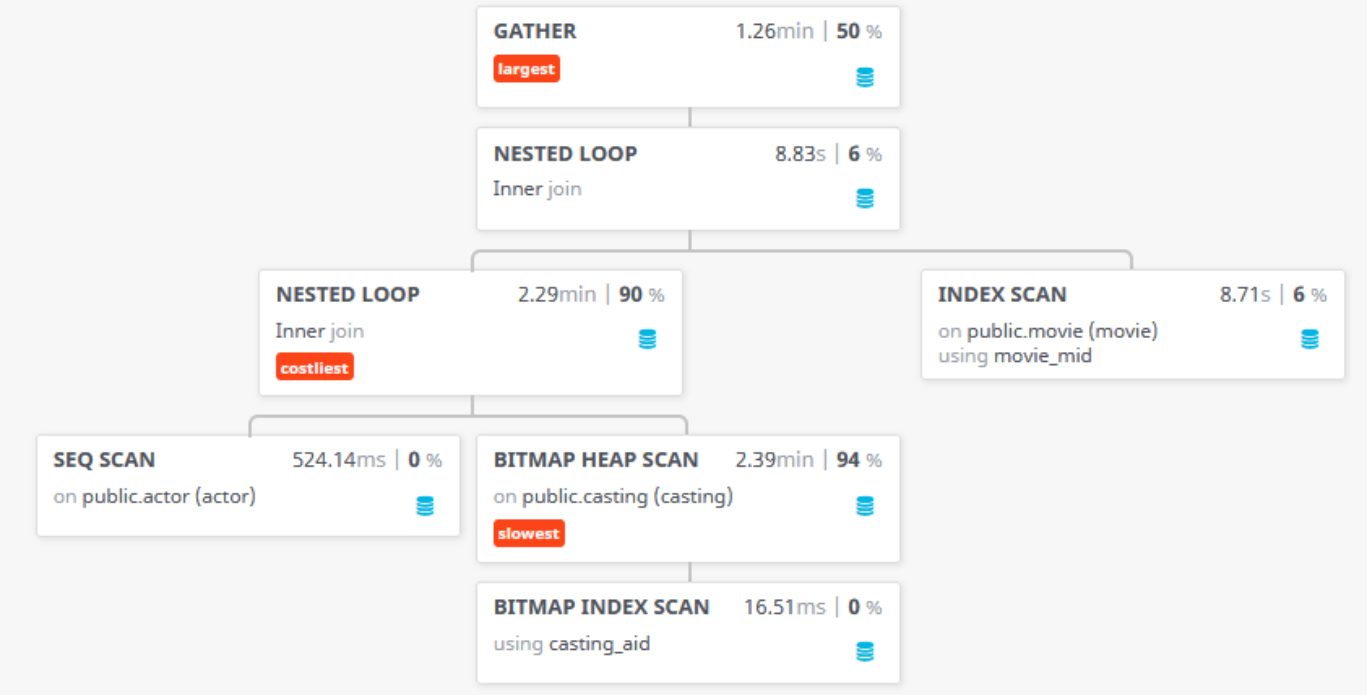
Part B: Join Strategies

1) Queries were run on the database and query plans analyzed.

2) The following are the query plans for each of the given commands and our interpretation of the same:

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

```
                                                          QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------
Gather  (cost=1007.39..29019.44 rows=640 width=27) (actual time=1.114..5847.014 rows=18252 loops=1)
  Workers Planned: 1
  Workers Launched: 1
  ->  Nested Loop  (cost=7.39..27955.44 rows=376 width=27) (actual time=1795.818..4717.324 rows=9126 loops=2)
        ->  Nested Loop  (cost=6.96..27782.50 rows=376 width=20) (actual time=1795.802..3247.745 rows=9126 loops=2)
              ->  Parallel Seq Scan on actor  (cost=0.00..4116.88 rows=28 width=20) (actual time=1795.694..1795.745 rows=25 loops=2)
                    Filter: (a_id < 50)
                    Rows Removed by Filter: 149976
              ->  Bitmap Heap Scan on casting  (cost=6.96..841.93 rows=327 width=8) (actual time=0.111..59.149 rows=372 loops=49)
                    Recheck Cond: (a_id = actor.a_id)
                    Heap Blocks: exact=18076
                    ->  Bitmap Index Scan on casting_aid  (cost=0.00..6.88 rows=327 width=0) (actual time=0.049..0.049 rows=372 loops=49)
                          Index Cond: (a_id = actor.a_id)
        ->  Index Scan using movie_mid on movie  (cost=0.42..0.46 rows=1 width=15) (actual time=0.160..0.160 rows=1 loops=18252)
              Index Cond: (m_id = casting.m_id)
Planning Time: 28.937 ms
Execution Time: 5849.103 ms
```



