DBSI Assignment 2:
Deepak Kumar Sood(MT15013), Manisha Sharma(MT15031), Neha Rani Deswal(MT15040).

Question 1:

Part A: Range Query and Index.

Write SQL statements for the following VOTES based range queries.

Q1: List the movies with votes between 10 and 100.

```
SELECT title FROM movie WHERE votes BETWEEN 10 AND 100;
```

Q2: List the movies with votes between 20 and 59000.

```
SELECT title FROM movie WHERE votes BETWEEN 20 AND 59000;
```

Submit the explain plan output of above two queries, explain these plans and compare them.

Q1 explain analyse select title from movie where votes between 10 and 100;

```
"Bitmap Heap Scan on movie (cost=4.46..49.51 rows=17 width=17) (actual time=0.061..0.143 rows=15
loops=1)"

"  Recheck Cond: ((votes >= 10) AND (votes <= 100))"

"  Heap Blocks: exact=14"

"  ->  Bitmap Index Scan on movie_votes  (cost=0.00..4.46 rows=17 width=0) (actual time=0.039..0.039
rows=15 loops=1)"

"        Index Cond: ((votes >= 10) AND (votes <= 100))"

"Planning time: 0.540 ms"

"Execution time: 0.242 ms"
```

Here the index movie_votes is used to filter out which blocks satisfy the index condition. Bitmap Index scan is used for that.

For recheck condition, bitmap heap scan is used to find out attribute of our interest. It will return the movie tilte of movies which satisfies the inner condition (Index scan).

Q2 explain analyse select title from movie where votes between 20 and 59000;

```
"Seq Scan on movie  (cost=0.00..248.48 rows=9711 width=17) (actual time=0.012..5.658 rows=9707
loops=1)"

"  Filter: ((votes >= 20) AND (votes <= 59000))"

"  Rows Removed by Filter: 192"

"Planning time: 0.258 ms"

"Execution time: 6.284 ms"
```

Sequential scan is used to find out the movies titles which satisfies the condition on votes attribute.Since we need to retrieve the tilte of the movies and the condition is on votes attribute, so instead of accessing the *index table first and then accessing the file blocks, we can directly access the file blocks because the range of the query contains many recoreds of file and index table only is not sufficient to answer the query.*

Comparision:-

For Q1, the execution time is very less as compared to Q2 because we use indexes in Q1 whereas sequntial scan is done in Q2 and also rows returned by Q2 are more than Q1. However planning time in Q2 is less than Q1.

5. Explain why the index on VOTES is not always used for range queries based on VOTES attribute.

Index on votes is not always helpful when we need to retrieve the values for the attribute other than votes and we need to retrieve large number of blocks.

So, the index on votes will not be useful to answer the queries where the range is high (the answer of the query contains almost all the records, only few are removed by the filter) and we are retrieving values for some other attribute because in this case instead of going to the index and then going to the file, we can directly access the file.

Example of this type of queries is Q2.

Part B: Selectivity

Q1: SELECT title FROM movie WHERE votes < 10000;

Q2: SELECT title FROM movie WHERE votes between 60000 and 100000;

Q3: SELECT title FROM movie WHERE yr between 1900 and 1990;

Q4: SELECT title FROM movie WHERE yr between 1890 and 1900;

Run the explain plan for Q1, Q2, Q3 and Q4 and Submit the output.

Q1

```
 "Bitmap Heap Scan on movie (cost=40.99..161.47 rows=1639 width=17) (actual time=1.234..3.386 rows=1632 loops=1)"

"  Recheck Cond: (votes < 10000)"

"  Heap Blocks: exact=100"

"  ->  Bitmap Index Scan on movie_votes  (cost=0.00..40.58 rows=1639 width=0) (actual time=1.141..1.141 rows=1632 loops=1)"

"        Index Cond: (votes < 10000)"

"Planning time: 0.504 ms"

"Execution time: 3.705 ms"
```

Q2

```
"Bitmap Heap Scan on movie (cost=5.26..109.47 rows=95 width=17) (actual time=0.042..0.049 rows=2 loops=1)"

"  Recheck Cond: ((votes >= 60000) AND (votes <= 100000))"
"  Heap Blocks: exact=2"
"  ->  Bitmap Index Scan on movie_votes  (cost=0.00..5.24 rows=95 width=0) (actual time=0.027..0.027 rows=2 loops=1)"
"        Index Cond: ((votes >= 60000) AND (votes <= 100000))"
"Planning time: 0.395 ms"
"Execution time: 0.108 ms"
```

Q3

```
 "Seq Scan on movie (cost=0.00..248.48 rows=8145 width=17) (actual time=0.031..21.529 rows=8163 loops=1)"
"  Filter: ((yr >= '1900'::numeric) AND (yr <= '1990'::numeric))"
"  Rows Removed by Filter: 1736"
"Planning time: 0.723 ms"
"Execution time: 22.645 ms"
```

Q4

```
"Index Scan using movie_yr on movie (cost=0.29..17.29 rows=15 width=17) (actual time=0.051..0.083 rows=11 loops=1)"
"  Index Cond: ((yr >= '1890'::numeric) AND (yr <= '1900'::numeric))"
"Planning time: 0.728 ms"
"Execution time: 0.157 ms"
```

Compare the number of tuples selected from each query. Which query had lower selectivity (i.e. returns fewer rows)?

Q2 has lower selectivity as it returns only 2 tuples.

Q1 returns 1632 tuples.Q3 returns 8163 and Q4 returns 11.

Is the index on VOTES used for Q1 and Q2 queries? Give an intuitive explanation of the observed results.

Yes, index on votes is used on Q1 and Q2. Bitmap index scan on movie_votes is done in both of the queries. Q1 returns 100 heap blocks whereas Q2 gives only 2. Now bitmap heap scan on those 100 blocks takes more time than only on 2. Therefore, time for Q1 is quite high as compared to that of Q2.

Is the index on YR used for both Q3 and Q4 queries? Give an intuitive explanation of the observed results.

No, it is used only for Q4.

Sequential scanning is done in Q3 whereas index is used in Q4.

In Q3 the range of the query is high and index only is not sufficient to answer the query so anyhow we need to scan the file blocks and since the range is large so the index is not going to help much instead it will increase the overhead of looking the index table.

So, instead of using index in Q3, you can sequentially scan the file to return the values which satisfy the condition.

In Q4, index is helpful as it removes many blocks which does not satisfy the index condition. So, we can simply retrieve required data using the filtered blocks only.

Part C: String Matching and Index

Consider following SQL queries.

Q1: SELECT * FROM actor WHERE name LIKE 'B%';

Q2: SELECT * FROM actor WHERE name LIKE '%b';

Q3: SELECT * FROM actor WHERE substr(name, 1, 1) = 'B';

Run the explain plan for each Q1, Q2 and Q3, and submit the output.

Q1

```
"Bitmap Heap Scan on actor (cost=22.70..95.53 rows=635 width=18) (actual time=0.643..2.336 rows=570 loops=1)"
"  Filter: ((name)::text ~~ 'B%'::text)"
"  Heap Blocks: exact=65"
"  ->  Bitmap Index Scan on actor_name  (cost=0.00..22.54 rows=626 width=0) (actual time=0.561..0.561 rows=570 loops=1)"
"        Index Cond: (((name)::text ~>=~ 'B'::text) AND ((name)::text ~<~ 'C'::text))"
"Planning time: 0.682 ms"
"Execution time: 2.554 ms"
```

Q2

```
"Seq Scan on actor (cost=0.00..196.38 rows=212 width=18) (actual time=0.228..9.071 rows=30 loops=1)"
"  Filter: ((name)::text ~~ '%b'::text)"
"  Rows Removed by Filter: 10480"
"Planning time: 0.255 ms"
"Execution time: 9.126 ms"
```

Q3

```
"Seq Scan on actor (cost=0.00..222.65 rows=53 width=18) (actual time=0.073..16.787 rows=570 loops=1)"
"  Filter: (substr((name)::text, 1, 1) = 'B'::text)"
"  Rows Removed by Filter: 9940"
"Planning time: 0.189 ms"
"Execution time: 16.911 ms"
```

Compare the execution plans of query Q1 and Q2. What's the difference? Explain.

DBSI Assignment 2:
Deepak Kumar Sood(MT15013), Manisha Sharma(MT15031), Neha Rani Deswal(MT15040).

The exeution plan on Q1 uses the index on actor_name whereas the Q2 uses sequential scan. In postgresql indexes on varchar_pattern_ops uses B+ tree structure.

Q1 is the query on prefix whereas Q2 is query on postfix. So, we can't use the index for Q2. So, we have done seq scan in Q2. Therefore time taken for Q2 is more.

