1. Explain the Algorithms for implementing Join methods?

Join algorithms in Database

There are 2 algorithms to compute natural join & conditional join of two relations in database:-

1. Nested loop join
2. Block nested loop join

To understand these algorithms we will assume there are two relations, relation R and relation S. Relation R has T_R tuples and occupies B_R blocks. We will also assume relation R is the outer relation & S is the inner relation.

1. Nested loop join :-

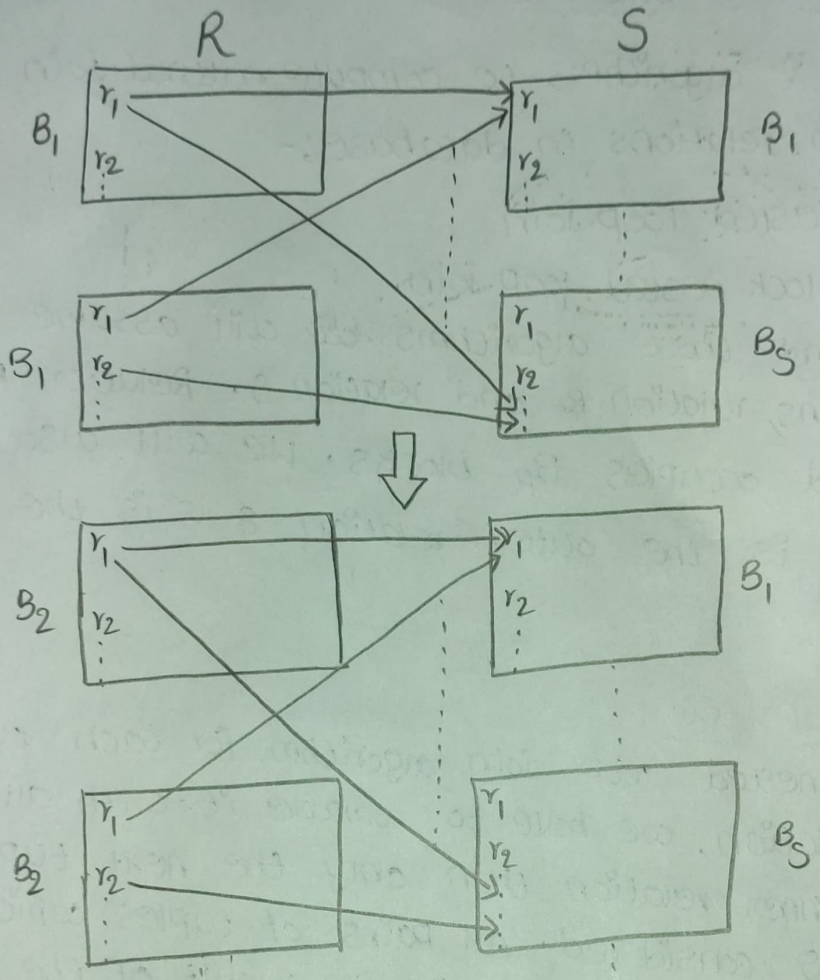
In the nested loop join algorithm, for each tuple in outer relation, we have to compare it with all the tuples in the inner relation then only the next tuple of outer relation is considered. All pairs of tuples which satisfy the condition are added in the result of the join.

```
for each tuple tR in TR do
    for each tuple ts in Ts do
        compare (tR, ts) if they satisfies the condition add them
            in the result of the join
    end
end
```

This algorithm is called nested join because it consists of nested for loops

Cases

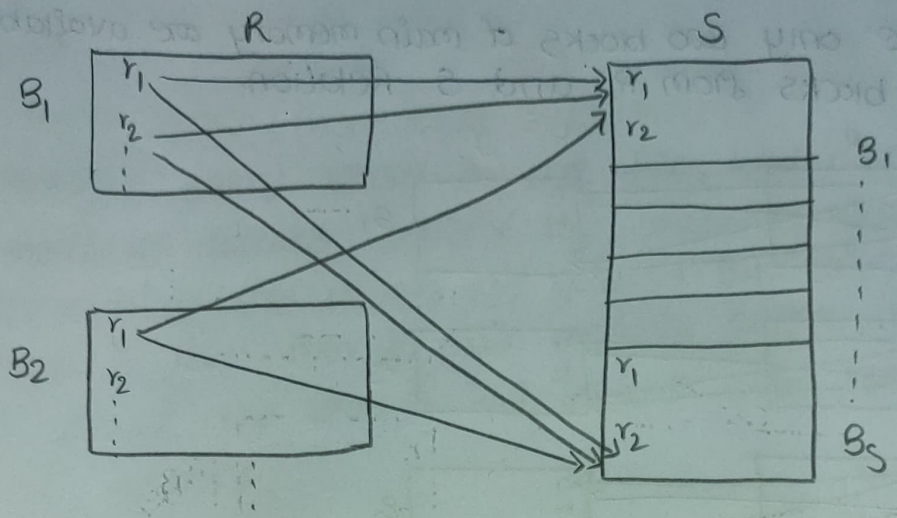
Case-1:- Assume only two blocks of main memory are available to store blocks from R and S relation.



