

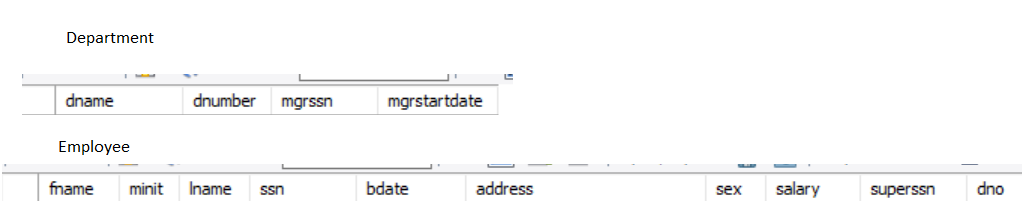
CST325

Assignment No. 3



**Task 1: Relational Algebra *(10 points)***

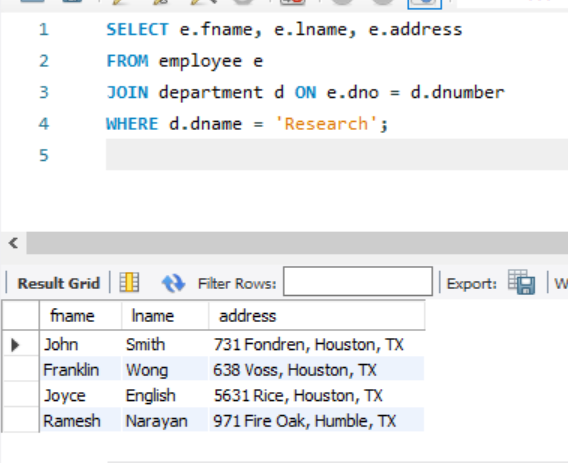
1. Relational Algebra Expressions
   1. Retrieve the name and address of employees who work for the 'Research' department.



Algebraic Expression:

|  |
| --- |
| Π fname, lname, address (σ dname=’Research’ (department ⨝ employee)) |

Query execution:



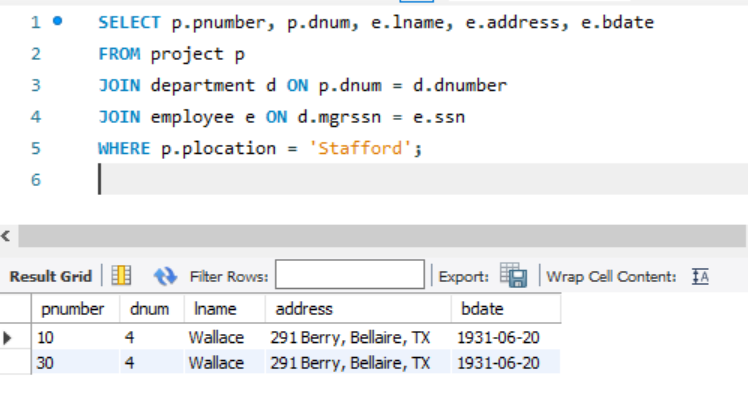
Explanation of the expression:   
  
Select (σ) tuples from the department database with 'Research' as the department name (dname).   
Create a natural join (⨝) between the employee and department tables.   
Project (π) the resulting tuple's address (address), last name (lname), and first name (fname).

* 1. For every project located in ‘Stafford’, list the project number, the controlling department number, and the department manager’s last name, address, and birth date.

*Algebraic Expression:*

|  |
| --- |
| Π pnumber, dnum, lname, bdate (σ plocation=’Stafford’ (project ⨝ department ⨝ employee)) |

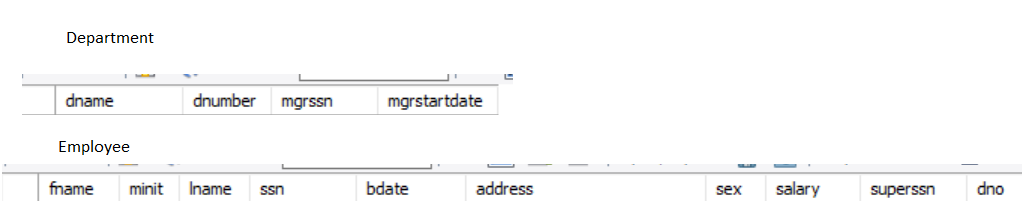
Query Execution:



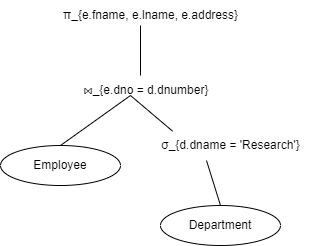
Explanation of Expression:   
  
Select (σ) tuples from the project table whose project location (plocation) is 'Stafford'.   
Create a natural join (⨝) between the department and project tables.   
Use a natural join (⨝) to connect the resulting and employee tables.   
The generated tuples include the project number (pnumber), controlling department number (dnum), manager's last name (lname), address (address), and birthdate (bdate).

**Task 2: Query Processing and Optimization *(30 points)***

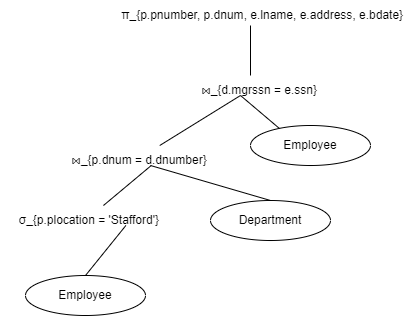
1. Query Trees:



* 1. Retrieve the name and address of employees who work for the 'Research' department.



* 1. For every project located in ‘Stafford’, list the project number, the controlling department number, and the department manager’s last name, address, and birth date.



1. Identify the selection, projection, and join operations:

Selection:

|  |  |
| --- | --- |
| Query 1 | Query 2 |
| * 𝜎d.dname = ‘Research’ | * 𝜎p.plocation = ‘Stafford’ |

Projection:

|  |  |
| --- | --- |
| Query 1 | Query 2 |
| * ​π e.fname,e.lname,e.address | * πp.pnumber,p.dnum,e.lname,e.address,e.bdate |

Join:

|  |  |
| --- | --- |
| Query 1 | Query 2 |
| * employee⋈ e.dno= | * project⋈ p.dnum= * project⋈ p.dnum= ⋈ |

1. Data distribution:
   * Employee table has 10,000 records.
   * Department table has 100 records.

Given Query:

|  |
| --- |
| SELECT E.Name, E.Address  FROM Employee E, Department D  WHERE E.DepartmentID = D.DepartmentID AND D.Name = 'Research'; |

Assumptions:

* Across the department distribution of employees is uniform.
* There are proportional number of employees in Research department.

Steps to Estimate Size of the Output:

* Calculation of selectivity factor for Research Department.
* Estimation of the number of employees in the Research Department.

So, the estimated size of output is 100 records

1. Algorithms for implementation of selection operations in query processing:

1. Linear Search:

* Scan each record in the table and check if it meets the selection criteria.
* Suitable for small tables or queries with low selectivity.

2. Binary Search:

* Requires the table to be sorted on the attribute used in the selection condition.
* Efficient for equality selections on indexed attributes.

3. Index-Based Search:

* Use an index to locate records that satisfy the selection condition.
* B-tree indexes are effective for range queries.
* Hash indexes are efficient for equality searches.

4. Hash-Based Selection:

* Use a hash function to partition the records based on the selection attribute.
* Fast and efficient for equality selections.

5. Bitmap Indexes

* Use bitmap indexing for attributes with low cardinality.
* Efficient for selections involving multiple attributes using bitwise operations.

6. Composite Indexes:

* Use composite indexes for selection conditions involving multiple attributes.
* Reduces the need for multiple index lookups.

Each algorithm has its strengths and is chosen based on factors like the size of the dataset, the nature of the selection criteria, and the availability of indexes.

**Task 3: Algorithms for External Sorting *(25 points)***

1. External Merge Sort Algorithm

External Sorting: A class of sorting algorithms capable of handling enormous volumes of data is known as "external sorting." When the data to be sorted must live in slower external memory rather than the main memory of a computing device (typically RAM), external sorting is necessary (usually a hard drive).

