Database System Implementation (CSE507) : Homework 3

Anshuman Suri : 2014021
Rounaq Jhunjhunu Wala : 2014089

Query Selectivity (part A)

A2.

```
SELECT name FROM movie WHERE imdb score < 2;
```

1. [('Seq Scan on movie (cost=0.00..20834.00 rows=941006 width=11) (actual time=0.005..189.715 rows=950000 loops=1)',), ('Filter: ("imdb score" < \'2\'::numeric)',), ('Rows Removed by Filter: 50000',), ('Planning time: 0.234 ms',), ('Execution time: 216.959 ms',)]

   Selected : 95% records *(obtained via COUNT(*))*

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

2. [('Seq Scan on movie (cost=0.00..23334.00 rows=512094 width=11) (actual time=0.017..195.460 rows=516167 loops=1)',), ('Filter: (("imdb score" >= 1.5) AND ("imdb score" <= 4.5))',), ('Rows Removed by Filter: 483833',), ('Planning time: 0.216 ms',), ('Execution time: 209.559 ms',)]

   Selected : 51.61617% records *(obtained via COUNT(*))*

```
SELECT name FROM movie WHERE year between 1900 and 1990;
```

3. [('Seq Scan on movie (cost=0.00..23334.00 rows=888663 width=11) (actual time=0.013..147.045 rows=900921 loops=1)',), ('Filter: ((year >= 1900) AND (year <= 1990))',), ('Rows Removed by Filter: 99079',), ('Planning time: 0.326 ms',), ('Execution time: 173.301 ms',)]

   Selected : 90.0921% records *(obtained via COUNT(*))*

```
SELECT name FROM movie WHERE year between 1990 and 1995;
```

4. [('Bitmap Heap Scan on movie (cost=1228.86..10429.85 rows=57799 width=11) (actual time=10.520..27.385 rows=59581 loops=1)',), ('Recheck Cond: ((year >= 1990) AND (year <= 1995))',), ('Heap Blocks: exact=8331',), ('-> Bitmap Index Scan on movie_year_idx (cost=0.00..1214.41 rows=57799 width=0) (actual time=9.209..9.209 rows=59581 loops=1)',), ('Index Cond: ((year >= 1990) AND (year <= 1995))',), ('Planning time: 0.124 ms',), ('Execution time: 29.419 ms',)]

Selected : 5.9581% records *(obtained via COUNT(*))*

A3. We observe that *secondary indices* like the B-Tree index on imdb-score column store only pointers to the records, not the records themselves. Hence, each access of the record can be a full cycle of seek-and-transfer on the disk. This cost added with the looping hierarchical cost of the tree **outweighs** the benefit we gain, as the selectivity is very high. Thus the index is not used.

A4. The reason for query 3 is same as in **A2**. For query 4, however, the index is used because the selection cardinality is very low, hence doing a range selection on the index and fetching records is better than scanning the whole table, as most part of it would be rejected.

Aggregate Operations (part B)

```
SELECT name FROM movie WHERE imdb score <= (SELECT MIN (imdb score)
FROM movie WHERE year between 1990 and 1995);
```

A. [('Bitmap Heap Scan on movie (cost=7725.42..20226.09 rows=333333 width=11) (actual time=0.097..0.104 rows=10 loops=1)',), ('Recheck Cond: ("imdb score" <= $1)',), ('Heap Blocks: exact=10',), (' InitPlan 2 (returns $1)',), ('-> Result (cost=1.66..1.67 rows=1 width=0) (actual time=0.081..0.081 rows=1 loops=1)',), ('InitPlan 1 (returns $0)',), ('-> Limit (cost=0.42..1.66 rows=1 width=12) (actual time=0.078..0.078 rows=1 loops=1)',), ('-> Index Scan using "movie_imdb score_idx" on movie movie_1 (cost=0.42..71248.23 rows=57799 width=12) (actual time=0.077..0.077 rows=1 loops=1)',), ('Index Cond: ("imdb score" IS NOT NULL)',), ('Filter: ((year >= 1990) AND (year <= 1995))',), ('Rows Removed by Filter: 9',), ('-> Bitmap Index Scan on "movie_imdb score_idx" (cost=0.00..7640.42 rows=333333 width=0) (actual time=0.092..0.092 rows=10 loops=1)',), ('Index Cond: ("imdb score" <= $1)',), ('Planning time: 0.518 ms',), ('Execution time: 0.136 ms',)]

```
SELECT name FROM movie WHERE imdb score <= ALL (SELECT imdb score FROM
movie);
```