Compare the execution plans of query Q1 and Q3. Explain difference,Why the index on NAME is not always used for queries based on NAME attribute.

In Q1 we use the index actor_name whereas we are performing sequential scan for Q2 and Q3.

For Q1, Bitmap Index scan is used on actor_name index and then on the top bitmap heap scan is performed to retrieve the desire result. For Q2 and Q3, seq scan is perfomed to find the result.

The index on name is not always used for queries based on NAME attribute because here indexing on name is done using varchar_pattern_ops and structure for this query is B+ like. So, it is used for searching the name directly or the prefix of that but it can't help in queries on postfix and substrings.

Part D: Aggregate Operations

Consider following two equivalent SQL queries.

Q1: SELECT title FROM movie WHERE votes <= (SELECT MIN (votes) FROM movie);

Q2: SELECT title FROM movie WHERE votes <= ALL (SELECT votes FROM movie);

Run the explain plan for each Q1 and Q2 and submit the output.

Q1

```
"Seq Scan on movie  (cost=0.37..224.11 rows=3300 width=17) (actual time=7.479..9.564 rows=1 loops=1)"
"  Filter: (votes <= $1)"
"  Rows Removed by Filter: 9898"
"  InitPlan 2 (returns $1)"
"    ->  Result  (cost=0.36..0.37 rows=1 width=0) (actual time=0.102..0.102 rows=1 loops=1)"
"          InitPlan 1 (returns $0)"
"            ->  Limit  (cost=0.29..0.36 rows=1 width=4) (actual time=0.092..0.092 rows=1 loops=1)"
"                  ->  Index Only Scan using movie_votes on movie movie_1  (cost=0.29..733.51 rows=9899 width=4) (actual time=0.090..0.090 rows=1 loops=1)"
"                        Index Cond: (votes IS NOT NULL)"
"                        Heap Fetches: 1"
"Planning time: 0.649 ms"
"Execution time: 9.675 ms"
```

Q2

```
"Seq Scan on movie  (cost=0.00..1352588.00 rows=4950 width=17) (actual time=37.484..42.882 rows=1 loops=1)"
```

```
"  Filter: (SubPlan 1)"

"  Rows Removed by Filter: 9898"

"  SubPlan 1"

"    ->  Materialize  (cost=0.00..248.49 rows=9899 width=4) (actual time=0.000..0.002 rows=9 loops=9899)"

"          ->  Seq Scan on movie movie_1  (cost=0.00..198.99 rows=9899 width=4) (actual time=0.016..4.350 rows=9899 loops=1)"

"Planning time: 0.443 ms"

"Execution time: 43.310 ms"
```

Is there any difference in the execution plans for Q1 and Q2? Which query is more efficient to execute? Explain.

Yes. Index only scan is used for Q1 and no index is used for Q2. So, Q1 is more efficient than Q2 because Q2 first seq scan and materialize for computing where condition and then again seq scan for finding out the title whereas Q1 uses index only scan for computing where condition because index is on votes attribute and we are retrieving votes only in our condition so, only index is sufficient to answer it and no need of accessing file blocks for where part.

Part E: Alternate Join Strategies

Consider following three SQL queries for retrieving name of the movie and id's of their lead actors under different conditions.

Q1: SELECT m.title, c.actorid FROM casting c, movie m WHERE m.id=c.movieid;

Q2: SELECT m.title, c.actorid FROM casting c, movie m WHERE m.id=c.movieid AND c.ord = 21;

Q3: SELECT m.title, c.actorid FROM casting c, movie m WHERE m.id=c.movieid AND m.yr between 1890 and 1900;

Run the explain plan for each Q1, Q2 and Q3 and submit the output.

Q1

```
"Hash Join  (cost=322.73..940.04 rows=21150 width=21) (actual time=20.903..53.046 rows=21150 loops=1)"

"  Hash Cond: (c.movieid = m.id)"

"  ->  Seq Scan on casting c  (cost=0.00..326.50 rows=21150 width=8) (actual time=0.039..6.449 rows=21150 loops=1)"

"  ->  Hash  (cost=198.99..198.99 rows=9899 width=21) (actual time=20.747..20.747 rows=9899 loops=1)"

"        Buckets: 16384  Batches: 1  Memory Usage: 657kB"

"        ->  Seq Scan on movie m  (cost=0.00..198.99 rows=9899 width=21) (actual time=0.031..9.059 rows=9899 loops=1)"

"Planning time: 1.404 ms"

"Execution time: 55.190 ms"
```

DBSI Assignment 2:
Deepak Kumar Sood(MT15013), Manisha Sharma(MT15031), Neha Rani Deswal(MT15040).

It does a sequential scan of table movie, and computes the hash for each of its rows. Then, it does a sequential scan of casting, and for each row, computes the hash of the row and compares it to the movie hashed table. If it matches, the row will be put in the resulting set. If it doesn't match, the row is skipped.

Q2

```
"Nested Loop  (cost=4.77..237.75 rows=25 width=21) (actual time=0.219..0.877 rows=27 loops=1)"

"  ->  Bitmap Heap Scan on casting c  (cost=4.48..65.94 rows=25 width=8) (actual time=0.157..0.303 rows=27 loops=1)"

"        Recheck Cond: (ord = 21)"

"        Heap Blocks: exact=24"

"        ->  Bitmap Index Scan on casting_ord  (cost=0.00..4.47 rows=25 width=0) (actual time=0.103..0.103 rows=27 loops=1)"

"              Index Cond: (ord = 21)"

"  ->  Index Scan using movie_pkey on movie m  (cost=0.29..6.86 rows=1 width=21) (actual time=0.016..0.016 rows=1 loops=27)"

"        Index Cond: (id = c.movieid)"

"Planning time: 1.724 ms"

"Execution time: 1.152 ms"
```

Bitmap Index scan on casting_ord for checking condition c.ord and the bitmap heap scan on top to retrieve data is done.Index scan on movie_id for condition m.id=c.movieid is performed. Then nested loop join is performed for AND condition.

Q3

```
"Nested Loop  (cost=0.57..79.66 rows=32 width=21) (actual time=0.206..0.206 rows=0 loops=1)"

"  ->  Index Scan using movie_yr on movie m  (cost=0.29..17.29 rows=15 width=21) (actual time=0.037..0.062 rows=11 loops=1)"

"        Index Cond: ((yr >= '1890'::numeric) AND (yr <= '1900'::numeric))"

"  ->  Index Only Scan using casting_pkey on casting c  (cost=0.29..4.11 rows=5 width=8) (actual time=0.010..0.010 rows=0 loops=11)"

"        Index Cond: (movieid = m.id)"

"        Heap Fetches: 0"

"Planning time: 1.979 ms"

"Execution time: 0.327 ms"
```

Index scan is done on movie_yr on movie to check the between condition. Index only scan is done on casting primary key to check another condition. Nested loop join is performed for for AND condition.

DBSI Assignment 2:
Deepak Kumar Sood(MT15013), Manisha Sharma(MT15031), Neha Rani Deswal(MT15040).

Compare and contrast three execution plans for Q1, Q2 and Q3 in terms of join algorithm and indexes used? Is there any difference? Why?

Q1 uses hash join, Q2 and Q3 uses nested loop join.

Q1 does sequential scan of the tables movie and casting, so no indexes used for Q1. Q2 uses bitmap index scan and bitmap heap scan for the condition c.ord=21 and it also uses index scan for another condition. Q3 uses index scan and index only scan.

Since Q1 uses no index, so it takes more time to execute. Q3 uses index and index only scan so it uses less time and Q2 uses bitmap heap scan along with two more scans so it uses more time than Q3 but less than Q1.

Question 2:

Rank order the four join strategies for the following join operation:

Join table DEPARTMENT and EMPLOYEE on the condition Mgr_ssn = Ssn

Given Data:
Number of records in DEPARTMENT table = 125
Total number of disk blocks for DEPARTMENT table = 13
Number of records in EMPLOYEE table = 10000
Total number of disk blocks for the EMPLOYEE table = 200

Secondary index on mgr-ssn with level (x_mgr-ssn) = 2
Primary index on ssn with level (x_ssn) =4
Selection cardinality ($S_{Mgr\text{-}ssn}$) = 1

1. **J1-Nested-loop join:**
   A **nested loop join** is a naive algorithm that joins two sets by using two **nested loops**.
   - With Employee as outer loop:
     C = b_E + (b_E * bD) + ((js * |E| * |D|)/bfr_{ED})
     = 2000 + (2000 * 13) + (((1/10000) * 10000 * 125)/4)
     = 2000 + 26000 + 32
     = 28032

   - With Department as outer loop:

C = b_E + (b_E * bD) + ((js * |E| * |D|)/bfr_{ED})
= 13 + (2000 * 13) + (((1/10000) * 10000 * 125)/4)
= 13 + 26000 + 32
= 26045

2. **J2-Single-loop join:**
   - With Employee as outer loop:

C = b_E + |E| * (x_Mgr-ssn + 1 + $S_{Mgr\text{-}ssn}$)
= 2000 + 10000 * (2 + 1 + 1) + (((1/10000) * 10000 * 125)/4)
= 2000 + 40000 + 32
= 42032

   - With Department as outer loop:

C = b_D + |D| * (x_ssn + 1) + (((1/10000) * 10000 * 125)/4)

= 13 + 125 * (4 + 1) +32
= 13 + 125 * 5 + 32
= 13 + 625 +32
= 670

3. J3-sort merge sort

The **sort-merge join** (also known as merge join) is a join algorithm and is used in the implementation of a relational database management system.