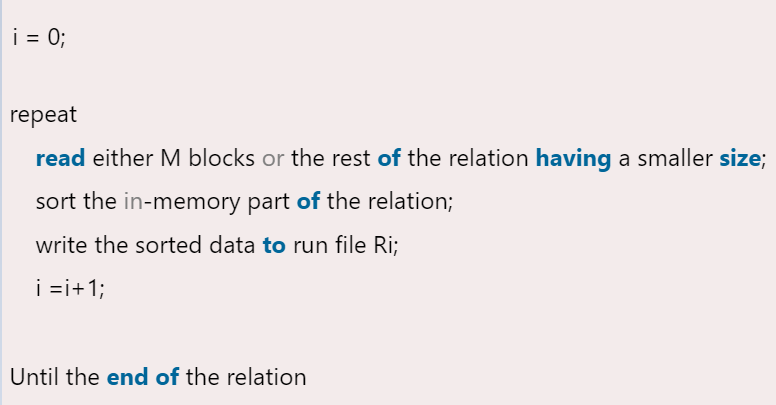
In addition to producing a sequential output, sorting is used to meet the requirements of different database algorithms. The sorting method in query processing is used to efficiently accomplish several relational operations, such joins, etc. However, the system must receive a sorted input value. Building an index on the sort key and using it to read the relation in sorted order is necessary for sorting any relation. But rather than physically sorting the relation, we do so rationally by utilising an index. Sorting is therefore done for the following cases:   
  
Case 1: Relationships that are smaller or medium-sized compared to primary memory.   
  
Case 2: Relationships with sizes greater than those stored in memory.

In Case 1: The size of the main memory is not exceeded by the small or medium size relations. Thus, they will fit into our memory. Therefore, we may accomplish this by using common sorting techniques like quicksort, merge sort, etc.

In Case 2: The conventional algorithms are ineffective. Thus we apply the External Sort-Merge algorithm for such relations whose size is greater than the memory limit.   
The arrangement of relationships whose sizes are too big for the memory to hold them. External sorting is the term for this kind of sorting. Therefore, the best technique for external sorting is the external-sort merge.

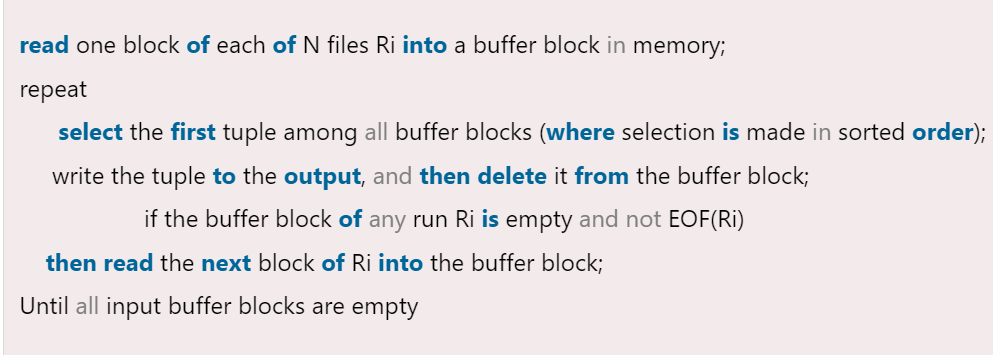
External Merge Sort Algorithm:

The number of disc blocks that are available for sorting in the main memory buffer is indicated by the algorithm's variable M.   
Step 1: We start by making several sorting runs. Arrange every one of them. There aren't many records of the relation in these runs.



Sorting the disc blocks is what we can see we are doing in Stage 1. Step 1 should be finished before moving on to Step 2.

Step 2: The runs are combined in Step 2. Assume that N, or the total number of runs, is fewer than M. As a result, we may assign one block to each run and yet have room for one output block. We carry out the procedure as follows:



Stage 2 will result in an output that is sorted relations. Next, in order to reduce the amount of disk-write operations, the output file is buffered. This procedure is called an N-way merge since it combines N runs.   
In the event that the relation's size surpasses the memory capacity, Stage 1 will generate M or more runs. Additionally, when processing Stage 2, it is not possible to allocate a single block for each run. In this scenario, the merging procedure is carried out in stages. Since M-1 input buffer blocks have enough memory, M-1 runs can be readily used as the input for each merge. Thus, the first stage operates as follows:

In order to obtain a single run for the following M-1, it combines the first one.   
It also combines the subsequent M-1 runs. Until all of the initial runs are processed, this phase is repeated. In this case, the M-1 value of the number of runs is lower. However, we must do another pass if this lowered value is higher than or equal to M. The runs produced by the previous pass will be the input for this new pass.   
Every pass's task is to cut down on the number of runs by M-1. Until the number of runs is either fewer than or equal to M, this task is repeated as many times as necessary.   
The sorted output is therefore produced in a last step.