B. [('Seq Scan on movie (cost=0.00..15358520834.00 rows=500000 width=11) (actual time=2542.708..4352.755 rows=1 loops=1)',), ('Filter: (SubPlan 1)',), (' Rows Removed by Filter: 999999',), ('SubPlan 1',), ('-> Materialize (cost=0.00..28217.00 rows=1000000 width=12) (actual time=0.000..0.002 rows=22 loops=1000000)',), ('-> Seq Scan on movie movie_1 (cost=0.00..18334.00 rows=1000000 width=12) (actual time=0.007..98.600 rows=1000000 loops=1)',), ('Planning time: 0.181 ms',), ('Execution time: 4355.726 ms',)]

B3. We can see that query 1 uses a Bitmap Heap Scan at the top level, and use the individual indices to create the heap bitmaps.
In the first query, the index on year column is not used because the selectivity of the condition (year between 1990 and 1995) is very high and from the reason stated in part A, high selectivity causes the DBMS to not use indexes. So, there is a sequential scan on the table (actually an index scan but the condition *imdb_score NOT NULL* is true for all the tuples in our case). For evaluating the outer WHERE condition, we have as expected used the index to create the heap bitmap because of the key-order B-Tree provides and the low selectivity.
In the second query, the inner query result is **materialized**, because , the result is used for all the tuples of the outer query, and is too big to calculate again (actually, the result set is an existing table, but we think the DBMS was conservative on that decision). Since we simply want the WHERE clause to be evaluated with every tuple, a sequential scan on the outer level is the only possible plan to execute.

Join strategies (part C)

```
SELECT a.name, b.name FROM Actor a INNER JOIN Casting c ON (a.a_id =
c.a_id) INNER JOIN Movie b  ON (b.movie_id = c.m_id) where a.a_id < 50
```

A. [('Nested Loop (cost=1.27..723.25 rows=989 width=27) (actual time=0.035..4.650 rows=951 loops=1)',), (' -> Nested Loop (cost=0.85..259.67 rows=1000 width=20) (actual time=0.024..0.283 rows=951 loops=1)',), (' -> Index Scan using actor_pkey on actor a (cost=0.42..9.29 rows=50 width=20) (actual time=0.003..0.012 rows=49 loops=1)',), (' Index Cond: (a_id < 50)',), (' -> Index Only Scan using casting_pkey on casting c (cost=0.43..4.80 rows=21 width=8) (actual time=0.002..0.004 rows=19 loops=49)',), (' Index Cond: (a_id = a.a_id)',), (' Heap Fetches: 0',), (' -> Index Scan using movie_pkey on movie b (cost=0.42..0.45 rows=1 width=15) (actual time=0.004..0.004 rows=1 loops=951)',), (' Index Cond: (movie_id = c.m_id)',), ('Planning time: 0.728 ms',), ('Execution time: 4.720 ms',)]

Nested loop join is generally used whenever it is feasible to **keep one of the relations in the main memory**. In the above case, the following optimizations happen:

- The selection condition (a_id < 50) is pushed from top level to the table level, since the columns concerned belong to just one table. Also, the index is used to evaluate the same as the selectivity is low.
- Then we keep the above result in main memory and proceed to nested loop join with Casting table, which yields 19 rows. We use an index only scan on the casting pkey as the pkey essentially has the whole tuple, so we don't need to go to the table.
- Then we proceed to nested loop join of this result with Movie table. The index on movie id is used to make the process fast.

```
SELECT a.name, b.name FROM Actor a INNER JOIN Casting c ON (a.a_id =
c.a_id) INNER JOIN Movie b  ON (b.movie_id = c.m_id) where c.m_id < 100
```