- If they are sorted:

$C = b_E + b_D + ((js * |E| * |D|)/bfr_{ED})$
$= 2000 + 13 + (((1/10000) * 10000 * 125)/4)$
$= 2,045$

- If they are not sorted:

When the tables are not sorted, both tables need to sort on the attribute of join attribute. Since there is a primary index on ssn of Employee, it can be consider as sorted. Department table need to sort.
$n_r = \lceil b_D/3 \rceil = \lceil 13/3 \rceil = 5$
$d_M = Min(n_b-1,n_R) = Min(2,5) = 2$
$C = b_E + b_D + (2*b_D + 2*b_D*(\log_{dM}(n_r))) + ((js * |E| * |D|)/bfr_{ED})$

$= 2000 + 13 + 2*13 + 2*13*\lceil \log_2(5) \rceil + 32$
$= 2000 + 13 + 26 + 78 + 32$
$= 2149$

4. J4-Partition hash join

$C = 3*(b_E + b_{S)}) + ((js * |E| * |D|)/bfr_{ED})$
$= 3*(2000+13) + (((1/10000) * 10000 * 125)/4)$
$= 3*2013 + 32$
$= 6039 + 32$
$= 6071$

a. when records in the tables are sorted with join attribute:

j2 < j3 < j4 < j1

b. when records in the tables are not sorted and primary index on ssn is considered as sorted:

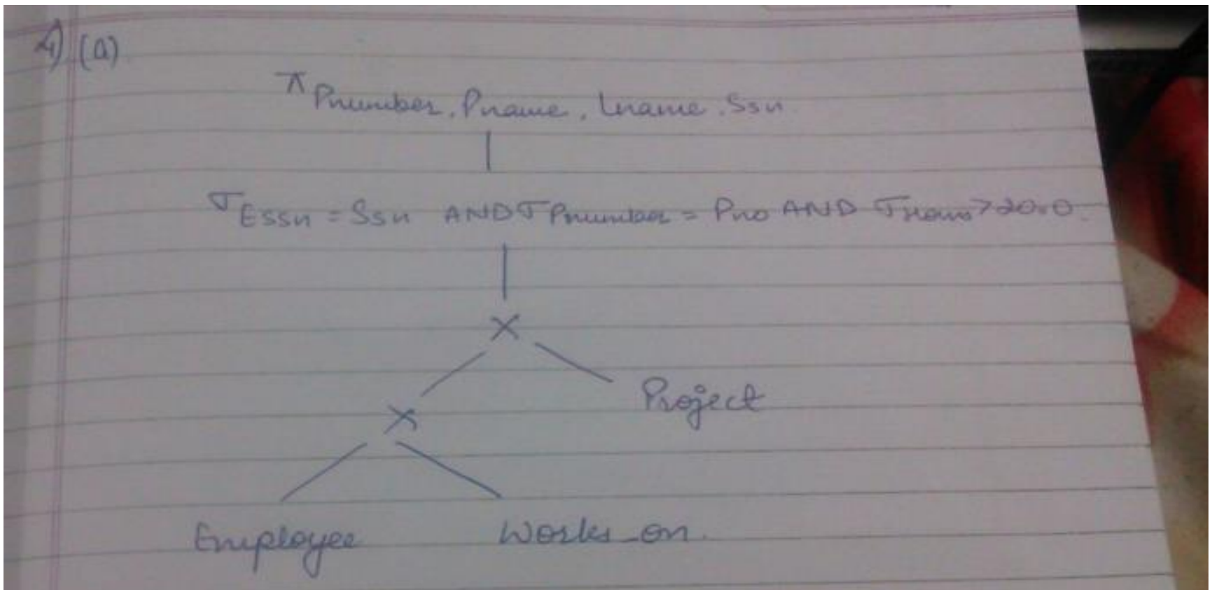
j2 < j3 < j4 < j1

DBSI Assignment 2:
Deepak Kumar Sood(MT15013), Manisha Sharma(MT15031), Neha Rani Deswal(MT15040).

Question 4:

4). Given Query:

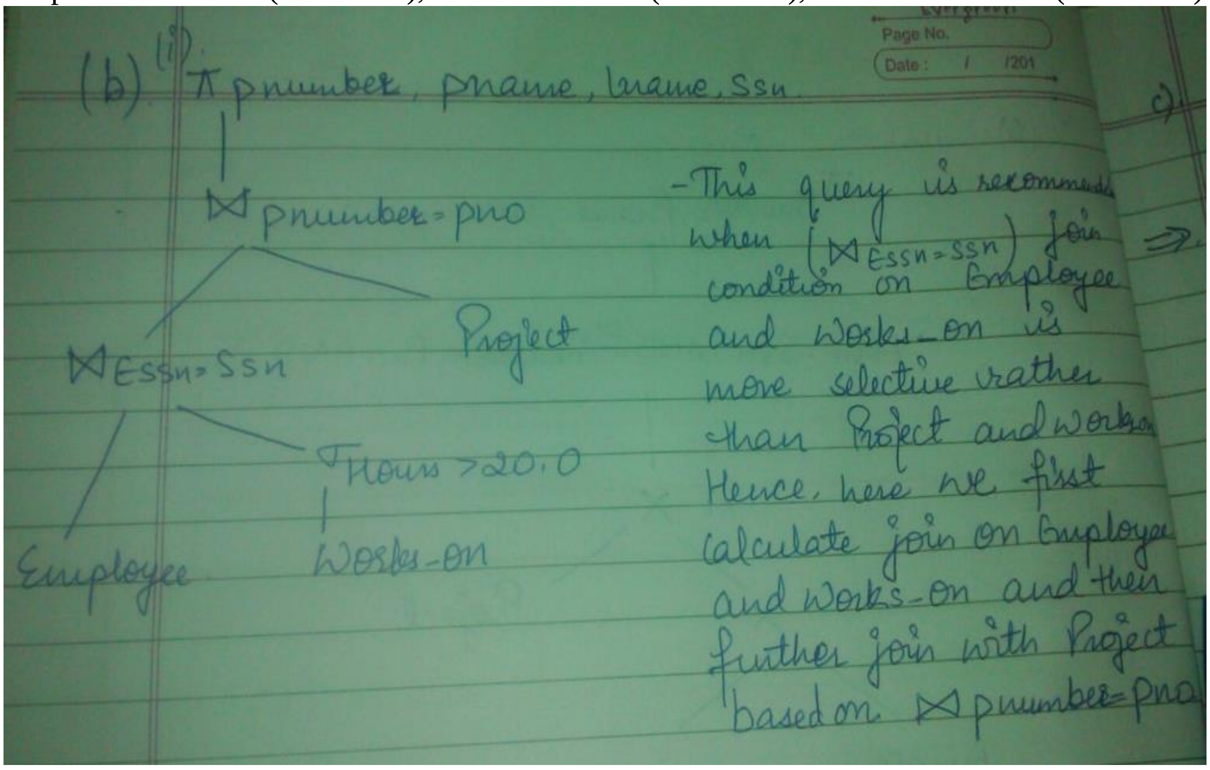
Select Pnumber, Pname, Lname, Ssn
From PROJECT, WORKS_ON, EMPLOYEE
Where Pnumber=Pno AND Ssn=Essn AND Hours>20.0


4) (a)
π Pnumber, Pname, Lname, Ssn
σ Essn = Ssn AND σ Pnumber = Pno AND σ Hours > 20.0
×
×
Project
Employee
Works_on

