**CS 623**

**PROJECT**

**Guidelines**

* This is a group project that you will have to do in a group of 3 students (maximum).
* Post your team group as well as the data source for your group’s data set in the spreadsheet.
* You will use PostgreSQL (rather than MySQL).
* Your code should also be on your individual GitHub. This is where I will check it. The code is developed as a team but available on the GitHub of participating students.
* You have two parts, the Practical and the Theory part. There is an extra 1 mark available for attempting the project.

**Deliverables**

* Code in GitHub(individually) and link to the github. I will check the code there.
* Submit a Video of < 3 minutes to show and explain your work • Screenshots of the code plus output.
* PDF/Word doc of solutions to theory questions

**Description**

Involves working with spatial data and utilizing the access methods and query executions and optimizations we would discuss in class. The project would involve writing SQL queries to retrieve information such as the locations of specific features, distances between points, and areas of interest. Using indexing, aggregate and join executors, sort+ limit executors, sorting, and top-N optimization.

**THEORY PART (24%)**

**You have 12 Theory questions, each with 2 marks.**

1. We have a file with a million pages (N = 1,000,000 pages), and we want to sort it using external merge sort. Assume the simplest algorithm, that is, no double buffering, no blocked I/O, and quicksort for in-memory sorting. Let B denote the number of buffers.

How many passes are needed to sort the file with N = 1,000,000 pages with 6 buffers?

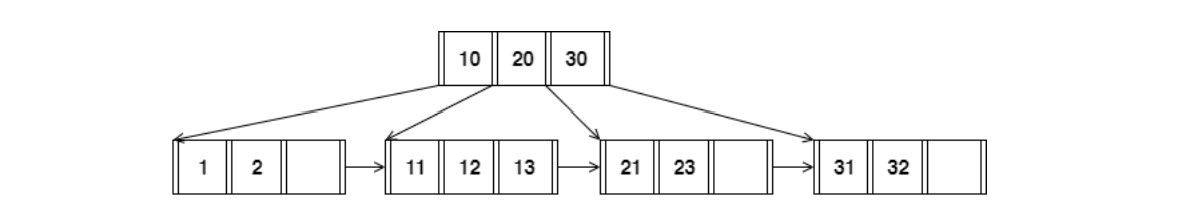
Ans) Substituting the given values, we get:

For N = 1,000,000, we get:

passes = ceil(log(1,000,000)) - 1 passes = ceil(20) - 1 passes = 20

So, 20 passes are needed to sort the file with N = 1,000,000 pages using 6 buffers.

1. Consider the following B+tree.



When answering the following question, be sure to follow the procedures described in class and in your textbook. You can make the following assumptions:

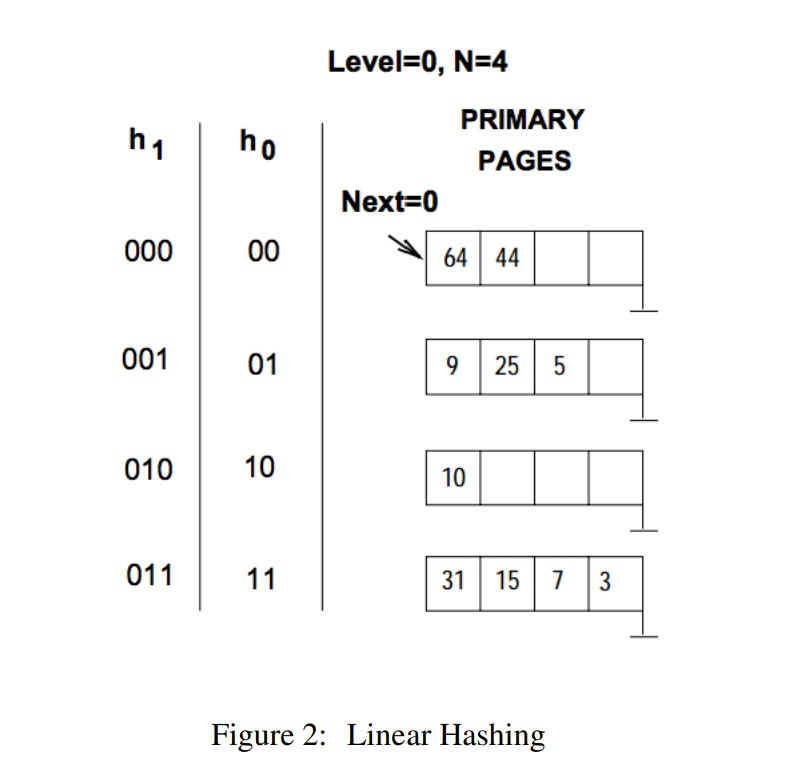
* + A left pointer in an internal node guide towards keys < than its corresponding key, while a right pointer guides towards keys ≥.
  + A leaf node underflows when the number of keys goes below [ (d−1)/ 2] e.
  + An internal node(root node) underflows when the number of pointers goes below d /2 .