B. [('Nested Loop (cost=1.27..6232.83 rows=415 width=27) (actual time=0.023..2.207 rows=396 loops=1)',), (' -> Nested Loop

```
(cost=0.85..3414.83 rows=415 width=15) (actual time=0.010..0.607
rows=396 loops=1)',), ('        ->  Index Scan using casting_m_id_idx
on casting c  (cost=0.43..20.78 rows=420 width=8) (actual
time=0.007..0.069 rows=396 loops=1)',), ('              Index Cond:
(m_id < 100)',), ('        ->  Index Scan using movie_pkey on movie b
(cost=0.42..8.07 rows=1 width=15) (actual time=0.001..0.001 rows=1
loops=396)',), ('              Index Cond: (movie_id = c.m_id)',), ('
->  Index Scan using actor_pkey on actor a  (cost=0.42..6.78 rows=1
width=20) (actual time=0.004..0.004 rows=1 loops=396)',), ('
Index Cond: (a_id = c.a_id)',), ('Planning time: 0.272 ms',),
('Execution time: 2.240 ms',)]
```

Nested loop join is normally used whenever it is feasible to **keep one of the relations in the main memory**. In the above case, the following optimizations happen:

- The selection condition (m_id < 100) is pushed from top level to the table level, since the columns concerned belong to just one table. Also, the index is used to evaluate the same as the selectivity is low.
- Then we next-join the above result with Movie table. An index scan is used to match the m_id column.
- Then we keep the above result in main memory and proceed to nested loop join with Actor table. It uses the actor a_id index to query the records directly. This yields 1 row.

SELECT a.name, b.name FROM Actor a INNER JOIN Casting c ON (a.a_id = c.a_id) INNER JOIN Movie b ON (b.movie_id = c.m_id) where b.year between 1990 and 2000

C.
```
[('Hash Join  (cost=21251.40..141673.07 rows=440239 width=27) (actual
time=104.241..1138.135 rows=435324 loops=1)',), ('  Hash Cond: (c.a_id
= a.a_id)',), ('  ->  Hash Join  (cost=14305.40..123201.79 rows=440239
width=15) (actual time=56.316..907.407 rows=435324 loops=1)',), ('
Hash Cond: (c.m_id = b.movie_id)',), ('        ->  Seq Scan on casting
c  (cost=0.00..57700.00 rows=4000000 width=8) (actual
time=0.002..244.007 rows=4000000 loops=1)',), ('        ->  Hash
(cost=12369.68..12369.68 rows=111337 width=15) (actual
time=56.020..56.020 rows=108831 loops=1)',), ('              Buckets:
131072  Batches: 2  Memory Usage: 3587kB',), ('              ->  Bitmap
Heap Scan on movie b  (cost=2365.63..12369.68 rows=111337 width=15)
(actual time=8.636..36.220 rows=108831 loops=1)',), ('
Recheck Cond: ((year >= 1990) AND (year <= 2000))',), ('
Heap Blocks: exact=8334',), ('                    ->  Bitmap Index Scan
on movie_year_idx  (cost=0.00..2337.80 rows=111337 width=0) (actual
time=7.597..7.597 rows=108831 loops=1)',), ('
Index Cond: ((year >= 1990) AND (year <= 2000))',), ('  ->  Hash
(cost=3274.00..3274.00 rows=200000 width=20) (actual
```

```
time=47.746..47.746 rows=200000 loops=1)',), ('        Buckets: 65536
Batches: 4  Memory Usage: 3066kB',), ('        ->  Seq Scan on actor a
(cost=0.00..3274.00 rows=200000 width=20) (actual time=0.002..18.539
rows=200000 loops=1)',), ('Planning time: 0.449 ms',), ('Execution
time: 1151.140 ms',)]
```

Since neither of the 2 relations in both the INNER JOINs are small, a **hash join** is used instead of a nested loop join.

- The selection condition (year between 1990 and 2000) is pushed from top level to the table level, since the columns concerned belong to just one table. To solve this query, the DBMS employs a similar plan we saw in B.Qry1 using Bitmap Heap Scan.
- To hash join, we do sequential scan on the Casting table and hash it. We also hash the result from above. And we join them both.
- We do the same thing as in point 2 with the Actor table and the result set.

```
SELECT a.name, b.name FROM Movie a INNER JOIN ProductionCompany b ON
(a.\"production company\" = b.pc_id) WHERE b.pc_id < 50
```