For each tuple in relation to R, we have to transfer all blocks of relation S and each block of relation R should be transferred only once.

So, the total block transfers needed = $T_R * B_S + B_R$

Case-2:- Assume one relation fits entirely in the main memory and there is at least space for one extra block.



In this case, the blocks of relation S are only transferred once & kept in the main memory & the blocks of relation R are transferred sequentially. So, all the blocks of both the relation are transfered only once.

So, the total block transfers needed = $B_R + B_S$

The relation with a less number of blocks should be the outer relation to minimizing the total no.of blocks access required in the main memory to complete the join.

i.e $\min(B_R, B_S) + 1$ is the minimum no.of blocks in the main memory required to join two relations so that no block is transfered more than once.

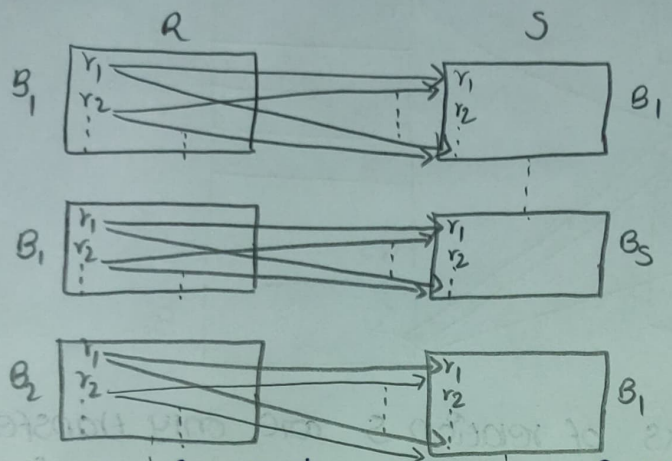
2. Block Nested loop join:-

In block nested loop join, for a block of outer relation, all the tuples in that block are compared with all the tuples of the inner relation, then only the next block of outer relation is considered. All pairs of tuples which satisfy the condition are added in the result of the join.

```
for each block bR in BR do
  for each block bS in BS do
    for each tuple tR in TR do
      for each tuple tS in TS do
        compare(tR, tS) if they satisfies the condition add them in the result
                                                          of the join.
      end
    end
  end
end
```

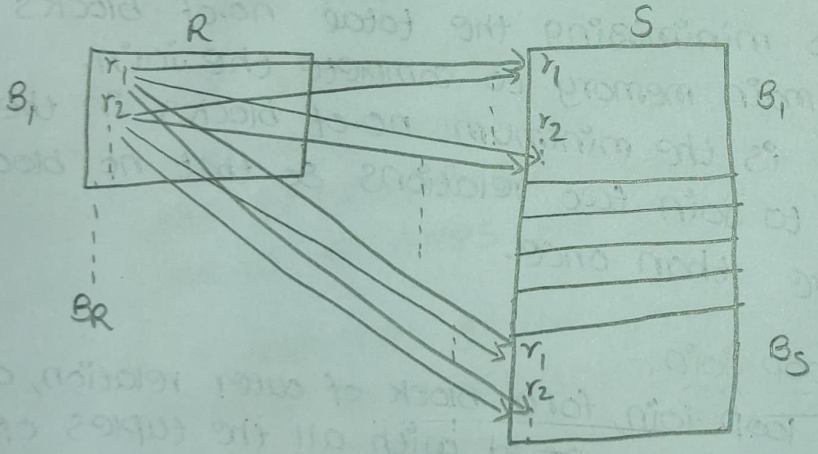
Cases

Case-1:- Assume only two blocks of main memory are available to store blocks from R and S Relation.



For each block of relation R, we have to transfer all blocks of relation S & each block of relation R should be transferred only once. So, total block transfers needed = $B_R + B_R * B_S$

Case-2:- Assume one relation fits entirely in the main memory & there is atleast space for one extra block.



In this case, total block transfers needed are similar to nested loop join.

Block nested loop join algorithm reduces the access cost compared to nested loop join if the main memory space allocated for join is limited.

2 a) Explain the methods for implementing selection operations?

Methods

S1 - Linear search (brute force algorithm):-
Retrieve every record in the file, and test whether its attribute values satisfy the selection condition. Since the records are grouped into disk blocks, each disk block is read into a main memory buffer, and then a search through the records within the disk block is conducted in main memory.