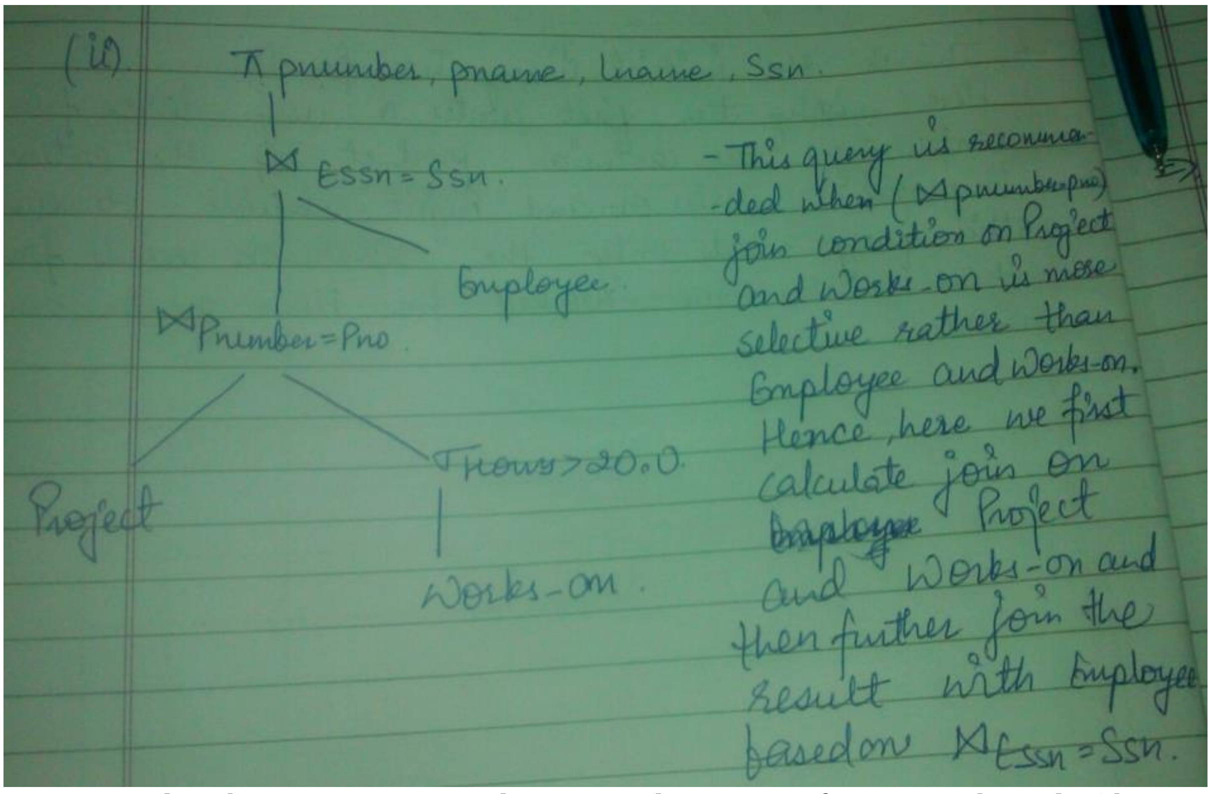
→ This is the initial Query Tree for ①.
Here existing tree first creates a very large file containing the cartesian product of the entire Employee, Works_on and Project relations. However, this query needs only the Works_on records for those who have worked for Hours greater than 20.0.

→ Above is the canonical Tree form of query given. Here cartesian product is performed first and then all select conditions are imposed. This is the worst form of query Tree in terms of cost required. One cartesian product is itself very costly as it considers all tuples of two given relation, however here are 2 cartesian products, so i.e. the worst scenario to be considered.

DBSI Assignment 2:
Deepak Kumar Sood(MT15013), Manisha Sharma(MT15031), Neha Rani Deswal(MT15040).



– This query is recommende when ($\bowtie_{Essn=ssn}$) join condition on Employee and Works-on is more selective rather than Project and works-on. Hence, here we first calculate join on Employee and Works-on and then further join with Project based on $\bowtie_{pnumber=pno}$



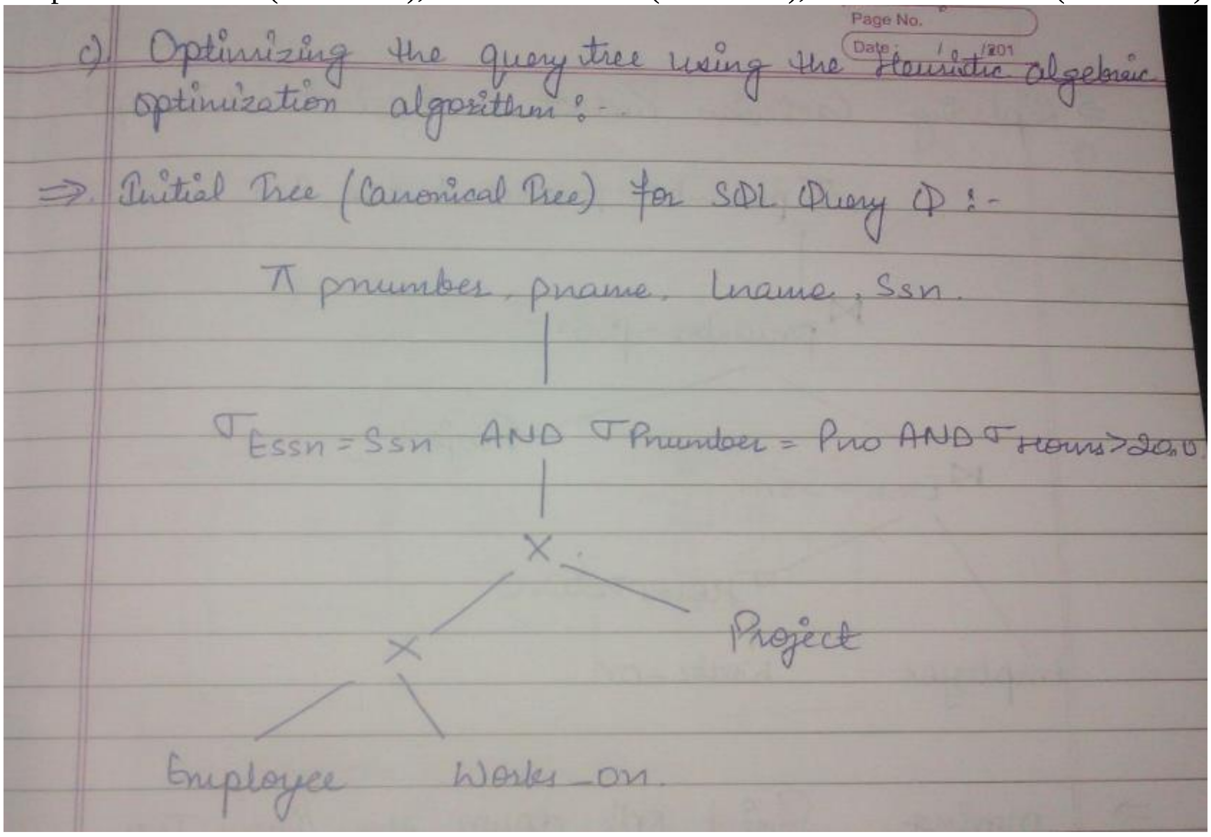
– This query is recommended when ($\bowtie_{pnumber=pno}$) join condition on Project and Works-on is more selective rather than Employee and Works-on. Hence, here we first calculate join on ~~Employee~~ Project and Works-on and then further join the result with Employee based on $\bowtie_{Essn=Ssn}$.

As compared to these two scenarios, the canonical tree query form is much costly (shown as 4(a) where the Cartesian product among given relations is performed). The query results as per requirement. But its cost is very much as Cartesian product compares each tuple of one relation with all tuples of another relation. One Cartesian product itself costs alot and here two Cartesian products are used.

4). (c). Here we are optimizing the query tree using Heuristic algebraic optimization step by step. Each image depicts steps followed to get the optimized query tree.

First step is to draw Canonical tree of query to be executed.

DBSI Assignment 2:
Deepak Kumar Sood(MT15013), Manisha Sharma(MT15031), Neha Rani Deswal(MT15040).

