**Geographic Information System (GIS) Analysis of coffee shop**

**Team Members:**

1. **Sandeep Malyala (U01935264)** sm86925n@pace.edu
2. **Pravallika Challa (U01939539)** pc94436n@pace.edu
3. **Chetan Kumar Reddy Vaddi (U01883016)** CV33511N@pace.edu
4. **Ramadugu Bhavitha (U01915121)** br31457n@pace.edu
5. **Sai Kiran Nunna (U01920311)** sn01516n@pace.edu

**Question1 Answer:**

External merge sort involves dividing the file into chunks that fit into memory, sorting those chunks in memory using a sorting algorithm like quicksort, and then merging the sorted chunks back together. The number of passes needed for external merge sort can be calculated using the formula:

Passes=⌈logB−1​N⌉

Where:

* BB is the number of buffers.
* NN is the number of pages in the file.

In this case, N=1,000,000N=1,000,000 pages and B=6B=6 buffers.

Passes=⌈log⁡51,000,000⌉Passes=⌈log5​1,000,000⌉

Let's calculate this:

Passes=⌈log10​1,000,000/log105​⌉

Using a calculator, you can find:

Passes≈⌈8.861⌉

So, you would need 9 passes to sort the file with N=1,000,000N=1,000,000 pages using external merge sort with 6 buffers.

**Question2 Answer):**

To find all keys between 9 and 19 in the given B+tree, we need to chase the following pointers:

* **Parent-to-child pointer:** From the root node to the leaf node that contains the keys between 9 and 19.
* **Sibling-to-sibling pointer:** To all the leaf nodes to the right of the leaf node that contains the keys between 9 and 19, until we reach a leaf node that contains a key greater than 19.

**Total number of pointers:** 1 parent-to-child pointer + (number of leaf nodes to the right of the leaf node that contains the keys between 9 and 19) sibling-to-sibling pointers

In the given B+tree, the leaf node that contains the keys between 9 and 19 is the second leaf node from the left. To the right of this leaf node, there are two more leaf nodes. Therefore, we need to chase 1 parent-to-child pointer and 2 sibling-to-sibling pointers.

**Total number of pointers:** 1 + 2 = 3

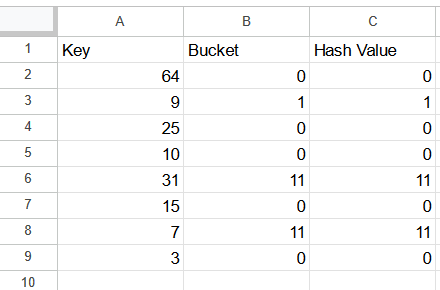
Therefore, we need to chase **3 pointers** to find all keys between 9 and 19 in the given B+tree.

**Question3 Answer):**

The longest key less than 5 whose insertion will cause a split is **4**.

This is because the bucket with hash value 00 has already reached its maximum capacity of 4 entries. If we insert another key into this bucket, it will cause a split, creating a new bucket with hash value 01.

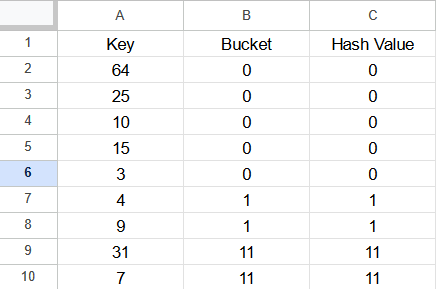
The following table shows the state of the hash table after each key is inserted:

****

**Insertion of 4:**

Since the bucket with hash value 00 is already full, inserting the key 4 will cause a split. A new bucket with hash value 01 will be created, and the keys 4 and 9 will be moved to the new bucket.

State of the hash table after insertion of 4:



Therefore, the longest key less than 5 whose insertion will cause a split is **4**.

**Question4 Answer):**

In a B+ tree, the order (d) is the maximum number of children each internal node can have. In a sparse B+ tree, most nodes are not filled, and the tree is designed to be more space-efficient.

In a B+ tree, all the keys are present in the leaf nodes, and each internal node (except the root) has between d/2 and dd children.

Let us consider the keys 1 through 20 inclusive in a sparse B+ tree of order d=2.