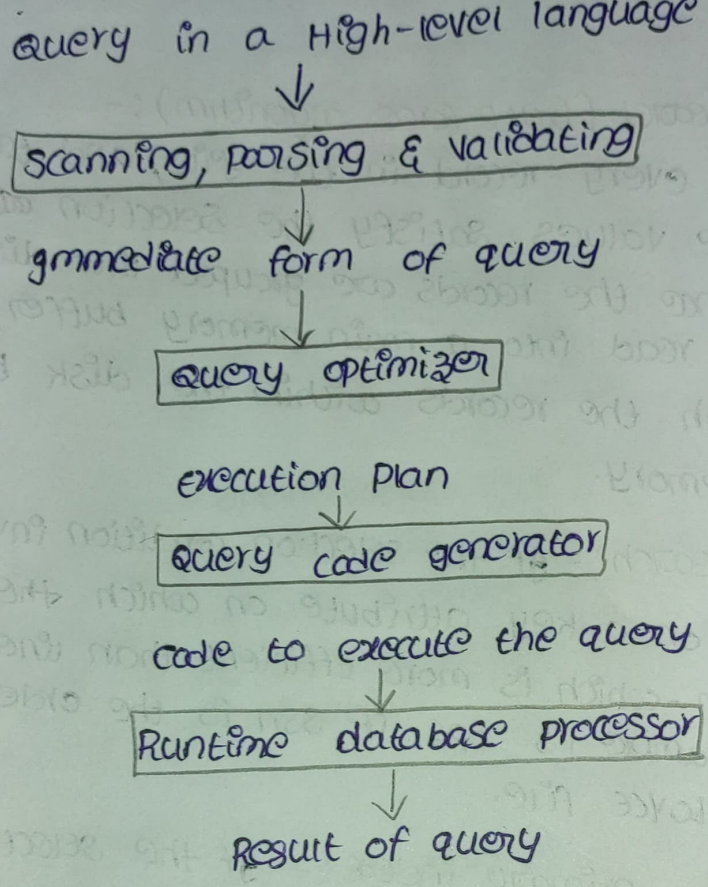
S2 - Binary search:- If the selection condition involves an equality comparision on a key attribute on which the file is ordered, binary search - which is more efficient than linear search - can be used. An example is op1 if ssn is the ordering attribute for the EMPLOYEE file.

S3a - Using a primary index:- If the selection condition involves an equality comparision on a key attribute with a primary index - for example, ssn = '123456789' in op1 - use the primary index to retrieves the single record.

S3b - Using a hash key:- If the selection condition involves an equality comparison on a key attribute with a hash key - for example ssn = '123456789' in op1 - use the hash key to retrieve the record.

S4 - Using primary index to retrieve multiple records:- If the comparison condition is >, >=, < or <= on a key field with a primary index - to find for example, Dnumber > 5 in op2 - use the index to find the record satisfying the corresponding equality condition (Dnumber = 5), then retrieve all subsequent records in the file. For the condition Dnumber < 5, retrieve all the preceding records.

2b Explain the steps involved in the query processing with neat diagram?



STEP-1:-

Parser: During parse call the database performs the following checks syntax check, semantic check & shared pool check after converting the query into relational algebra.

1. Syntax check:- concludes SQL syntax validity.

   Ex:- SELECT * FORM employee

   Here error of worong spelling of FROM is given by this check.

2. Semantic check:- Determines whether the statement is meaningful or not. Example: query contains a tablename which does not exist is checked by this check.

3. Shared Pool check:- every query posses a hash code during its execution. So this check determines existence of written hash code in shared pool if code exists in shared pool then database will not take additional steps for optimization and execution.

Hard Parse & soft Parse:- If there is a fresh query & its hash code does not exist in shared pool then that query has to pass through from the additional steps known as hard parsing otherwise if hash code exists then query does not passes through additional steps. It just passes directly to execution engine (refer detailed diagram). this is known as soft parsing.

Hard parse includes following steps- optimizer and Row Source generation

STEP-2:-

optimizer:- During optimization stage, database must perform a hard parse atleast for one unique DML statement and perform optimization during this parse. this database never optimizes DDL unless it includes a DML component such as subquery that requires optimization.

Row source generation:-

The Row source generation is a software that receives a optional execution plan from the optimizer and produces an iterative execution plan that is usable by the rest of the database. the iterative plan is the binary program that when executes by the sql engine produces the result set.

STEP-3:-

Execution engine:- Finally runs the query and display the required result.

3. Explain the algorithm for implementing
   a) External sorting
   b) outer join

a) External sorting:-

External sorting refers to sorting algorithm that have are suitable for large files of records stored on disk that do not fit entirely in main memory such as most database files.

The sort-merge algorithm like other database algorithms requires buffer space in main memory where the actual sorting & merging of the runs is performed.

In the sorting phase runs of the file that can fit in the available buffer space are read into main memory sorted using an internal sorting algorithm and written back to disk as temporary sorted subfiles.