How many pointers (parent-to-child and sibling-to-sibling) do you chase to find all keys between 9  and 19 ?

Ans) 1 in order to reach the leaf node. 2 more until it detects a key greater than 19

Thus, we require 3 pointers in total.

1. Answer the following questions for the hash table of Figure 2. Assume that a bucket split occurs whenever an overflow page is created. h0(x) takes the rightmost 2 bits of key x as the hash value, and h1(x) takes the rightmost 3 bits of key x as the hash value



What is the largest key less than 25 whose insertion will cause a split?

Ans) The biggest key, 23 out of less than 25, whose insertion will result in a split

1. Consider a sparse B+ tree of order d = 2 containing the keys 1 through 20 inclusive. How many nodes does the B+ tree have?

**Ans)** **There can be a maximum of two keys and three pointers per node in an order two B+ tree. Every node in a sparse B+ tree needs to have a minimum of keys and pointers in order to meet the requirements of the B+ tree. In this instance, every internal node needs to have 1 key and 2 pointers, while every leaf node needs to have 2 keys and no pointers.**

The picture's tree has three levels:

\* \*Level 1 (root): \* One node with two pointers and one key. \* Level 2 (intermediate): \* consists of 2 nodes with a total of 4 pointers and 2 keys. \*Level 3 (leaf):\* Five nodes totaling ten keys and no pointers.

Consequently, 1 + 2 + 5 is 8 the total number of nodes in the tree.

1. Consider the schema R(a,b), S(b,c), T(b,d), U(b,e).

Below is an SQL query on the schema:

SELECT R.a

FROM R, S,

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

For the following SQL query, I have given two equivalent logical plans in relational algebra such that one is likely to be more efficient than the other:

I. πa(σc=3(R b=b (S))) II. πa(Rb=b σc=3(S)))

Which plan is more efficient than the other?

Ans) Let's analyse the given logical plans in relational algebra and determine which one is likely to be more efficient.

The SQL query is:

SELECT R.a

FROM R, S, U

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

And the two equivalent logical plans are:

I. πa (σc=3(R⋈b=b S))

II. πa (R⋈b=b σc=3 (S))

Now, let's analyze each plan:

I. πa (σc=3(R⋈b=b S)):

* This plan performs the join between R and S first and then applies the selection on the result.
* It filters rows from the joined result where c is not equal to 3 and then projects the column a.
* This plan might involve joining more tuples before applying the selection.

II. πa (R⋈b=b σc=3 (S)):

* This plan first applies the selection on S by filtering out rows where c is not equal to 3.
* After that, it performs the join with R based on the condition b=b.
* This plan filters out rows from S early in the process, potentially resulting in a smaller intermediate result for the join.

In general, the second plan (II) is likely to be more efficient. By applying the selection early on S, fewer rows will be involved in the join operation, leading to a potentially smaller intermediate result set. This can result in a more efficient execution plan, especially if the condition c=3 is selective.

So, plan II (πa (R⋈b=b σc=3 (S))) is more likely to be more efficient than plan I.