1. The leaves of the B+ tree will contain the keys 1 through 20.
2. The internal nodes will guide the search for a specific key.

B+ tree:

[10]

/ \

[5] [15]

/ \ / \

[2, 8] [12] [14, 18]

/ \ | / | \

[1, 3] [7] [11] [13] [17] [19, 20]

In this B+ tree, the internal nodes are represented by square brackets, and the keys are in the leaves. Each internal node has between d/2d/2 and dd children.

Now, let's count the total number of nodes:

1. Root node: 1
2. Internal nodes: 6
3. Leaf nodes: 10

So, the B+ tree has a total of 1+6+10=171+6+10=17 nodes.

**Question5 Answer):**

SQL query:

SELECT R.a

FROM R, S, U

WHERE R.b = S.b AND S.b = U.b AND U.e = 6;

This query retrieves values of attribute a from the relation R where there are matching values of b in R, S, and U, and the corresponding value in U for attribute e is equal to 6.

Now, let'sanalyse the two equivalent logical plans in relational algebra:

1. πa(σc=3(R⋈b=b(S)))πa​(σc=3​(R⋈b​=b(S)))
2. πa(R⋈b=bσc=3(S))πa​(R⋈b​=bσc=3​(S))

In Plan I, the selection (σc=3(R⋈b=b(S))σc=3​(R⋈b​=b(S))) is performed first, and then the projection (πaπa​) is applied. This plan retrieves tuples from the join of RR and SS where bb matches and then selects only those where cc is equal to 3.

In Plan II, the projection (πaπa​) is applied first, and then the selection (σc=3(S)σc=3​(S)) is performed. This plan retrieves all tuples from SS where cc is equal to 3 and then performs the join with RR based on matching values of bb.

Generally, the efficiency of a query plan depends on various factors, including the size of relations involved and the selectivity of conditions. In many cases, the query optimizer of a relational database management system (RDBMS) can choose the most efficient plan based on statistics about the tables.

However, considering the given plans without specific statistics:

* Plan I might be more efficient if the selection (σc=3(R⋈b=b(S))σc=3​(R⋈b​=b(S))) significantly reduces the number of tuples before the projection.
* Plan II might be more efficient if the projection (πa(R⋈b=bσc=3(S))πa​(R⋈b​=bσc=3​(S))) reduces the size of the join result significantly before projecting.

In practice, the efficiency can vary based on the data distribution and the specific characteristics of the tables. The database optimizer is typically responsible for choosing the most efficient execution plan based on these factors.

**Question6 Answer):**

The answer is false. In the vectorized processing model, the focus is on processing batches of data (vectors) rather than individual tuples. The idea is to take advantage of modern hardware architectures, such as SIMD (Single Instruction, Multiple Data) instructions, to process multiple data elements simultaneously.

In a vectorized processing model, operators are designed to work on entire batches of data at once, and this can lead to improved performance compared to row-wise processing. However, this does not necessarily imply that multi-threaded execution is required for operators that receive input from multiple children.

The decision to use multi-threading depends on various factors, including the architecture of the system, the characteristics of the workload, and the specific implementation of the database system. Some vectorized databases may use multi-threading to parallelize the processing of different batches or to concurrently process data from multiple sources, but it's not a strict requirement of the vectorized processing model itself.

Therefore, whether an operator requires multi-threaded execution depends on the specific implementation and optimization strategies chosen by the database system, and it's not an inherent characteristic of the vectorized processing model.

**Question7 Answer):**

Several strategies can be employed to optimize a Hash join algorithm, enhancing its efficiency and performance:

* **Careful Join Condition Selection:** The choice of join condition significantly impacts the hash join's performance. Selecting a join condition with high selectivity, meaning it produces a smaller number of matching tuples, reduces the overall processing burden.
* **Hash Function Selection:** Choosing an appropriate hash function is crucial for distributing join keys evenly across hash buckets, minimizing collisions and maximizing hash table utilization. Popular hash functions include Murmur Hash and MD5.
* **Bucket Size Optimization:** Adjusting the bucket size can influence the hash join's performance. Larger buckets can reduce the number of overflow buckets, while smaller buckets can minimize the risk of collisions.
* **Memory Allocation:** Allocating sufficient memory for the hash table is essential to avoid spilling data to disk, which can lead to significant performance degradation.
* **Partitioning Input Data:** Partitioning the input data before the hash join can improve performance by dividing the data into smaller chunks, allowing for more efficient processing.
* **Hybrid Hash Join:** Combining hash join with other join algorithms, such as nested loop or sort-merge join, can provide better performance under specific conditions.
* **Index Utilization:** Leveraging indexes on join columns can accelerate the hash join process by reducing the number of tuples that need to be considered.
* **Hardware Optimization:** Utilizing hardware features like SIMD instructions and vectorization can enhance hash join performance on modern CPUs.
* **Query Optimization:** Refining the SQL query to eliminate unnecessary subqueries or reduce the number of join operations can improve overall query performance, including hash joins.
* **Database Monitoring:** Monitoring database performance metrics, such as I/O throughput and CPU utilization, can help identify potential bottlenecks and guide further optimization efforts.

**Question8 Answer):**

To determine the cost of the query plan below, we need to consider the number of page I/Os required for each operation.

**Cost of Index Nested Loop Operation:**

The index nested loop operation will involve scanning the index pages of the Major table, retrieving matching tuples, and joining them with the Applicants table. The number of page I/Os for this operation can be estimated as follows:

Number of page I/Os = (Number of Major table pages) \* (Fraction of Major table pages that are index pages) \* (Two page I/Os per tuple)

In this case, there are 100 index pages for the Major table. If 50% of the Major table pages are index pages, the number of page I/Os for the index nested loop operation is:

Number of page I/Os = (100 pages) \* (0.5) \* (2 I/O/tuple) = 100 pages

**Cost of Sort-Merge Join Operation:**

The sort-merge join operation will involve sorting the two input relations (Applicants and the result of the index nested loop operation) and then merging them. The number of page I/Os for this operation can be estimated as follows:

Number of page I/Os = (Number of Applicants pages) \* (Number of pages in the result of the index nested loop operation) \* (Number of I/Os per page)

In this case, there are 100 pages for the Applicants table. Assuming that the result of the index nested loop operation has 500 pages, the number of page I/Os for the sort-merge join operation is:

Number of page I/Os = (100 pages) \* (500 pages) \* (4 I/O/page) = 200,000 pages

**Total Cost of Query Plan:**

The total cost of the query plan is the sum of the costs of the individual operations. In this case, the total cost is:

Total cost = 100 pages + 200,000 pages = 200,100 pages

Therefore, the cost of the query plan is **200,100 page I/Os**.

**Question9 Answer):**

A.) **High Cardinality Hash Collisions:**

When a high cardinality of distinct values hash to the same bucket (hash collisions), it can lead to poor performance in hash join algorithms. To rectify this issue, you can use a technique called **hash partitioning or hash clustering**. In this approach, you reorganize the data so that tuples with the same hash value are stored together, forming clusters. This helps in reducing the number of random I/O operations during the hash join.

The steps involved in hash partitioning are as follows:

1. **Choose a New Hash Function:**
   * Select a new hash function h2h2​ that distributes the tuples more evenly across the available buckets.
2. **Partition the Relations:**
   * Apply the new hash function h2h2​ to both tables R and S, creating new partitions based on the updated hash values.
3. **Reorganize the Data:**
   * Sort each partition separately based on the new hash values.
4. **Re-run the Join Algorithm:**
   * After reorganizing the data, you can re-run the hash join algorithm using the new hash function.

This approach helps reduce the impact of hash collisions and can improve the efficiency of hash join algorithms.

B.) **I/O Cost of Block Nested Loop Join:**

The Block Nested Loop Join is a simple nested loop join algorithm where the outer relation is scanned block by block, and for each block of the outer relation, the inner relation is scanned entirely.