In the merging phase, the sorted runs are merged during one or more passes. Each merge pass can have one or more merge steps.

```
set
    i←1;
    j←b;            {size of the file in blocks}
    k←n_B;          {size of buffer in blocks}
    m←⌈(j/k)⌉;
{Sorting Phase}
while (i≤m)
do{
    read next k blocks of the file into the buffer or if
    there are less than k blocks remaining, then read
    in the remaining blocks;
    Sort the records in the buffer and write as a temporary
                                                subfile;
        i←i+1;
}
```

{merging phase: merge subfiles until only 1 remains}
set i←1;
p←⌈log$_{k-1}$ m⌉ {p is the number of passes for the merging phase}
j←m;
while(i≤p)
do{
n←1;
q←⌈j/(k-1)⌉; {number of subfiles to write in this pass}
while(n≤q)
do{
read next k-1 subfiles or remaining subfiles one block at a time;
merge and write as new subfile one block at a time;
n←n+1;
}
j←q;
i←i+1;
}

The performance of the sort-merge algorithm can be measured in the no.of disk block reads and writes before the sorting of the whole files is completed. The following formula approximates this cost:

$$(2*b)+(2*b*(\log_{dM} n_R))$$

* the 1st term (2*b) represents the no.of block access for the sorting phase, since each file block is accessed twice: once for reading into a main memory buffer & once for writing the sorted records back to disk into one of the sorted subfiles.
* the second term represents the no.of block accesses for the merging phase. During each merge pass, a no.of disk

blocks approximately equal to the original file blocks b is read & written. Since the no.of merge passes is $(\log_{d_M} n_R)$, we get the total merge cost of $[2*b*(\log_{d_M} n_R)]$.

The minimum no.of main memory buffers needed is $n_B=3$, which gives a d_M of 2 and an n_R of $[(b/3)]$. The minimum d_M of 2 gives the worst-case performance of the algorithm, which is:

$$(2*b)+(2*(b*(\log_2 n_R)))$$

b) Outer join:-

Outer join can be computed by modifying one of the join algorithms, such as nested-loop join or single-loop join. For example to compute a left outer join, we use the left relation as the outer loop or single-loop because every tuple in the left relation must appear in the result. If there are matching tuples in the other relation, the joined tuples are produced and saved in the result. However, if no matching tuple is found, the tuple is still included in the result but is padded with NULL value(s). The sort-merge and hash-join algorithms can also be extended to compute outer joins.

Theoritically, outer join can also be computed by executing a combination of relational algebra operators. For example, the left outer join operation shown above is equivalent to the following sequence of relational operations:

1. Compute the (inner) JOIN of the EMPLOYEE and DEPARTMENT tables.

TEMP1 $\leftarrow \pi_{Lname, Fname, Dname}$ (EMPLOYEE $\bowtie_{Dno=Dnumber}$ DEPARTMENT)

2. Find the EMPLOYEE tuples that do not appear in the (inner) JOIN result.

   TEMP2 $\leftarrow \pi_{Lname, Fname}(EMPLOYEE) - \pi_{Lname, Fname}(TEMP1)$

3. Pad each tuple in TEMP2 with a NULL Dname field

   TEMP2 $\leftarrow$ TEMP2 x NULL

4. Apply the UNION operation to TEMP1, TEMP2 to produce the LEFT OUTER JOIN result.

The cost of the outer join as computed above would be the sum of the costs of the associated steps (inner join, projections, set difference & union). However, note that step 3 can be done as the temporary relation is being constructed in step 2; that is, we can simply pad each resulting tuple with a NULL. In addition, in step 4, we know that the two operands of the union are disjoint (no common tuples), so there is no need for duplicate elimination.

4. Explain the notation for query trees and query Graphs & also explain the heuristic optimization of query trees?

A query tree is a tree data structure that corresponds to a relational algebra expression. It represents the input relations of the query as leaf nodes of the tree & represents a relational algebra operations as internal nodes. An execution of the query tree consists of executing an internal node operation whenever its operands are available & then replacing that internal node by the relation that results from executing the operation. The order of execution of operations starts at the leaf nodes, which represents the input database relations for the query & ends at the root node, which represents the final operation of the query. The execution terminates when the root node operation is executed & produces the result relation for the query.

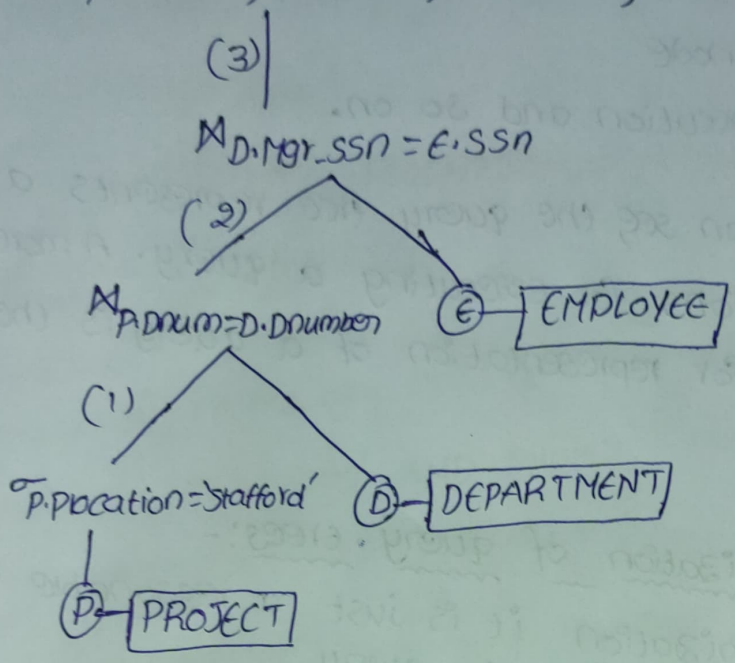
For every project located in 'Stafford', retrieve the project number, the controlling department number, and the department manager's last name, address, and birth date. This query is specified on the COMPANY relational schema and corresponds to the following relational algebra expression:

$$\Pi_{Pnumber, Dnum, Lname, Address, Bdate}(((\sigma_{Plocation='Stafford'}(PROJECT)) \bowtie_{Dnum=Dnumber} (DEPARTMENT)) \bowtie_{Mgr\_ssn=ssn} (EMPLOYEE))$$

This corresponds to the following SQL query:

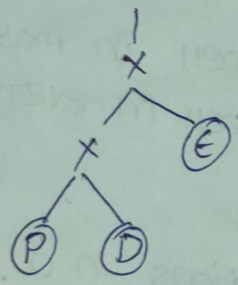
Eg:- SELECT P.pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
FROM PROJECT AS P, DEPARTMENT AS D, EMPLOYEE AS E
WHERE P.Dnum = D.Dnumber AND D.Mgr_ssn = E.ssn AND
P.Plocation = 'Stafford';

a) $\pi_{P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate}$

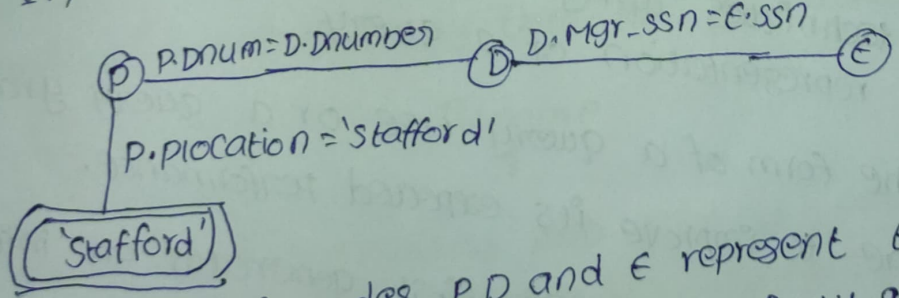


(b) $\pi_{P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate}$

$\sigma_{P.Dnum = D.Dnumber \text{ AND } D.Mgr\_ssn = E.Ssn \text{ AND } P.Plocation = 'Stafford'}$



c) [P, Pnumber, P.Dnum] [E.Lname, E.Address, E.Bdate]



The leaf nodes P, D and E represent the three relations PROJECT, DEPARTMENT, and EMPLOYEE, respectively and the internal tree nodes represent the relational algebra operations of the expression. When this query tree is executed, the node marked (3)

(1) must be available before we can begin executing operation
2, Similarly, node (2) Must begin executing and producing results before node
3, can start execution and so on.

As we can see the query tree represents a specific order of operations for executing a query. A more neutral data structure for representation of a query is the query graph notation.

Heuristic optimization of query trees:-

* Heuristic optimization it is just a reasonable efficient strategy for executing the query.
* A query typically has many possible execution strategies
* the process of choosing suitable one for processing a query is known as query optimization.
* A heuristic is a rule that works well in most cases but is not guaranted to work well in every possible cases.
* the rules typically reorder the operations in a query tree.
* optimization techniques apply heuristic rules to modify the internal representation of a query.
* Usually in the form of a query tree or a query graph data structure to improve its expected performance.
* the parse of a high-level query first generates an initial internal representation.
* Then optimized according to heuristic rules.

Ex:-

EMPLOYEE(FNAME, MNAME, LNAME, SSN, BDATE, ADDRESS, GENDER, SALARY, SUPERSSN, DNO)

PROJECT(PNAME, PNUMBER, PLOCATION, DNUM)

WORKS_ON(ESSN, PNO, HOURS)

