CS 387 LAB 8 - Pandurang(200050096) and Shikhar(200070076)

1. EXPLAIN select * from takes where id='12345';

```
                          QUERY PLAN
------------------------------------------------------------------------
 Bitmap Heap Scan on takes  (cost=4.40..52.84 rows=15 width=24)
   Recheck Cond: ((id)::text = '12345'::text)
   ->  Bitmap Index Scan on takes_pkey  (cost=0.00..4.40 rows=15 width=0)
         Index Cond: ((id)::text = '12345'::text)
(4 rows)
```

Here Bitmap Index scan is used to create a bitmap over all the blocks(pages) of the relation takes which contain records(tuples) in which id is '12345'. These blocks are the only ones which are fetched for later processing. This is something intermediate between sequential scan and index scan. Bitmap Index scan feeds this data to a parent Bitmap Heap Scan, which can decode the bitmap to fetch the underlying data, grabbing data page by page.

2. EXPLAIN select * from student where id='12345' and tot_cred<30;

```
                          QUERY PLAN
------------------------------------------------------------------------
 Index Scan using student_pkey on student  (cost=0.28..8.30 rows=1 width=24)
   Index Cond: ((id)::text = '12345'::text)
   Filter: (tot_cred < '30'::numeric)
(3 rows)
```

Here, only those records which have id as '12345' need to be fetched. Hence, a simple index scan of the index data structure will reveal the record since id is the primary key of the student relation. This query was chosen as it will result in at most one tuple. Now the fetched record is tested if tot_cred<30.

3. EXPLAIN select * from takes where id = '123' or course_id = '123';

First run these create index commands, and then run the above query.

- create index pandu_id on takes(id);
- create index pandu_cid on takes(course_id);

```
                          QUERY PLAN
------------------------------------------------------------------------
-----
 Bitmap Heap Scan on takes  (cost=8.70..57.20 rows=15 width=24)
   Recheck Cond: (((id)::text = '123'::text) OR ((course_id)::text =
'123'::text))
   ->  BitmapOr  (cost=8.70..8.70 rows=15 width=0)
         ->  Bitmap Index Scan on pandu_id  (cost=0.00..4.40 rows=15 width=0)
```

```
               Index Cond: ((id)::text = '123'::text)
         ->  Bitmap Index Scan on pandu_cid  (cost=0.00..4.29 rows=1 width=0)
               Index Cond: ((course_id)::text = '123'::text)
(7 rows)
```

Here, we created two indices, one on course_id and the other on id. The query plan was expected to create two bitmaps, one for each of the indices. The bitmap OR of these two maps results in the logical OR operation. At the end we get all those blocks of the 'takes' relation which are satisfied by the given condition. These are then fetched. Hence we have written a query with index scans on both the predicates.

4. EXPLAIN select * from student join takes on student.id=takes.id where student.id='12345' and takes.year='2008';

```
                                QUERY PLAN
-------------------------------------------------------------------------------
-----
 Nested Loop  (cost=0.56..16.75 rows=1 width=48)
   ->  Index Scan using student_pkey on student  (cost=0.28..8.29 rows=1
width=24)
         Index Cond: ((id)::text = '12345'::text)
   ->  Index Scan using takes_pkey on takes  (cost=0.29..8.45 rows=1 width=24)
         Index Cond: (((id)::text = '12345'::text) AND (year =
'2008'::numeric))
(5 rows)
```

This query is motivated by using conditions on each of the relations. The query plan first scans all records of takes where id is '12345' and year is '2008'. This can be done using an index scan using index `takes_pkey` on takes. Similar index scans happen on student relation as well. The where condition along with the join on condition enables the use of index scan in Nested loop join.

5. Time output of creating and dropping index

```
lab4db=# \timing
Timing is on.
lab4db=# create index i1 on takes(id, semester, year);
CREATE INDEX
Time: 60.727 ms
lab4db=# drop index i1;
DROP INDEX
Time: 6.879 ms
```

CS 387 LAB 8 - Pandurang(200050096) and Shikhar(200070076)

6. EXPLAIN ANALYSE insert into takes2 select * from takes;

```
`                                QUERY PLAN
-------------------------------------------------------------------------------
---------------------------------
 Insert on takes2  (cost=0.00..521.03 rows=0 width=0) (actual
time=25.100..25.100 rows=0 loops=1)
   ->  Seq Scan on takes  (cost=0.00..521.03 rows=30003 width=24) (actual
time=0.007..3.466 rows=30003 loops=1)
 Planning Time: 0.046 ms
 Execution Time: 25.111 ms
(4 rows)
```

7. Adding primary key to takes2

```
ALTER TABLE takes2 ADD CONSTRAINT takes2_pkey primary
key (ID, course_id, sec_id, semester, year);
ALTER TABLE
Time: 55.848 ms
```

8. EXPLAIN ANALYSE insert into takes2 select * from takes;

```
                                QUERY PLAN
-------------------------------------------------------------------------------
-----------------------------
 Insert on takes2  (cost=0.00..521.03 rows=0 width=0) (actual
time=143.294..143.294 rows=0 loops=1)
   ->  Seq Scan on takes  (cost=0.00..521.03 rows=30003 width=24) (actual
time=0.007..3.249 rows=30003 loops=1)
 Planning Time: 0.058 ms
 Execution Time: 143.306 ms
(4 rows)
```

Time required in above is more than sum of time in Q6,7 as when we are adding entries in Q8, we check through all entries for unique constraint and for each new entry we have to iterate over whole table once accounting to overall O(n^2) time complexity while in Q7 when we add Primary key constraint at the end and constraint violation can be checked within O(n*log(n)) time complexity hence resulting in a small time of overall execution.