Importance of External Merge Sort Algorithm:

Managing Huge Databases:   
  
The use of external merge sort is essential for managing datasets larger than what the main memory can hold. Because databases frequently handle terabytes or petabytes of data, sorting in-memory is not practical.  
Effective Disc I/O:  
  
By reading and writing huge amounts of data in a sequential manner, the technique reduces the amount of disc I/O operations. The procedure is more efficient because sequential I/O operations are quicker than random I/O operations.  
Scalability  
  
With several merging passes, external merge sort can expand to very big datasets. It is made to function with data that is saved on disc, which is necessary for contemporary database systems that handle enormous volumes of data.  
  
Foundation for Additional Activities  
  
Sorting huge datasets is a necessary step in many database processes, including indexing, join operations, and query optimisation. External merge sort efficiently sorts the data, laying the groundwork for these processes.  
The ability to tolerate faults  
  
Partial failures can be handled patiently by the algorithm. Only that portion of a temporary sorted file—not the complete dataset—needs to be reprocessed in the event of corruption.

Conclusion:

An essential algorithm in database systems for effectively sorting big datasets that can't fit in main memory is called external merge sort. It is a crucial tool for contemporary databases because of its capacity to manage enormous volumes of data and reduce disc I/O operations.

The file is initially split up into portions that will fit in the buffer.

As a sorted run, each chunk is read into memory, sorted, and then written back to disc.  
The first 64-block sorting runs will be of this size since the buffer space is 64 blocks.

The number of initial sorted runs:

Merge Phase:

Multiple sorted runs are combined with each pass of the merge step.  
The buffer space minus one for output places a restriction on the number of sorted runs that can be merged in a single pass. 63 runs can therefore be combined at once (64 - 1 = 63).

Calculate Number of Passes:

Continuously combine the runs until there is just one run left in order to determine the number of passes needed to combine all the runs into a single sorted file.  
  
The following formula for the number of merge passes can be used to explain this process:  
  
No. of Passes = []  
Where k is the number of merge able runs in a single pass and N is the number of starting sorted runs.  
  
N=64, k=63

No. of Passes = []

Logarithm Calculation:

Using properties of Algorithm:

[] =

[] ≈ 1.8

[] ≈ 1.799

So,

[] = = 2

**Task 4: Implementing the JOIN Operation *(35 points)***

1. Strategies for implementing the Join operation:

There are several strategies for implementing the join operation in relational database systems

**1. Nested-Loop Join**

Basic Nested-Loop Join: For each tuple in relation R, scan the entire relation S and check for matching tuples.

Block Nested-Loop Join: Instead of considering one tuple at a time, the algorithm reads a block of tuples from R and compares it with every block of S.

**2. Index Nested-Loop Join**

Uses an index on the join attribute of one of the relations (typically the smaller one). For each tuple in the other relation, use the index to find matching tuples efficiently.

**3. Sort-Merge Join**

Both relations are sorted on the join attribute, and then the sorted relations are merged, producing the join result.

**4. Hash Join**

Basic Hash Join: Partitions both relations into smaller sub-relations using a hash function on the join attribute, and then joins each pair of corresponding partitions.

**Grace Hash Join**: Divides the relations into partitions that fit into memory, processes partitions in stages.

**Hybrid Hash Join:** Uses the hash table more effectively by building it partially in memory.

1. Compare their performance based on their time complexity, advantages and disadvantages

**1. Nested-Loop Join**

**Time Complexity: O (|R| \* |S|)**

**Basic Nested-Loop Join:** Scans the entire relation

**Block Nested-Loop Join:** Considers blocks of tuples rather than individual tuples to reduce the number of I/O operations.

|  |  |
| --- | --- |
| Advantages | Disadvantages |
| Simple to implement. | Very slow for large datasets. |
| Does not require any pre-processing (like sorting or indexing). | High I/O cost due to repeated scanning of the entire inner relation. |

**2. Index Nested-Loop Join**

**Time Complexity:** O (|R| +|R| \* cost of Index lookup)

Uses an index on the join attribute of one of the relations to speed up the lookup process.

|  |  |
| --- | --- |
| Advantages | Disadvantages |
| Efficient if an appropriate index exists. | Requires an index on the join attribute, which might not always be available. |
| Can handle large relations efficiently if indexed. | Index maintenance can be costly. |

Performance heavily depends on the efficiency of the index lookup.