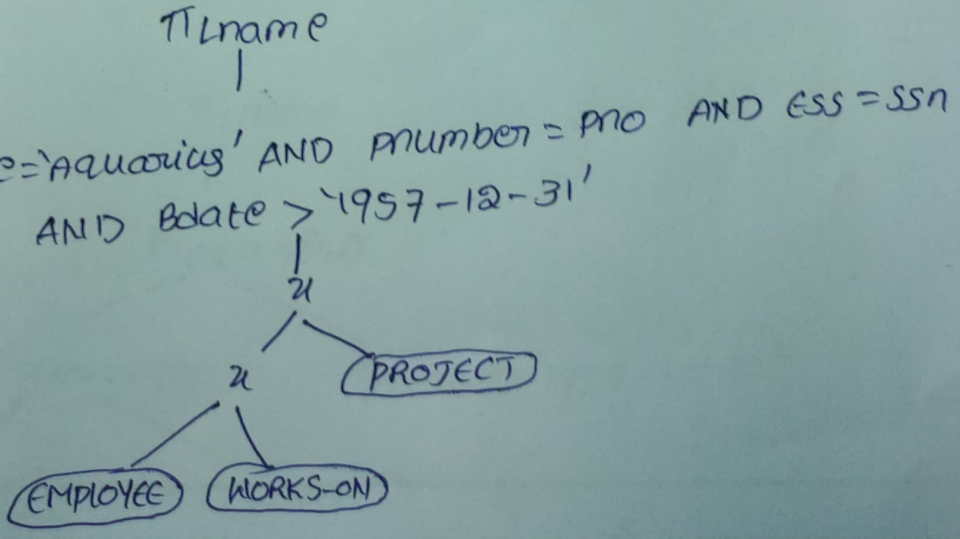
```
SELECT LNAME
FROM EMPLOYEE, WORKS_ON, PROJECT
WHERE DNAME='AQUARIUS' AND PNUMBER=PNO AND
  ESSN=SSN AND BDATE>'1957-12-31';
```

Steps in converting a query tree during heuristic optimization

1. Initial (canonical) query tree for SQL query.
2. Moving SELECT operations down the query tree.
3. Applying the more restrictive SELECT operation first
4. Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.
5. Moving PROJECT operations down the query tree.