D.
```
[('Hash Join  (cost=9.70..22103.10 rows=940 width=22) (actual
time=0.023..245.799 rows=900075 loops=1)',), ('  Hash Cond:
(a."production company" = b.pc_id)',), ('  ->  Seq Scan on movie a
(cost=0.00..18334.00 rows=1000000 width=15) (actual time=0.002..53.538
rows=1000000 loops=1)',), ('  ->  Hash  (cost=9.11..9.11 rows=47
width=15) (actual time=0.017..0.017 rows=49 loops=1)',), ('
Buckets: 1024  Batches: 1  Memory Usage: 11kB',), ('        ->  Index
Scan using "Production Company_pkey" on productioncompany b
(cost=0.29..9.11 rows=47 width=15) (actual time=0.002..0.010 rows=49
loops=1)',), ('              Index Cond: (pc_id < 50)',), ('Planning
time: 0.176 ms',), ('Execution time: 271.719 ms',)]
```

Since the result sets are smaller, we can use nested loop join. But, nested loop join has more overhead than hash join, and hence it is used in smaller tables only when it is beneficial in some other way, for example, in Q1 and Q2, both could benefit from pipelining of result set for higher join, but this is not possible when we use hash join.

- The selection condition (pc_id < 50) is pushed from top level to the table level, since the columns concerned belong to just one table. To solve this query, index is used because of the small selectivity.
- To hash join, we do sequential scan on the Movie table and hash it. We also hash the result from above. And we join them both.

```
SELECT a.name, b.name FROM Movie a INNER JOIN ProductionCompany b ON
(a.\"production company\" = b.pc_id) WHERE a."imdb score" < 1.5
```

E.
```
[('Hash Join  (cost=1593.00..29017.02 rows=479274 width=22) (actual
time=11.656..269.091 rows=475570 loops=1)',), ('  Hash Cond:
```

```
(a."production company" = b.pc_id)',), ('  ->  Seq Scan on movie a
(cost=0.00..20834.00 rows=479274 width=15) (actual time=0.003..161.602
rows=475570 loops=1)',), ('        Filter: ("imdb score" < 1.5)',), ('
Rows Removed by Filter: 524430',), ('  ->  Hash  (cost=968.00..968.00
rows=50000 width=15) (actual time=11.541..11.541 rows=50000
loops=1)',), ('        Buckets: 65536  Batches: 1  Memory Usage:
2856kB',), ('        ->  Seq Scan on productioncompany b
(cost=0.00..968.00 rows=50000 width=15) (actual time=0.001..4.490
rows=50000 loops=1)',), ('Planning time: 0.150 ms',), ('Execution time:
282.844 ms',)]
```

Since neither of the 2 relations in both the INNER JOINs are small, a **hash join** is used instead of a nested loop join.

- The selection condition (imdb score < 1.5) is pushed from top level to the table level, since the columns concerned belong to just one table. Due to the high selectivity, A sequential scan is done over Movie.
- To hash join, we do sequential scan on the Production Company table and hash it. We also hash the result from above. And we join them both.

```
SELECT a.name, b.name FROM Movie a INNER JOIN ProductionCompany b ON
(a.\"production company\" = b.pc_id) WHERE a.year BETWEEN 1950 and 2000
F. [('Hash Join  (cost=1593.00..31788.37 rows=499009 width=22) (actual
time=11.488..229.952 rows=504033 loops=1)',), ('  Hash Cond:
(a."production company" = b.pc_id)',), ('  ->  Seq Scan on movie a
(cost=0.00..23334.00 rows=499009 width=15) (actual time=0.003..117.857
rows=504033 loops=1)',), ('        Filter: ((year >= 1950) AND (year <=
2000))',), ('        Rows Removed by Filter: 495967',), ('  ->  Hash
(cost=968.00..968.00 rows=50000 width=15) (actual time=11.461..11.461
rows=50000 loops=1)',), ('        Buckets: 65536  Batches: 1  Memory
Usage: 2856kB',), ('        ->  Seq Scan on productioncompany b
(cost=0.00..968.00 rows=50000 width=15) (actual time=0.001..4.664
rows=50000 loops=1)',), ('Planning time: 0.145 ms',), ('Execution time:
244.803 ms',)]
```

Since neither of the 2 relations in both the INNER JOINs are small, a **hash join** is used instead of a nested loop join.

- The selection condition (year between 1950 and 2000) is pushed from top level to the table level, since the columns concerned belong to just one table. Due to the high selectivity, A sequential scan is done over Movie.
- To hash join, we do sequential scan on the Production Company table and hash it. We also hash the result from above. And we join them both.