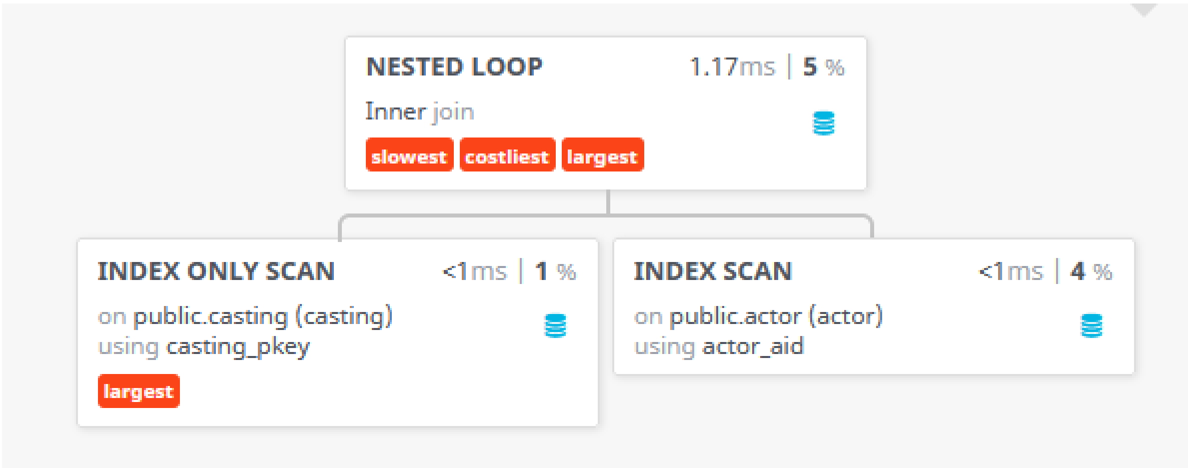
Since the GATHER clause is at the top, this means that the entire query plan is executed in parallel. Since the number of worker threads created is 1, the parallel processing becomes similar to a normal nested loop. The **outermost nested loop** is an INNER JOIN between two other scans/joins:

- Index scanning of movie table (using movie_mid index)
- **Inner Nested loop** between the actor and casting columns.
  - ➔ Sequential scan on actor table
  - ➔ Bitmap heap and bitmap index scan on casting table (using casting_aid index).

For the outer nested loop, the index scan of the **movie** is the INNER relation and the resultant joined output of **actor** and **casting** table is the OUTER relation. This is because the movie table has 10 lakh rows, but the combination of resultant **actor** and **casting** join table is much less than that. The table with fewer rows actually becomes the outer relation (Block cost for nested loop join: *br* + *br***bs*, where br=number of outer loop blocks, bs= number of inner loop blocks) For the inner nested loop, the **casting** table is the INNER relation (number of rows much larger than the actor table) and the **actor** table is the OUTER loop.

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

```
                                                            QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
Nested Loop  (cost=0.85..1524.73 rows=193 width=16) (actual time=102.583..612.196 rows=196 loops=1)
  ->  Index Only Scan using casting_pkey on casting  (cost=0.43..11.81 rows=193 width=4) (actual time=90.741..91.009 rows=196 loops=1)
        Index Cond: (m_id < 50)
        Heap Fetches: 0
  ->  Index Scan using actor_aid on actor  (cost=0.42..7.84 rows=1 width=20) (actual time=2.656..2.656 rows=1 loops=196)
        Index Cond: (a_id = casting.a_id)
Planning Time: 16.108 ms
Execution Time: 612.298 ms
```