The I/O cost of the Block Nested Loop Join can be calculated using the formula:

I/O cost=Number of blocks in outer relation (R)+Number of blocks in outer relation (R)×Number of blocks in inner relation (S)I/O cost=Number of blocks in outer relation (R)+Number of blocks in outer relation (R)×Number of blocks in inner relation (S)

Given the information provided:

* Number of blocks in R (BRBR​) = Number of pages in R (MM) = 2,400
* Number of blocks in S (BSBS​) = Number of pages in S (NN) = 1,200

Therefore, the I/O cost for the Block Nested Loop Join is:

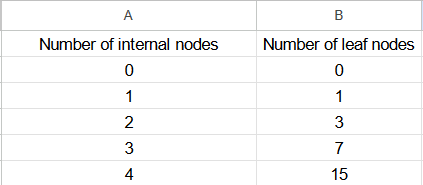
I/O cost=BR+BR×BSI/O cost=BR​+BR​×BS​ I/O cost=2,400+2,400×1,200I/O cost=2,400+2,400×1,200

You can calculate this to get the specific numerical value of the I/O cost.

**Question10 Answer):**

A full binary tree is a binary tree in which every node has exactly two child nodes. In a full binary tree with 2n internal nodes, there are 2^(n+1) - 1 leaf nodes.

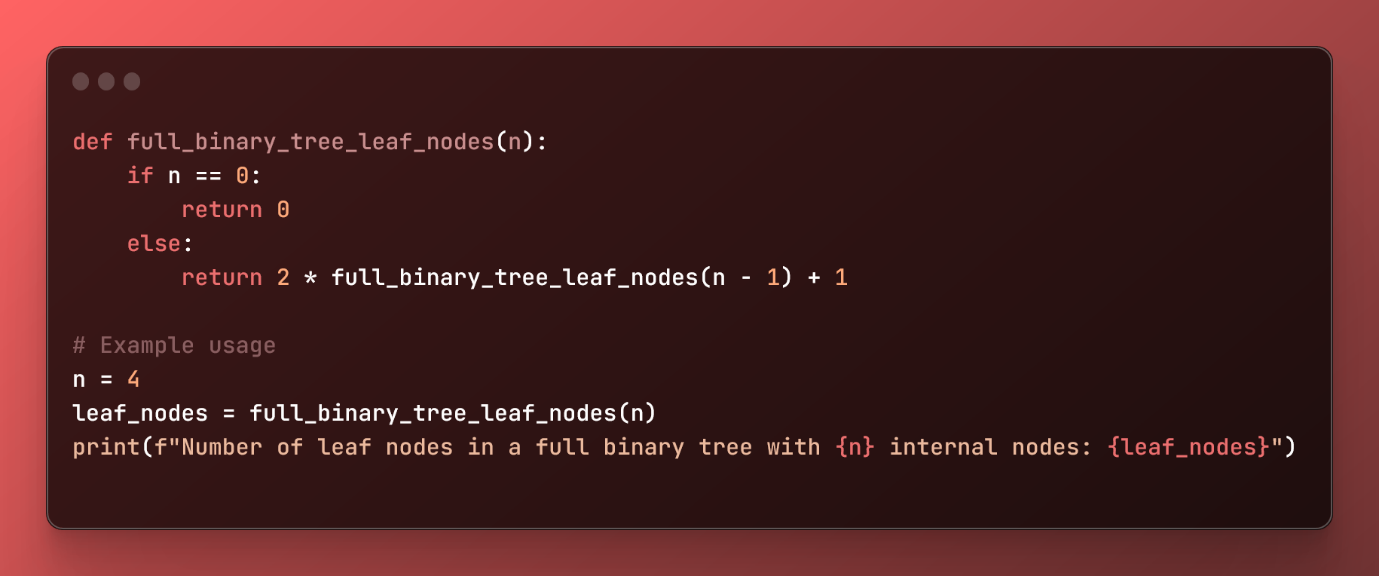
For example, a full binary tree with 4 internal nodes has 15 leaf nodes.



As the number of internal nodes increases, the number of leaf nodes increases exponentially.

Code:

Here’s the code using python

****

**Question11 Answer):**

To determine the number of leaf nodes in a complete binary tree, we can use the following formula:

Number of leaf nodes = 2^ (Number of internal nodes + 1) - 1

In this case, the number of internal nodes is 7, so the number of leaf nodes is:

Number of leaf nodes = 2^ (7 + 1) - 1 = 2^(8) - 1 = 127

Therefore, there are **127** leaf nodes in the complete binary tree.