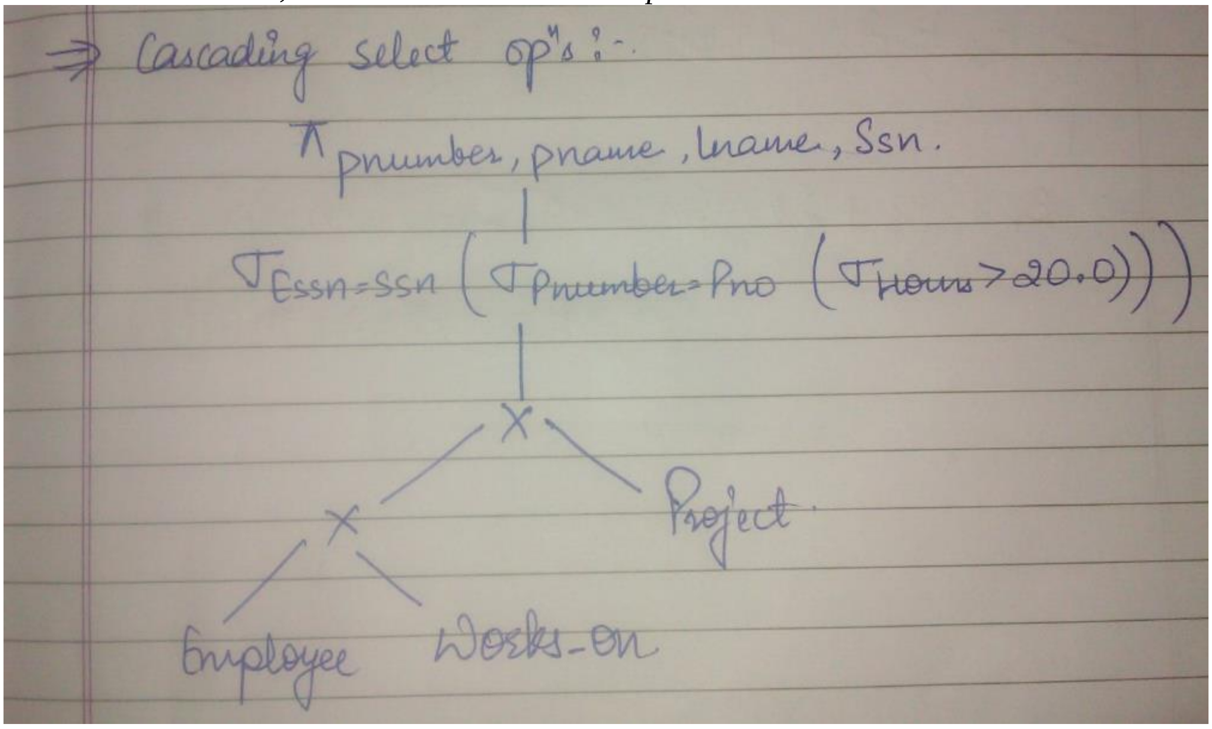


c) Optimizing the query tree using the Heuristic algebraic optimization algorithm:-

⇒ Initial Tree (Canonical Tree) for SQL Query Q:-

Page No.
Date: / /201
π pnumber, pname, lname, Ssn.
$\sigma_{Essn=Ssn}$ AND $\sigma_{Pnumber=Pno}$ AND $\sigma_{Hours>20.0}$
×
×
Project
Employee
Works_on.

Then we remove AND, then cascade all SELECT operations:

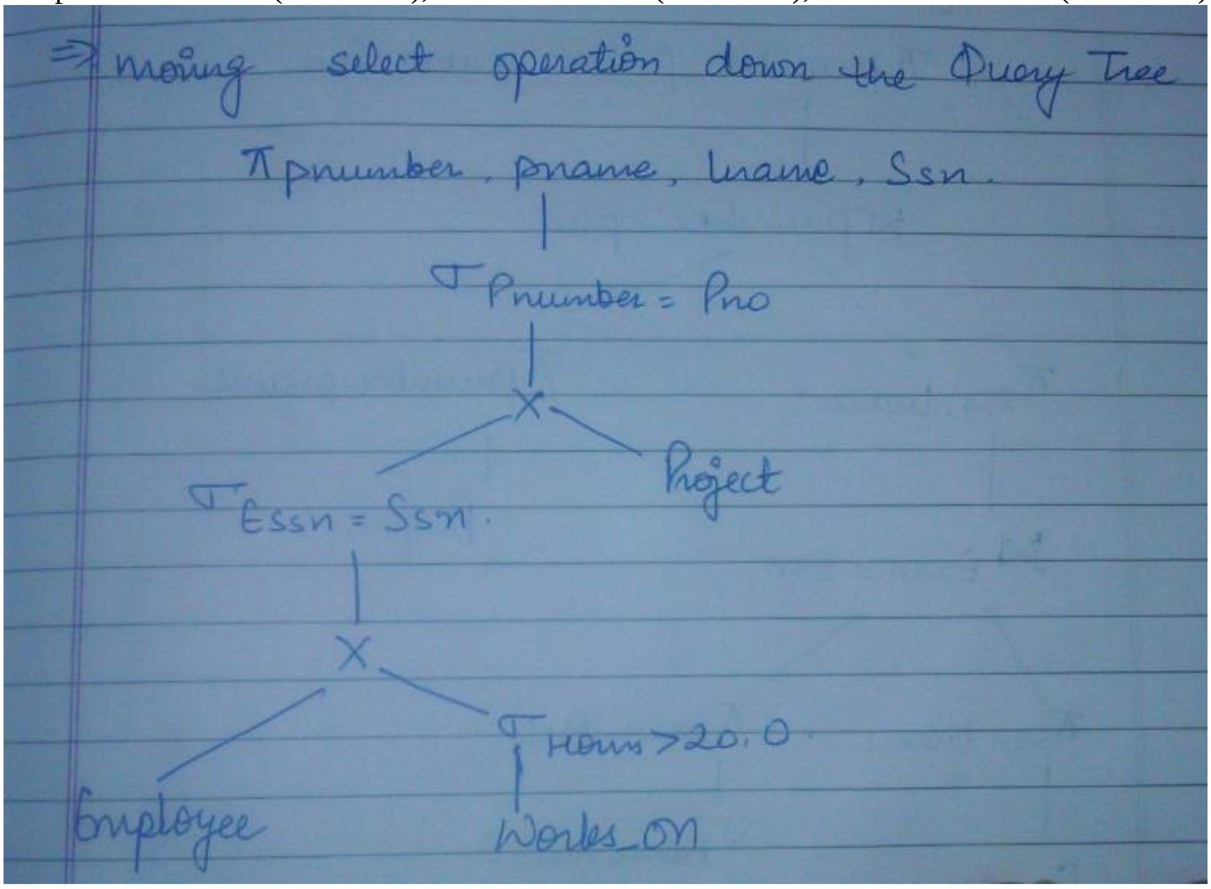


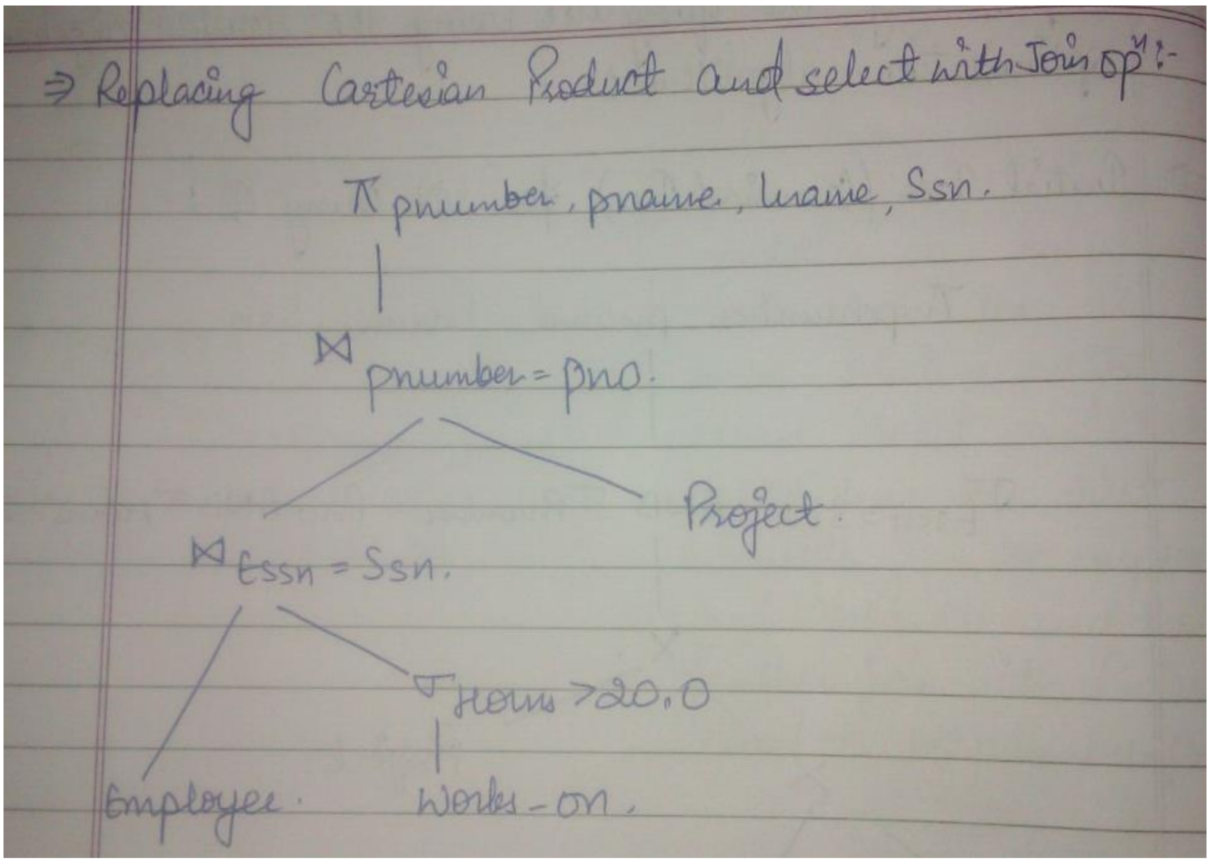
⇒ Cascading select opn's:-

π pnumber, pname, lname, Ssn.
$\sigma_{Essn=Ssn}(\sigma_{Pnumber=Pno}(\sigma_{Hours>20.0}))$
×
×
Project.
Employee
Works-on