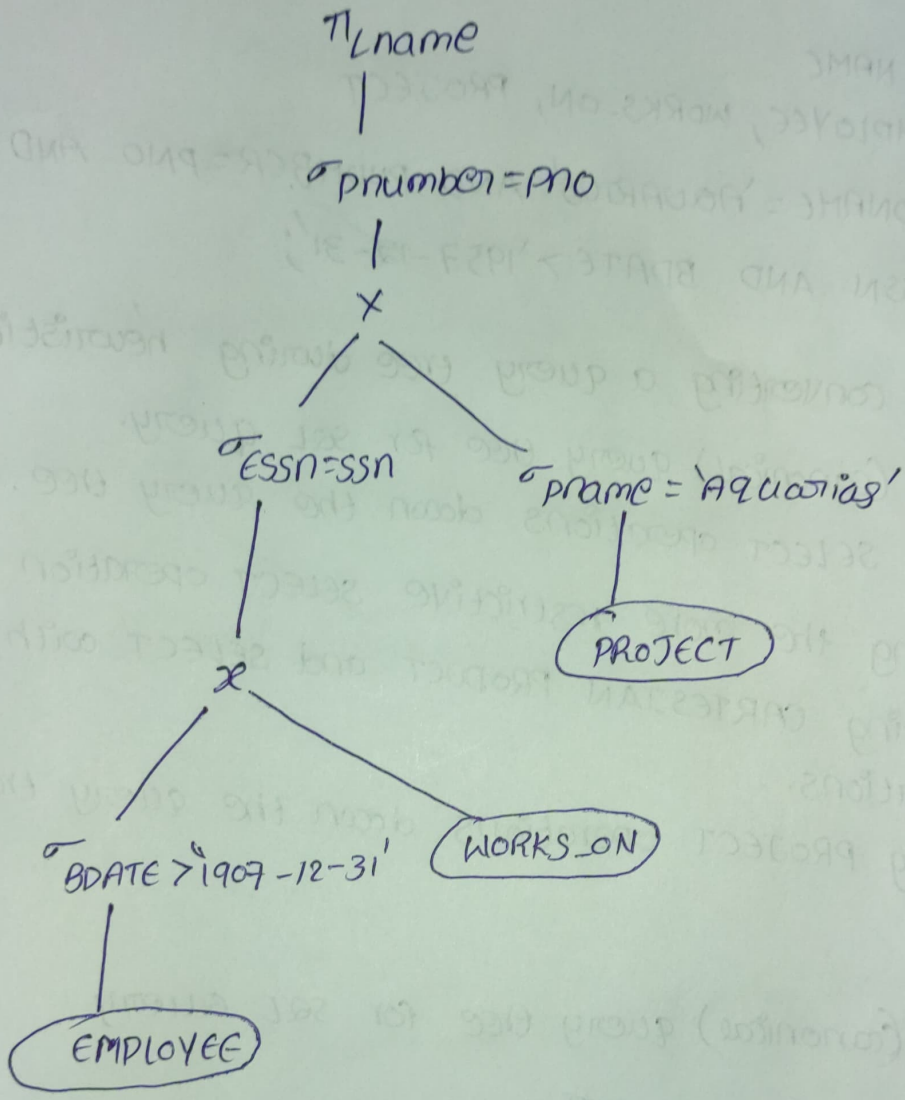
Step-1:-

Initial (canonical) query tree for SQL query



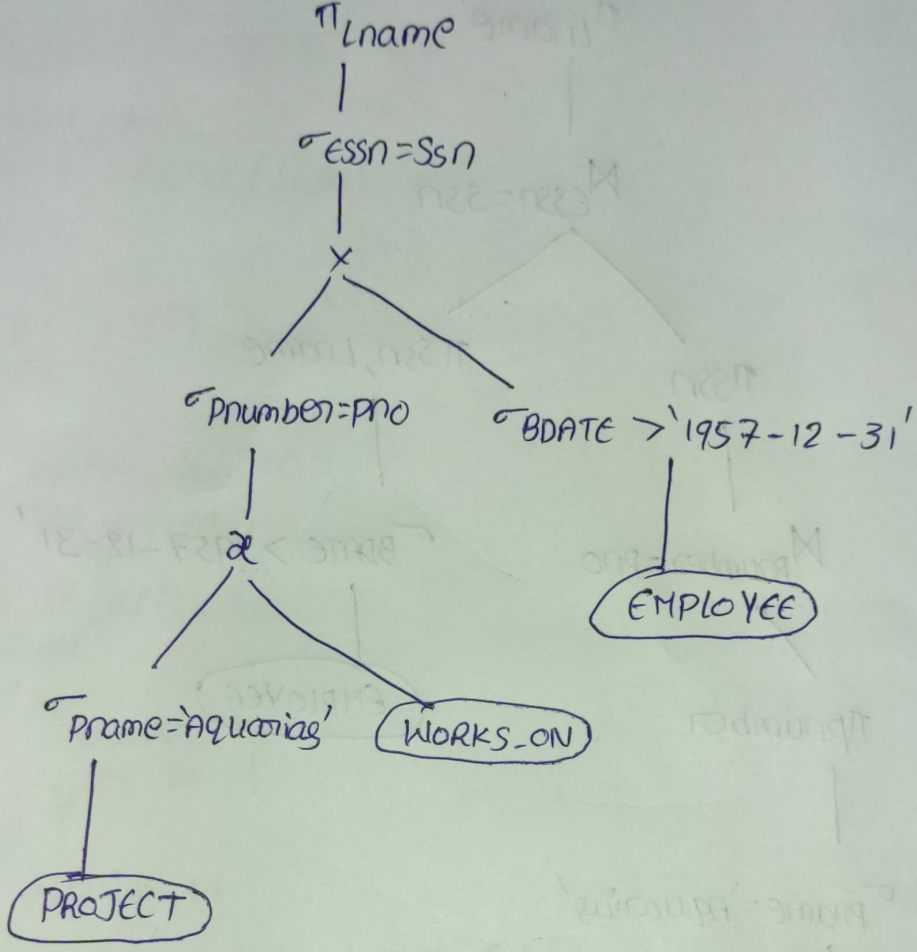
Step-2:-

Moving SELECT operation down the query tree



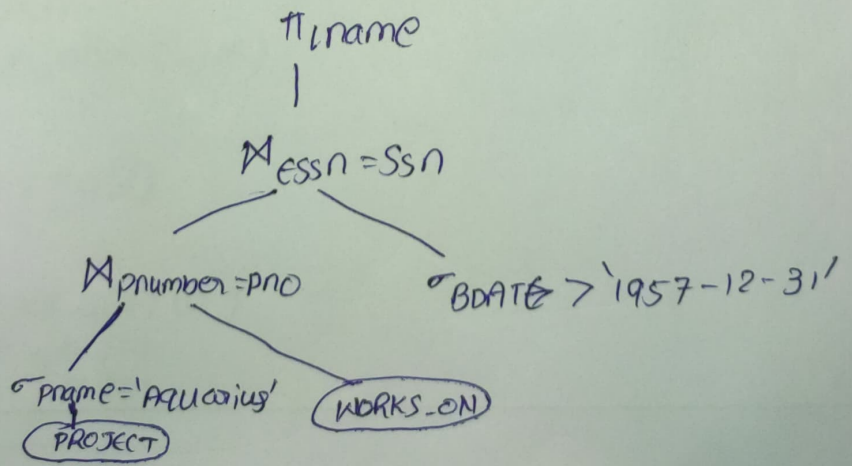
Step-3:-

Applying the more restrictive SELECT operation first



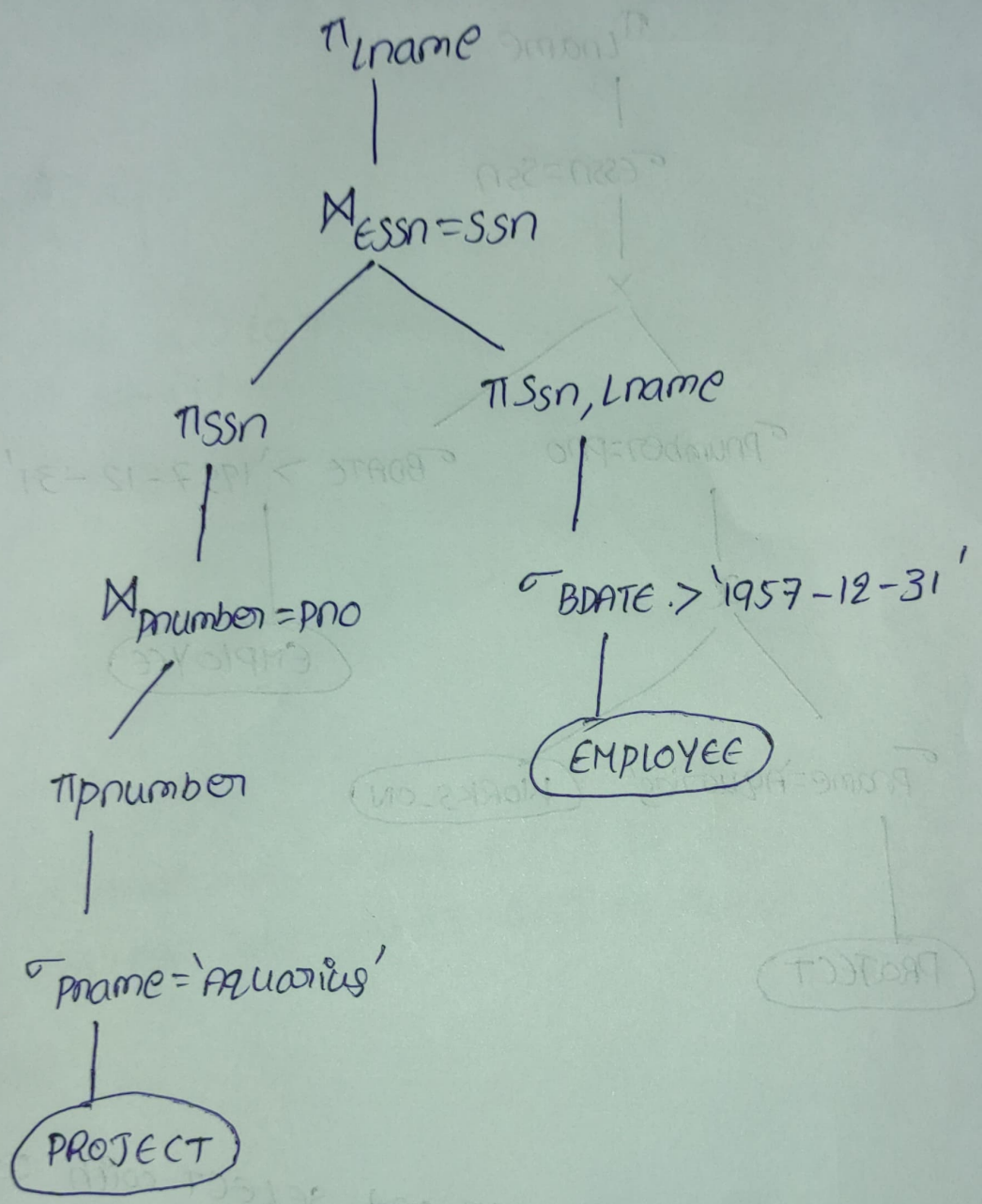
Step-4:-

Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.



Step-5:-

Moving PROJECT operations down the query tree



5. Write the Algorithms for implementing the union, intersection and set difference?

Union

```
T ← R ∪ S
set i ← 1, j ← 1
while (i ≤ n) and (j ≤ m)
do
{
  if R(i) > S(j)
  then
   {
      output S(j) to T
      set j ← j+1
   }
  else if R(i) < S(j)
    then
      {
        output R(j) to T
        set i ← i+1
      }
  else
      set j ← j+1
}
if (i ≤ n) then add tuples R(i) to R(n) to T
if (i ≤ m) then add tuples S(i) to S(m) to T
```

Intersection

```
T ← R ∩ S
set i ← 1, j ← 1
while (i ≤ n) and (j ≤ m)
do
{
  if R(j) == S(j)
  then
   {
      output S(j) to T
      set j ← j+1
      set i ← i+1
   }
```

```
    else
     {
       if R(i)>S(j) then set j←j+1
       else then set i←i+1
     }
}
```

Or

Set Difference

```
T←R-S
set i←1,j←1
while (i≤n) and (j≤m)
do
{
   if R(i)>S(j)
   then
      set j←j+1
   else if R(i)<S(j)
   then
     {
       output R(i) to T
       set i←i+1
     }
   else
     {
       set i←i+1
       set j←j+1
     }
if (i≤n) then add tuples R(i) to R(n) to T
```