1. In the vectorized processing model, each operator that receives input from multiple children requires multi-threaded execution to generate the Next() output tuples from each child. True or False? Explain your reason.

Ans) False. In the vectorized processing model, operators that receive input from multiple children may or may not require multi-threaded execution to generate the output tuples. The decision depends on the specific implementation and design choices made by the database management system (DBMS) or the query execution engine.

In the vectorized processing model, operators work on batches of tuples (vectors) rather than processing one tuple at a time. Multi-threading can be employed to enhance parallelism and improve the performance of query processing. However, whether an operator uses multi-threaded execution or not is a design decision, and it may vary based on the following factors:

1. **Hardware and System Configuration:** The decision to use multi-threading may depend on the availability of multiple CPU cores and the overall system architecture.
2. **Query Optimizer Strategy:** The query optimizer in the DBMS decides the execution plan for a given query. The chosen plan may or may not involve multi-threading based on cost estimation and available resources.
3. **Parallelism Requirements:** Some operators may benefit significantly from multi-threading when processing large datasets, while others may not see a substantial improvement.
4. **Concurrency Control and Isolation Requirements:** If there are concerns about maintaining consistency and isolation in a multi-threaded environment, the DBMS may choose to limit or avoid multi-threading.

Therefore, while multi-threading can be a valuable technique in vectorized processing for parallelism, it is not universally required for all operators that receive input from multiple children. The decision to use multi-threading is based on performance considerations, hardware capabilities, and the overall design philosophy of the DBMS.

1. How can you optimize a Hash join algorithm?

**Ans)** Hash join is a common join algorithm used in relational database systems to combine tables based on equality conditions. Optimizing a hash join algorithm involves improving its performance and efficiency. Here are some techniques to optimize a hash join:

1. **Memory Management:**

* Hash joins heavily depend on memory for building hash tables. Allocate an appropriate amount of memory to accommodate hash tables for both input relations. However, be mindful not to allocate too much memory, as it may lead to excessive paging.

1. **Partitioning:**

* Partitioning the input relations based on the join key can enhance the efficiency of the hash join. This allows each partition to fit into memory, reducing the memory footprint for each hash table and improving cache locality.

1. **Bucket Size:**

* Choose an optimal bucket size for the hash table. A good rule of thumb is to have a bucket size that fits into cache lines efficiently. Experiment with different bucket sizes to find the one that minimizes collisions and maximizes cache efficiency.

1. **Hash Function:**

* Use a high-quality hash function that distributes keys evenly across the hash table. Poorly chosen hash functions can lead to a high number of collisions, degrading performance.

1. **Build Phase Pipelining:**

* Pipeline the build phase of hash join to overlap CPU and I/O operations. While one thread is building the hash table, another can start probing it. This can reduce the overall time required to perform the hash join.

1. **Parallel Hash Join:**

* Implement parallel hash join to take advantage of multiple CPU cores. This involves parallelizing both the build and probe phases. However, be cautious of contention issues and ensure proper synchronization.

1. **Optimized Join Algorithms for Small Relations:**

* For small relations, consider using nested loop join or other algorithms that might be more efficient than building a hash table. Database optimizers often switch to alternative algorithms based on size estimates.

1. **Caching:**

* Leverage caching mechanisms to store frequently used data structures, such as hash tables. This can reduce the need to rebuild these structures when processing subsequent queries.

1. **Adaptive Hash Join:**

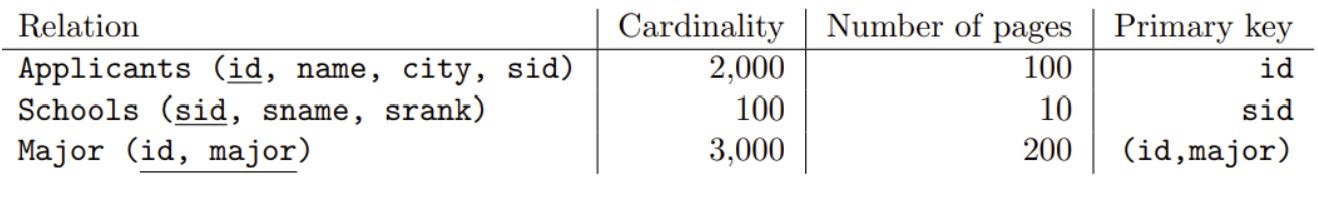
* Some database systems implement adaptive hash join algorithms that dynamically adjust their strategies based on the input data size and characteristics. This can lead to better performance across a range of scenarios.