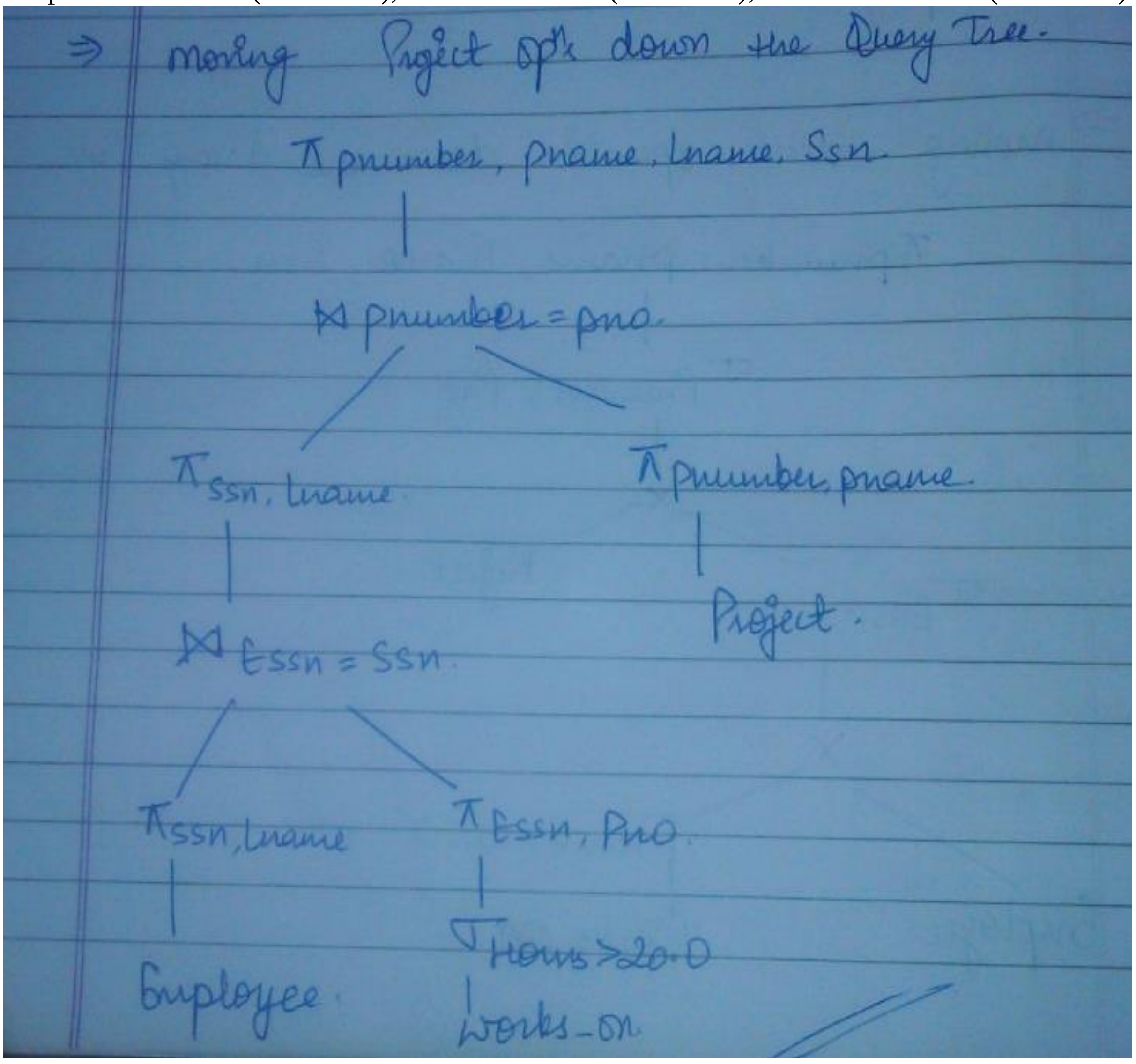
Further we move Select operation down the query tree to make it highly Selective. Produces less tuples from Works_on and less cost further.

DBSI Assignment 2:
Deepak Kumar Sood(MT15013), Manisha Sharma(MT15031), Neha Rani Deswal(MT15040).


⇒ moving select operation down the Query Tree
π pnumber, pname, lname, Ssn.
σ Pnumber = Pno
×
Project
σ Essn = Ssn.
×
σ Hours > 20.0
Employee
Works_on


⇒ Replacing Cartesian Product and select with Join opn :-
π pnumber, pname, lname, Ssn.
⋈ pnumber = pno.
Project.
⋈ Essn = Ssn.
σ Hours > 20.0
Employee.
Works-on.

DBSI Assignment 2:
Deepak Kumar Sood(MT15013), Manisha Sharma(MT15031), Neha Rani Deswal(MT15040).

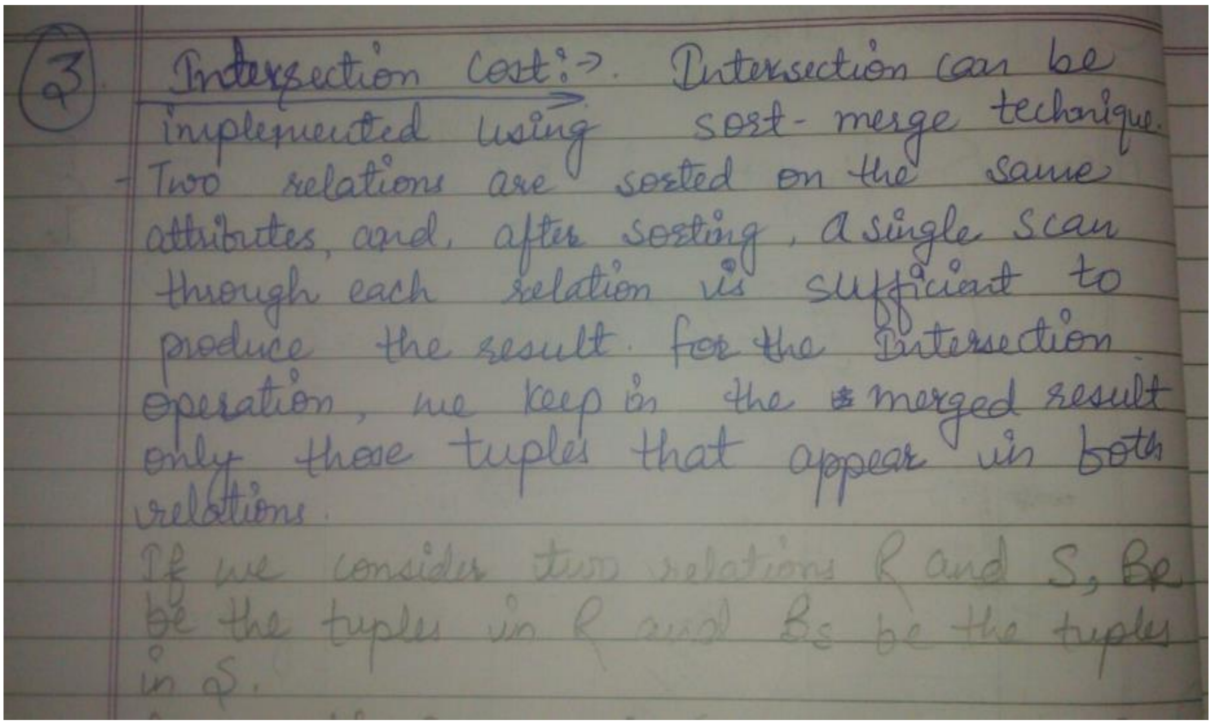


Note: The CARTESIAN PRODUCT operation *R* · *S* is quite expensive because its result includes a record for each combination of records from *R* and *S*. Also, each record in the result includes all attributes of *R* and *S*. If *R* has *n* records and *j* attributes, and *S* has *m* records and *k* attributes, the result relation for *R* · *S* will have *n* * *m* records and each record will have *j* + *k* attributes. Hence, it is important to avoid the CARTESIAN PRODUCT operation and to substitute other operations such as join during query optimization.