CS 387 LAB 8 - Pandurang(200050096) and Shikhar(200070076)

9. EXPLAIN select id from student natural join takes order by id;

```
                                QUERY PLAN
-------------------------------------------------------------------------------
-------------
 Merge Join  (cost=0.56..1636.65 rows=30003 width=5)
   Merge Cond: ((student.id)::text = (takes.id)::text)
   ->  Index Only Scan using student_pkey on student  (cost=0.28..70.28
rows=2000 width=5)
   ->  Index Only Scan using takes_pkey on takes  (cost=0.29..1186.33
rows=30003 width=5)
(4 rows)
```

Here the join is base on id and also ordered by id hence we have sorted relations, Merge join is used here as projections of the joined tables are sorted on the join columns. Merge joins are faster and uses less memory than hash joins hence Merge Join is Preferrable.

10.

- PART(a) explain select id from student natural join takes order by id LIMIT 10;

```
                                QUERY PLAN
-------------------------------------------------------------------------------
-------------------
 Limit  (cost=0.56..1.11 rows=10 width=5)
   ->  Merge Join  (cost=0.56..1636.65 rows=30003 width=5)
         Merge Cond: ((student.id)::text = (takes.id)::text)
         ->  Index Only Scan using student_pkey on student  (cost=0.28..70.28
rows=2000 width=5)
         ->  Index Only Scan using takes_pkey on takes  (cost=0.29..1186.33
rows=30003 width=5)
(5 rows)
```

- PART(b.1) explain select id from instructor natural join teaches order by id;

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Sort  (cost=7.73..7.98 rows=100 width=5)
   Sort Key: instructor.id
   ->  Hash Join  (cost=2.12..4.41 rows=100 width=5)
         Hash Cond: ((teaches.id)::text = (instructor.id)::text)
```

```
         ->  Seq Scan on teaches  (cost=0.00..2.00 rows=100 width=5)
         ->  Hash  (cost=1.50..1.50 rows=50 width=5)
                     ->  Seq Scan on instructor  (cost=0.00..1.50 rows=50
width=5)
(7 rows)
Time: 2.072 ms
```

- PART(b.2) explain select id from instructor natural join teaches order by id limit 10;

```
                                                                      QUERY
PLAN
------------------------------------------------------------------------------
---------------------------
 Limit  (cost=0.29..2.75 rows=10 width=5)
   ->  Nested Loop  (cost=0.29..24.82 rows=100 width=5)
         ->  Index Only Scan using teaches_pkey on teaches  (cost=0.14..13.64
rows=100 width=5)
         ->  Memoize  (cost=0.15..0.29 rows=1 width=5)
               Cache Key: teaches.id
               Cache Mode: logical
               ->  Index Only Scan using instructor_pkey on instructor
(cost=0.14..0.28 rows=1 width=5)
                     Index Cond: (id = (teaches.id)::text)
(8 rows)
Time: 0.485 ms
```

Since in the second query the number of rows in the output are limited to be at most 10, postgresql instead uses Nested loop join to compute the output. Hence, using Nested Loop instead of Hash Join isn't bad.

11. EXPLAIN select dept_name, sum(credits) from course group by dept_name;

```
                        QUERY PLAN
------------------------------------------------------------------
 HashAggregate  (cost=5.00..5.25 rows=20 width=41)
   Group Key: dept_name
   ->  Seq Scan on course  (cost=0.00..4.00 rows=200 width=14)
(3 rows)
```

We have used the sum aggregate function on credits. This creates a hashmap in memory and performs the aggregation.

12. EXPLAIN select dept_name, avg(salary) from instructor where name='Sudarshan' group by dept_name;

```
                          QUERY PLAN
------------------------------------------------------------------------
 GroupAggregate  (cost=1.64..1.66 rows=1 width=42)
   Group Key: dept_name
   ->  Sort  (cost=1.64..1.64 rows=1 width=19)
         Sort Key: dept_name
         ->  Seq Scan on instructor  (cost=0.00..1.63 rows=1 width=19)
               Filter: ((name)::text = 'Sudarshan'::text)
(6 rows)
```

Postgresql uses sort inside an aggregate function if the number of rows to be sorted are small. In this case, it expects that a particular name would appear in a few rows and hence sorts them first.