1. **Statistics and Cost-Based Optimization:**

* Utilize statistics about the data distribution and size to guide the query optimizer. Cost-based optimization can help the database engine choose the most efficient join strategy based on current conditions.

It's important to note that the effectiveness of these optimization techniques can vary depending on the specific characteristics of the data, the query, and the hardware configuration. Experimentation and performance profiling are crucial to fine-tuning hash join algorithms for a particular database system and workload.

1. Consider the following SQL query that finds all applicants who want to major in CSE, live in Seattle, and go to a school ranked better than 10 (i.e., rank < 10).



SELECT A.name

FROM Applicants A, Schools S, Major M

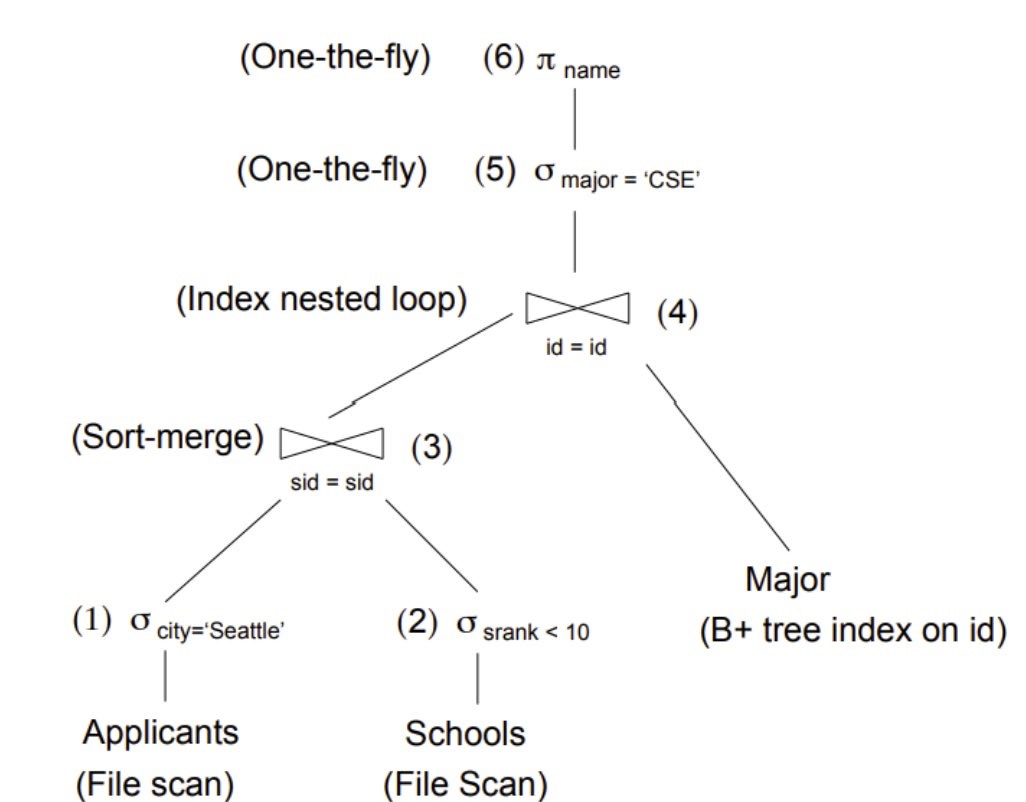
WHERE A.sid = S.sid AND A.id = M.id AND A.city = 'Seattle' AND S.rank

< 10 AND M.major = 'CSE'

Assuming:

* + - Each school has a unique rank number (srank value) between 1 and 100.
    - There are 20 different cities.
    - Applicants.sid is a foreign key that references Schools.sid.
    - Major.id is a foreign key that references Applicants.id.
    - There is an unclustered, secondary B+ tree index on Major.id and all index pages are in memory.