**3. Sort-Merge Join**

**Time Complexity:** O (|R| log |R| + |S| log |S|+ |R| + |S|)

Both relations are sorted on the join attribute and then merged.

|  |  |
| --- | --- |
| Advantages | Disadvantages |
| Efficient for large datasets if relations are already sorted. | Requires sorting if the data is not already sorted, which can be costly. |
| Performs well with large datasets. | Significant disk I/O if the datasets are large and not sorted. |
| Particularly good for equi-joins. |  |

**4. Hash Join**

**Time Complexity:** O (|R|+ |R| )

Partitions both relations into smaller sub-relations using a hash function on the join attribute, then joins each pair of corresponding partitions.

|  |  |
| --- | --- |
| Advantages | Disadvantages |
| Very efficient for large datasets. | Memory-intensive, requires sufficient memory to hold the hash table. |
| No need for sorting or indexing. | Performance can degrade with poor hash functions or hash collisions. |
| Performs well in memory with appropriate hash functions. | Potential issues with overflow if the partitions do not fit in memory. |

**Performance Comparison:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Join Strategy** | Time Complexity | Advantages | Disadvantages |
| **Nested loop join** | (O( | R | \times |
| **Index nested loop** | (O( | R | + |
| **Sort Merge Join** | (O( | R | \log |
| **Hash Join** | (O( | R | + |

1. Consider these relations with the following properties:  
   r(A, B, C) s(C, D, E)  
   30,000 tuples 60,000 tuples  
   25 tuples fit on 1 block 30 tuples fit on 1 block

**No. of blocks**

**Relation r**

No. of blocks for r = [

**Relation s**

No. of blocks for s = [

* 1. Estimate the number of disk block accesses required for a natural join of r and s using a nested-loop join if s is used as the outer relation.

When s is the outer relation and r is inner relation:

For each block of s all blocks of r are needed to be read.

Total disk accesses=Number of blocks of s+ (Number of blocks of s × Number of blocks of r)

= 2000 + (2000 \* 1200)

= 2000 + 2400000

= 2, 402, 000 disk access

* 1. Estimate the number of disk block accesses required for a natural join of r and s using a nested-loop join if r is used as the outer relation.

When s is the inner relation and r is outer relation:

For each block of r all blocks of s are needed to be read.

Total disk accesses=Number of blocks of r+ (Number of blocks of r × Number of blocks of s)

= 1200 + (1200 \* 2000)

= 1200 + 2400000

= 2, 401, 000 disk access

* 1. Conclude

**Disk accesses when s is the outer relation: 2,402,000**

**Disk accesses when r is the outer relation: 2,401,200**

The external relation is s. However, the enormous number of disc accesses that arise from both methods highlights the inefficiency of the nested-loop join for big relations. Using more complex join algorithms, such as sort-merge join or hash join, which are better suited for handling huge datasets with noticeably fewer disc accesses, can help reduce this inefficiency.

**Task 5: Query Trees and Heuristics for Query Optimization *(25 points)***

1. Query is given as:

SELECT E.Lname

FROM EMPLOYEE E, WORKS\_ON W, PROJECT P

WHERE P.Pname = ‘Aquarius’ AND P.Pnumber = W.Pno AND   
E.Ssn=W.Ssn AND E.Bdate = ‘1957-12-31’;

**Initial Query Tree**

**Initial Non-Optimized Query Tree:**

Join all three relations without any filtering.

Apply the selections and projections afterwards.

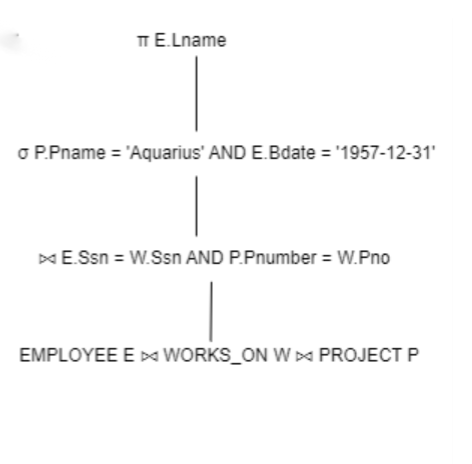
**Steps for creating the initial query tree:**

Join EMPLOYEE (E), WORKS\_ON (W), and PROJECT (P).