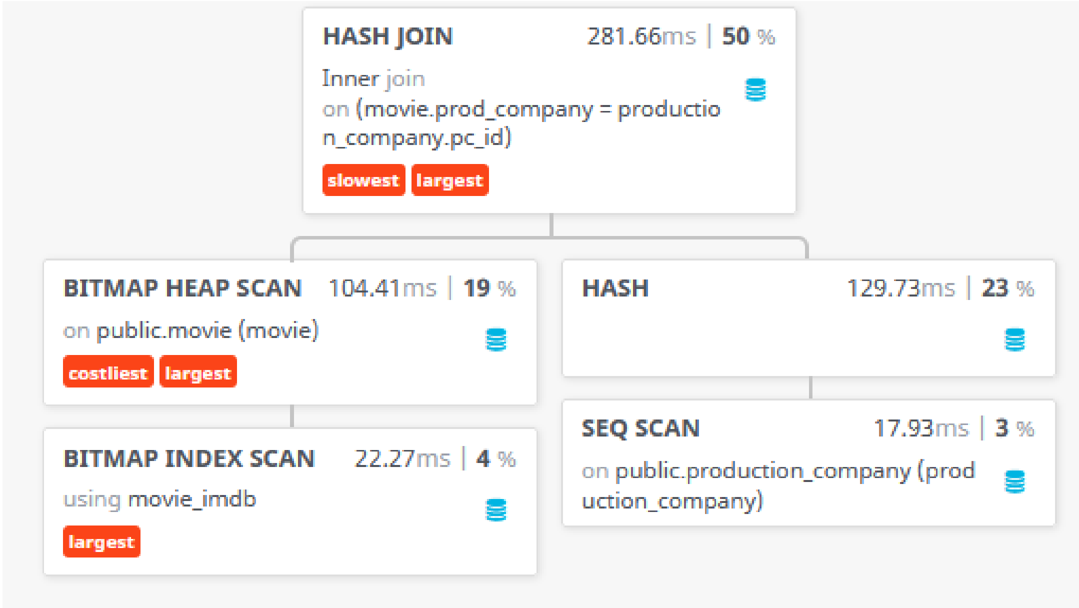


This plan is using a simple NESTED loop, with inner and outer relations. The casting table (largest number of rows) will be the INNER relation and the actor table (lesser number of rows) is the OUTER relation.

The *actor* table is scanned using *Index scan*, whereas the casting table is scanned using *Index only scan*. This is because the projection output of the join only uses the *actor.name* field, and nothing of the *casting* relation. So, the scanning of the *casting* relation doesn't require a fetch to bring the entire row data, and only the comparison of the m_id attribute is used (which is a *key attribute*). But for the *actor* relation, we need the entire row data (especially the name attribute).

3. **SELECT movie.name, production_company.name FROM movie, production_company WHERE movie.prod_company=production_company.pc_id and movie.imdb_score<1.5;**

```
                                                            QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
Hash Join  (cost=5243.96..23401.97 rows=122908 width=22) (actual time=1645.868..12983.140 rows=124111 loops=1)
  Hash Cond: (movie.prod_company = production_company.pc_id)
  ->  Bitmap Heap Scan on movie  (cost=2304.96..18547.31 rows=122908 width=15) (actual time=229.283..11397.818 rows=124111 loops=1)
        Recheck Cond: (imdb_score < 1.5)
        Heap Blocks: exact=7354
        ->  Bitmap Index Scan on movie_imdb  (cost=0.00..2274.24 rows=122908 width=0) (actual time=196.025..196.025 rows=124111 loops=1)
              Index Cond: (imdb_score < 1.5)
  ->  Hash  (cost=1548.00..1548.00 rows=80000 width=15) (actual time=1403.063..1403.064 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=13.977..1291.197 rows=80000 loops=1)
Planning Time: 310.462 ms
Execution Time: 12991.297 ms
```



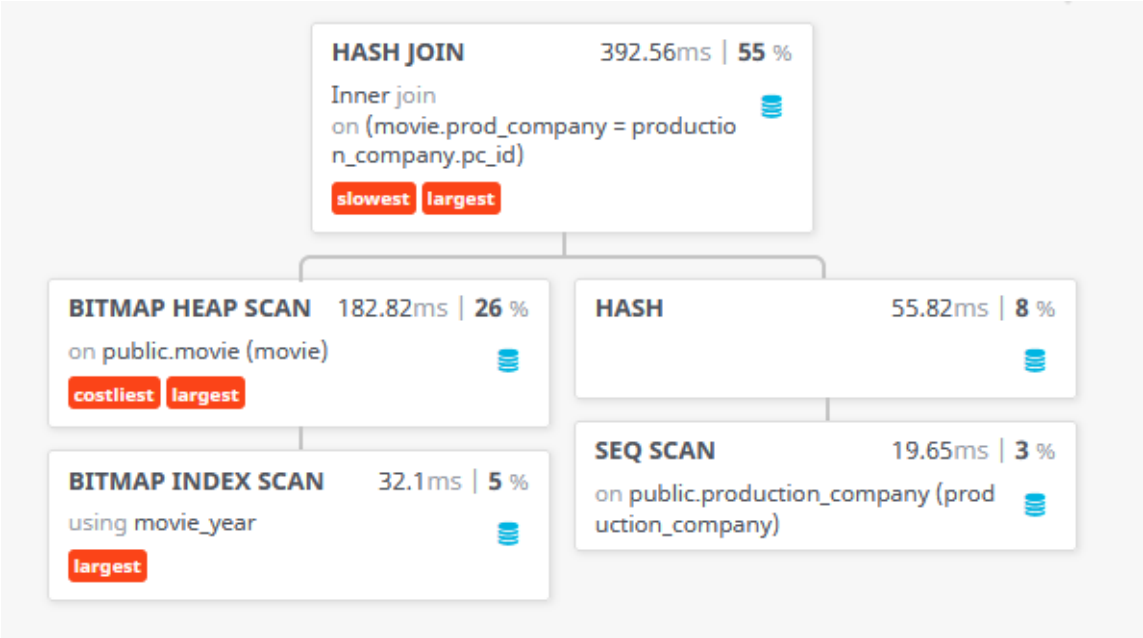
This query uses a **Hash Join** Query plan to optimise the output.