Deepak Kumar Sood(MT15013), Manisha Sharma(MT15031), Neha Rani Deswal(MT15040).

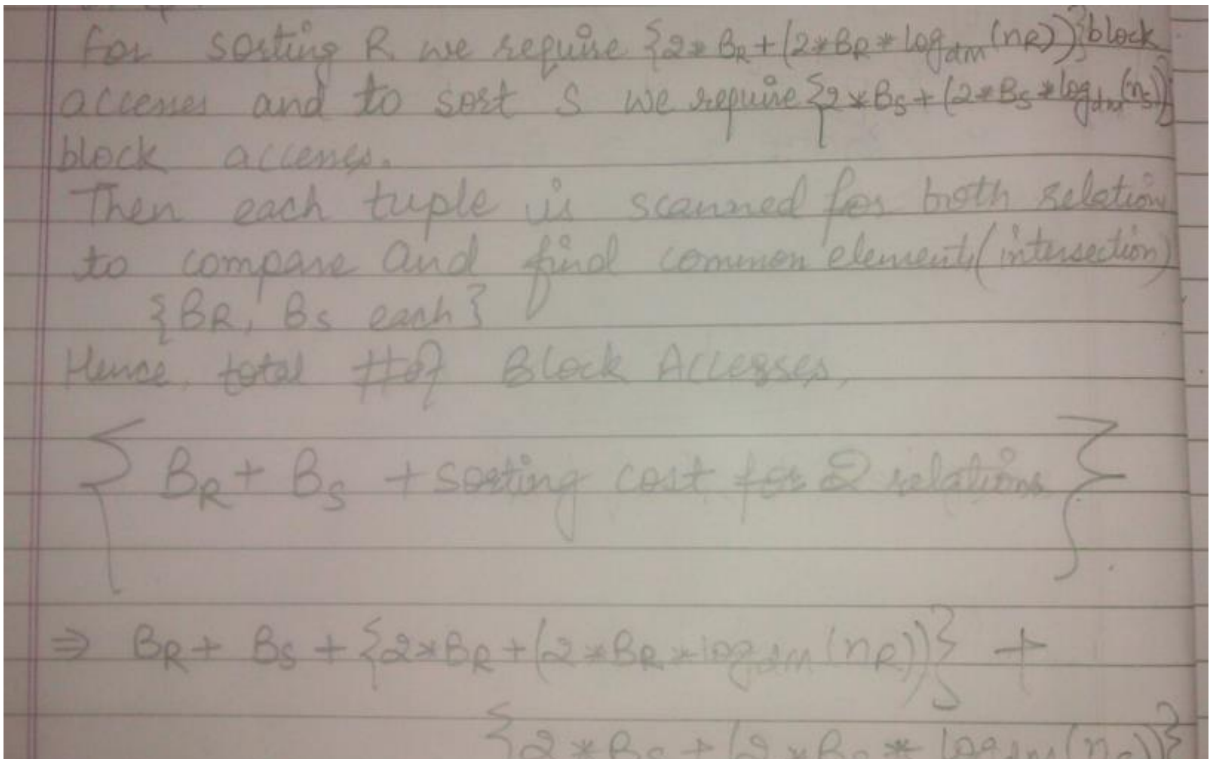
Question 3:

3). INTESECTION is a set operation that applies to only **type-compatible** (or union-compatible) relations which have the same number of attributes and the same attribute domains. To implement this, we use **sort-merge technique** where the given two relations are sorted on the same attributes and after sorting using a single scan through each relation the result gets produced. For the INTERSECTION operation, we keep in the merged result only those tuples that appear in *both sorted relations.*



③ Intersection cost:→. Intersection can be implemented using sort-merge technique.
- Two relations are sorted on the same attributes, and, after sorting, a single scan through each relation is sufficient to produce the result. for the Intersection operation, we keep in the merged result only those tuples that appear in both relations.

If we consider two relations R and S, B_R be the tuples in R and B_S be the tuples in S.



For sorting R we require $\{2*B_R+(2*B_R*\log_{dM}(n_R))\}$ block accesses and to sort S we require $\{2*B_S+(2*B_S*\log_{dM}(n_S))\}$ block accesses.

Then each tuple is scanned for both relation to compare and find common elements (intersection) $\{B_R, B_S \text{ each}\}$

Hence, total # of Block Accesses,

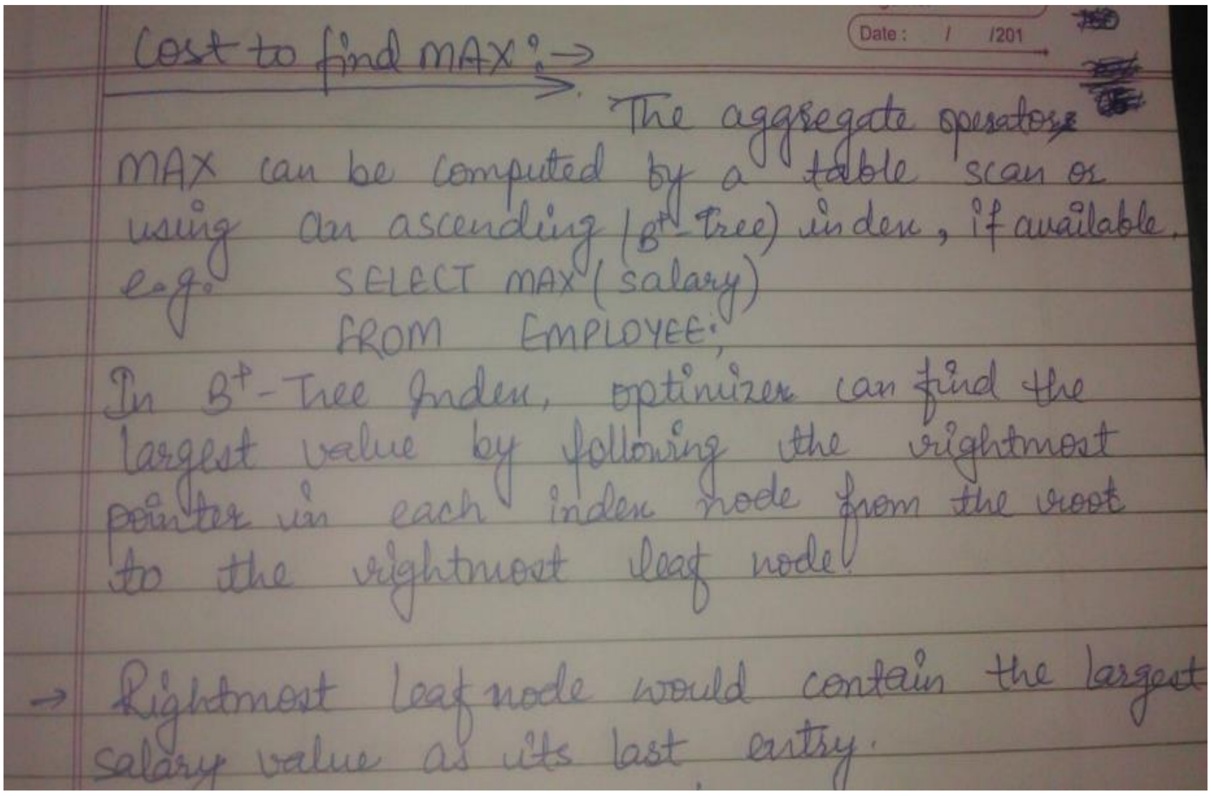
$$\{B_R + B_S + \text{sorting cost for 2 relations}\}$$

$$\Rightarrow B_R + B_S + \{2*B_R+(2*B_R*\log_{dM}(n_R))\} + \{2*B_S+(2*B_S*\log_{dM}(n_S))\}$$

Here External sorting has been used where n-way merge sort technique has been used. Br and Bs are number of File blocks in R and S respectively. dM is the degree of merging. nR and nS are number of runs in relation R and S respectively.

DBSI Assignment 2:
Deepak Kumar Sood(MT15013), Manisha Sharma(MT15031), Neha Rani Deswal(MT15040).