Apply selection conditions.

Apply projection.

Non Optimized Query Tree:



**Optimized Query Tree Using Heuristic Rules**

**Push down selections:** Apply selection operations as early as possible.

**Perform projections early**: Remove unused attributes early.

Reorder joins: Join the smallest relations first or those that reduce the size of intermediate results the most.

Step-by-Step Optimization

**Step 1: Apply Selections Early:**

* Apply σ P.Pname = 'Aquarius' to the PROJECT relation.
* Apply σ E.Bdate = '1957-12-31' to the EMPLOYEE relation.

**Step 2: Apply Projections Early:**

Project only the necessary columns from each relation as early as possible.

**Step 3: Reorder Joins:**

Join WORKS\_ON with the filtered PROJECT first since WORKS\_ON likely has fewer records than EMPLOYEE.

Optimized Query Tree:

**I Apply selections to PROJECT and EMPLOYEE:**

σ P.Pname = 'Aquarius' (PROJECT P)

σ E.Bdate = '1957-12-31' (EMPLOYEE E)

**II Project only the necessary attributes:**

π P.Pnumber (σ P.Pname = 'Aquarius' (PROJECT P))

π E.Ssn, E.Lname (σ E.Bdate = '1957-12-31' (EMPLOYEE E))

**III Join WORKS\_ON with the filtered PROJECT:**

WORKS\_ON W ⨝ W.Pno = P.Pnumber (π P.Pnumber (σ P.Pname = 'Aquarius' (PROJECT P)))

**IV Join the result with the filtered EMPLOYEE:**

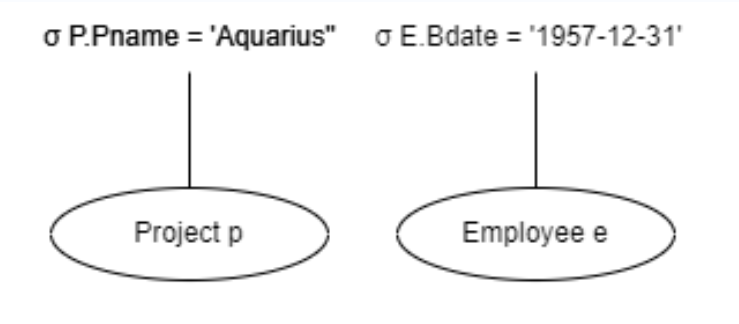
(π E.Ssn, E.Lname (σ E.Bdate = '1957-12-31' (EMPLOYEE E))) ⨝ E.Ssn = W.Ssn (WORKS\_ON W ⨝ W.Pno = P.Pnumber (π P.Pnumber (σ P.Pname = 'Aquarius' (PROJECT P))))

**V Finally, project the required attribute:**

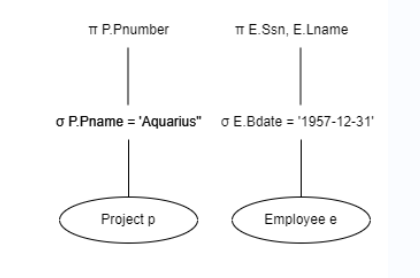
π E.Lname ((π E.Ssn, E.Lname (σ E.Bdate = '1957-12-31' (EMPLOYEE E))) ⨝ E.Ssn = W.Ssn (WORKS\_ON W ⨝ W.Pno = P.Pnumber (π P.Pnumber (σ P.Pname = 'Aquarius' (PROJECT P)))))

Optimized Query Tree Visualization:

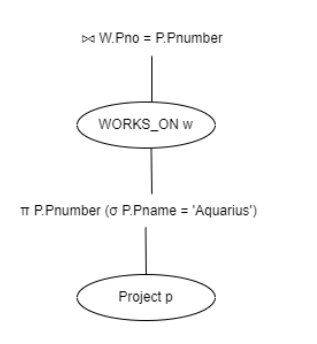
Apply Selection



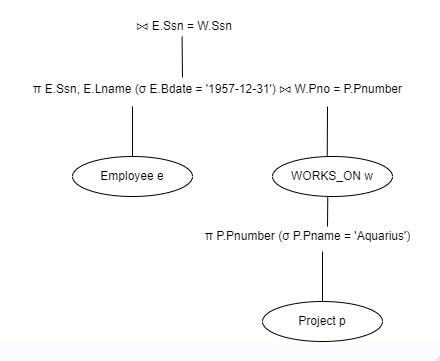
Apply Projection



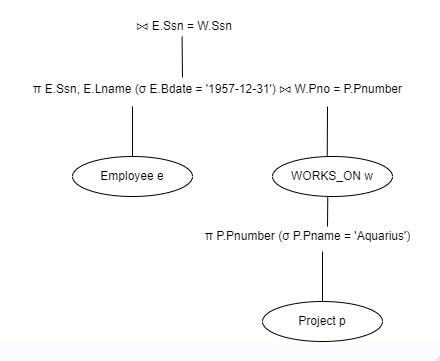
Join WORKS\_ON with Filtered projects



Join the result with filtered Employee



Final Projection



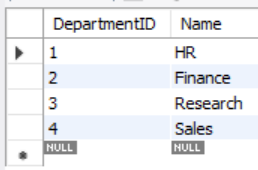
**Task 6: Query Execution Plans *(25 points)***

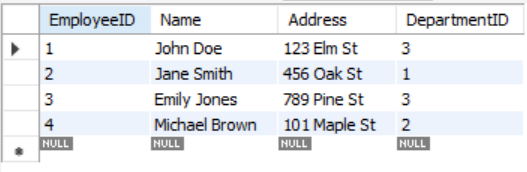
1: Setting Up environment:

Open the XAMP CP and start the MySQL service.

Open the browser and go to <http://localhost/phpmyadmin>.

2: Create Database and the Tables





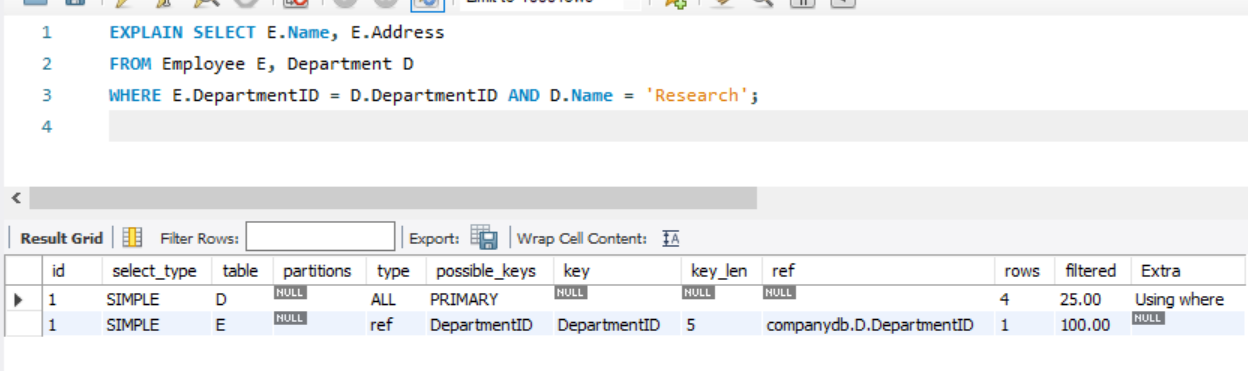
3: Running Query, Generating and Interpreting Execution Plan:

Explain statement tells us the execution Plan:

SELECT E.Name, E.Address

FROM Employee E, Department D

WHERE E.DepartmentID = D.DepartmentID AND D.Name = 'Research';



Interpretation

First Step:

Table: Department (D)

Type: ref - A reference to an indexed column.

Key: Name - MySQL uses the Name index.

Rows: 1 - MySQL expects to find 1 row in the Department table with Name = 'Research'.

Extra: Using index - MySQL uses the index to retrieve the data.

Second Step:

Table: Employee (E)

Type: ref - A reference to an indexed column.

Key: DepartmentID - MySQL uses the DepartmentID index.

Rows: 2 - MySQL expects to examine 2 rows in the Employee table.

Extra: NULL - No extra information.