- **Build Phase**: This uses the *production_company* relation, since the number of rows is much less than the *movie* table. First we do a sequential scan on the production_company relation, after which the outputs are hashed using a certain hash function. The hash key is calculated based on the Join clause key, and the hash table is created.

- **Probe Phase**: This uses the movie relation. We are first doing a bitmap index scan (where we are checking condition: *movie.imdb_score<1.5* by using the *movie_imdb* index) followed by the bitmap heap scan, and then the obtained records are hashed based on the join clause key to find entry in the generated hash table (of build phase) The join condition (*movie.prod_company=production_company.pc_id*) is checked in this phase, and then the required output columns are projected.

4. **SELECT movie.name, production_company.name FROM movie, production_company WHERE movie.prod_company=production_company.pc_id and movie.year between 1950 and 2000;**

```
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------
Hash Join  (cost=9920.16..39000.99 rows=510511 width=22) (actual time=316.135..698.507 rows=504252 loops=1)
  Hash Cond: (movie.prod_company = production_company.pc_id)
  ->  Bitmap Heap Scan on movie  (cost=6981.16..29344.83 rows=510511 width=15) (actual time=276.655..392.804 rows=504252 loops=1)
        Recheck Cond: ((year >= 1950) AND (year <= 2000))
        Heap Blocks: exact=7354
        ->  Bitmap Index Scan on movie_year  (cost=0.00..6853.53 rows=510511 width=0) (actual time=275.279..275.279 rows=504252 loops=1)
              Index Cond: ((year >= 1950) AND (year <= 2000))
  ->  Hash  (cost=1548.00..1548.00 rows=80000 width=15) (actual time=39.099..39.099 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=0.033..13.725 rows=80000 loops=1)
Planning Time: 31.808 ms
Execution Time: 715.789 ms
```



This query also uses a **Hash Join** Query plan to optimise the output.

- **Build Phase**: This uses the production_company relation, since the number of rows is much less than the movie table. First we do a sequential scan on the production_company relation, after which the outputs are hashed using a certain hash function. The hash key is calculated based on the Join clause key, and the hash table is created.

- **Probe Phase**: This uses the movie relation. We are first doing a bitmap index scan (where we are checking condition: *movie.year between 1950 and 2000* by using the *movie_year* index) followed by the bitmap heap scan, and then the obtained records are hashed based on the join clause key to find entry in the generated hash table (of build phase) The join condition (*movie.prod_company=production_company.pc_id*) is checked in this phase, and then the required output columns are projected.

3) The following query plans may be opted for by the query optimiser:

- **Nested Loop**:

  This requires a minimum of 3 disk blocks in the main memory for storage of blocks, and neither r nor s can be fit in the main memory.

  Estimated cost (worst case) = $br + br*bs$

  (where br=number of outer loop blocks, bs= number of inner loop blocks)

- **Single Loop join**:

  When we have a primary index of B+ tree on s, and the join attribute is a key attribute of s, and join condition is an equality condition:

Estimated cost (worst case) = *br + |r|* (height + 1) block accesses*

where s= inner relation, r= outer relation,

If we have secondary index on s, which join attribute non-key attribute of s and join condition is equality condition,

Estimated cost= *br + |r| * (height + sb + ceil(sb/BFR sb)) block accesses*

where s= inner relation, r=outer relation, sb= expected no. of records for unique value

- **Hash Join**

If recursive partitioning is not required, then

*Estimated cost = 3 * (br + bs) + 4*n*

Where r=outer relation, s=inner relation, n= no. of partitions created on r and s