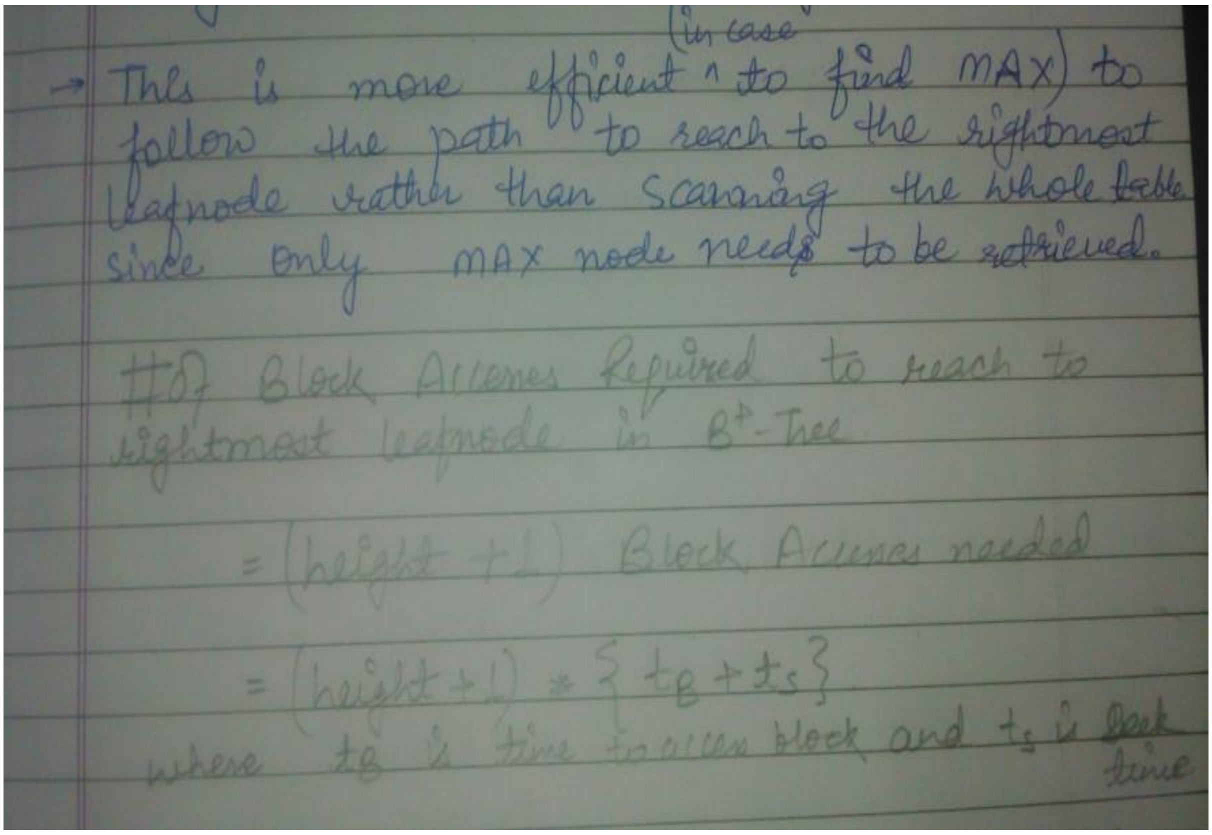
The aggregate operator MAX when applied to an entire table, B+ Tree is used to get the MAX value. Here the cost will be ***(height+1)*** as the Maximum element is at the rightmost position at leaf level of the B+ Tree.



Date : / /201

Cost to find MAX :→ The aggregate operator MAX can be computed by a table scan or using an ascending (B^+-Tree) index, if available.
e.g. SELECT MAX (salary)
FROM EMPLOYEE;
In B^+-Tree Index, optimizer can find the largest value by following the rightmost pointer in each index node from the root to the rightmost leaf node.

→ Rightmost Leaf node would contain the largest salary value as its last entry.



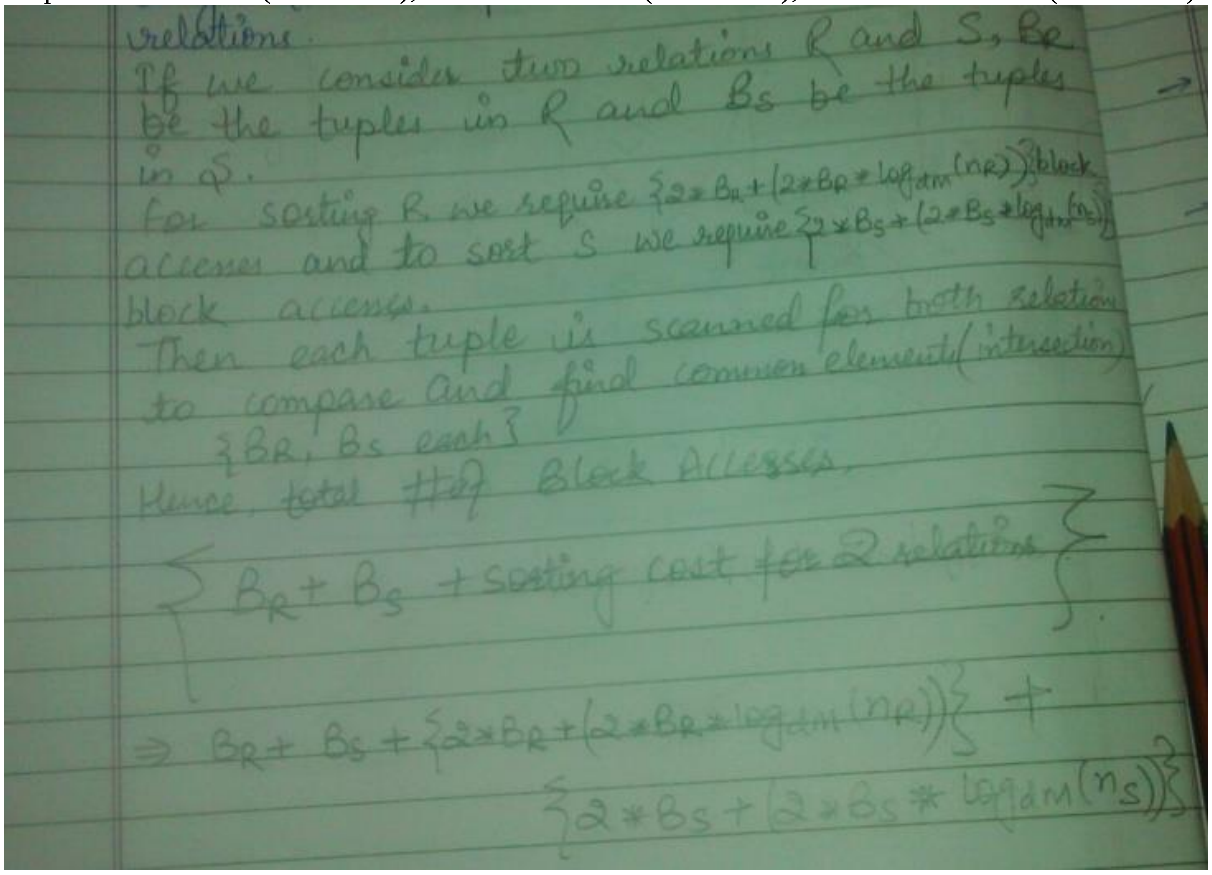
→ This is more efficient (in case to find MAX) to follow the path to reach to the rightmost leafnode rather than scanning the whole table since only MAX node needs to be retrieved.

# of Block Accesses Required to reach to rightmost leafnode in B^+-Tree

= (height + 1) Block Accesses needed

= (height + 1) * { $t_B + t_S$ }
where t_B is time to access block and t_S is seek time

DBSI Assignment 2:
Deepak Kumar Sood(MT15013), Manisha Sharma(MT15031), Neha Rani Deswal(MT15040).



relations.
If we consider two relations R and S, B_R be the tuples in R and B_S be the tuples in S.
For sorting R we require $\{2 * B_R + (2 * B_R * \log_{dm}(n_R))\}$ block accesses and to sort S we require $\{2 * B_S + (2 * B_S * \log_{dm}(n_S))\}$ block accesses.
Then each tuple is scanned for both relation to compare and find common elements (intersection) $\{B_R, B_S$ each$\}$
Hence, total # of Block Accesses,

$$\left\{ B_R + B_S + \text{sorting cost for 2 relations} \right\}.$$

$$\Rightarrow B_R + B_S + \{2 * B_R + (2 * B_R * \log_{dM}(n_R))\} + \{2 * B_S + (2 * B_S * \log_{dM}(n_S))\}$$

Br + Bs + {2 * Br + (2 * Br * $\log_{dm}$ (nR))} + {2 * Bs + (2 * Bs * $\log_{dm}$ (nS))} (If External Sort is used).

//Br + Bs + Br $\log_2$ Br + Bs $\log_2$ Bs. (If we estimate the cost of sorting an external file by B $\log_2$ B).