For the scanning purposes in the loops, the following may be used. Suppose relation r is being scanned, and it has br number of blocks. Then:

- **Sequential scan:**

Worst case cost estimate= br + 1 seek

If search value is a key-attribute, then average cost = br/2 + 1 seek

- **Index scan**

For primary index, and equality on key-attribute= height + 1

For primary index and equality on non-key attribute= height + b

Where b= number of blocks with matching records

For secondary index, search for key attribute= height +1

For secondary index, search on non-key attribute= height + ceil(sb/ BFR sb) + sb

Where, sb= estimated no. of records matched

- **Bitmap index scan**
- **Bitmap heap scan**

Based on the query plans decided by the query plans, we came up with the following interpretations regarding the choices:

Query	Query plan adopted	Resultant no. of rows	Reason
Q1	Gather (with nested loop)	18252	Intermediate number of rows, hence nested loop but some scope for parallel running of plan nodes
Q2	Nested loop	196	Least number of resultant rows, hence normal nested loop
Q3	Hash Join	124111	Large number of rows, hence hash join
Q4	Hash Join	504252	Large number of rows, hence hash join

For high query selectivity queries (large number of resultant rows after joining), hash join was being preferred since all rows need not to be scanned and matched. But for smaller query selectivity queries, nested loop was favourable since the hash join must create a hash table first, and this adds to the cost overhead. This can be explained as follows: when we use a nested loop, then the cost estimate will increase with increase in size of the inner and outer relations of the loop (the *br + br*bs* expression). This plan can be chosen if small subsets of the data are being joined, or if one of the relations has much fewer rows than the other relation (the smaller relation will be used as outer loop). In **Q1** and **Q2**, the actor table is much smaller than the casting table, and so nested loops are preferred.

But in the case of **Q3** and **Q4**, the *production_company* and *movie* table are comparable in size and the number of resultant rows is very large, then the hash join becomes more favourable. A nested loop would have matched every tuple of the outer loop with the inner loop, and this would have resulted in a huge number of positive matches. But using the hash join algorithm, we first do a sequential scan of the comparatively smaller relation and a hash table is made. Each row is inserted into a hash bucket based on the hash key. In the next step, instead of comparing all the tuples of the other relation, it is compared with each bucket and that is how the suitable matches are found for the join (It has to be noted that the sequential scan of the relation, building the hash table and then assigning buckets to each row also contributes to a significant cost. Hence if the number of rows being compared are too small, a simple nested loop might be favoured instead of the hash join)

How does order of join affect the overall cost ?

In **Q1** there is a significant importance of ordering of the joins since we are joining 3 tables in total. Two nested loops are getting executed, and how the inner and outer loops are arranged actually plays a significant role in the cost. The overall join has to be between *movie*, *casting* and *actor*. Either we can do the *actor+casting* join first, then join with *movie* relation. Or, we can join *movie+casting* first and then join with the *actor* relation. The *movie* relation has 10 lakh rows, *actor* relation has 3 lakh rows and *casting* relation has 40 lakh rows. If we need to join *casting* with any of the two, joining with *actor* seems to be a better option. Explanation: looking at the **br + br*bs expression**, the number of blocks (subsequently rows) in the outer loop matters the most for the overall cost. If we choose *actor* as the outer loop, then this value is much smaller than when *movie* is chosen as outer loop. Therefore, we first join *actor+casting* and then join the resultant with the *movie* relation.

Further for **Q1**, the next analysis comes in case of scan strategy being used for the loop iterations. The *actor* table is the smallest, hence sequential scan is being preferred. For scanning the *casting* table, we are using bitmap heap scan. But when we are scanning *movie* relations, we can directly use the index scan since the search attribute of the join (*m_id*) has an index, and it is a key attribute. The same interpretation goes for the other queries **Q2**-**Q4**.

Importance of hash join and buckets

In **Q3** and **Q4,** let's see how hash join has significantly reduced the total cost. Let's say there are 'x' number of buckets formed after utilising the inbuilt hash function on the *build* relation. Also, let's consider there are 'y' number of records in the *probe* relation. Also, let's assume on an average there are 'y/x' number of entries per bucket matching with the *build* relation. So when we run the hash join function, in the worst case we would have to compare the 'y/x' amount of entries for each tuple in the *probe* relation. This significantly reduces the cost as in the normal nested join we would have had to run them for 'm' (no of rows in build relation) times per record in *probe* relation (y*m >> (y*y)/x in general).

***Reference:** The charts for the query plan visualization has been created using the following open-source website : Postgres EXPLAIN visualizer (pev)*