You as an analyst devise the following query plan for this problem above:



What is the cost of the query plan below? Count only the number of page I/Os.

Ans) The total cost of this query plan is 119 I/Os computed as follows:

• (1) The cost of scanning Applicants is 100 I/Os. The output of the selection operator is

100

20 = 5 pages or 2000

20 = 100 tuples.

• (2) The cost of scanning Schools is 10 I/Os. The selectivity of the predicate on rank is

10−1

100 = 0.09. The output is thus 0.09 ∗ 10 ≈ 1 page or 0.09 ∗ 100 ≈ 9 tuples.

• (3) Given that the input to this operator is only six pages, we can do an in-memory sort-merge

join. The cardinality of the output will be 9 tuples. There are two ways to compute this: (a)

100∗9

(max(100,9)) = 9 (see book Section 15.2.1 on page 484) or (b) consider that this is a key-foreign

key join and each applicant can match with at most one school but keep in mind that the

predicates on city and rank were independent, hence only 0.9 of the applicants end-up with

a matching school.

• (4) The index-nested loop join must perform one look-up for each input tuple in the outer

relation. We assume that each student only declares a handful of majors, so all the matches

fit in one page. The cost of this operator is thus 9 I/Os.

• (5) and (6) are done on-the-fly, so there are no I/Os associated with these operators.

1. Consider relations R(a, b) and S(a, c, d) to be joined on the common attribute a. Assume that there are no indexes available on the tables to speed up the join algorithms. • There are B = 75 pages in the buffer
   * Table R spans M = 2,400 pages with 80 tuples per page
   * Table S spans N = 1,200 pages with 100 tuples per page

Answer the following question on computing the I/O costs for the joins. You can assume the simplest cost model where pages are read and written one at a time. You can also assume that you will need one buffer block to hold the evolving output block and one input block to hold the current input block of the inner relation.

* 1. Assume that the tables do not fit in main memory and that a high cardinality of distinct values hash to the same bucket using your hash function h1. What approach will work best to rectify this?

Ans) When there's a high cardinality of distinct values, leading to many hash collisions in the hash join, one approach to rectify this is to use a multi-step hash join. Here's how you can improve the situation:

1. **Partitioning:**

* Partition both relations R and S into smaller subsets based on the join attribute a.
* This helps reduce the number of collisions within each partition.

1. **Hash Function Variation:**

* Use a different hash function for each partition.
* This helps avoid global collisions and increases the chances of more even distribution within each partition.

1. **Multiple Hash Tables:**

* Build separate hash tables for each partition of both R and S.
* This reduces the impact of collisions within each partition and allows for more efficient probing.

1. **Intermediate Results:**

* Store the intermediate results of each partition in a temporary storage.
* Perform the final join on the intermediate results.

This approach helps in dealing with the high cardinality issue and reduces the number of collisions, making the hash join more efficient.

* 1. I/O cost of a Block nested loop join with R as the outer relation and S as the inner relation

Ans) In a block nested loop join, each block of the outer relation R is read once, and for each block, all blocks of the inner relation S are read. Let's calculate the I/O cost:

1. **Reading Outer Relation (R):**

* Number of blocks in R (M): 2,400
* Number of tuples per block: 80
* Number of blocks to read R: ⌈M/buffer size⌉ = ⌈2,400/75⌉ = 32

1. **Reading Inner Relation (S) for Each Block of R:**

* Number of blocks in S (N): 1,200
* Number of tuples per block: 100
* Number of blocks to read S for each block of R: ⌈N/buffer size-1⌉ = ⌈1,200/74⌉ = 17 (one buffer block is used for output, so buffer size - 1)

1. **Total I/O Cost:**

* Total I/O cost = Blocks read from R + (Blocks read from R × Blocks read from S for each block of R)
* Total I/O cost = 32 + (32 × 17)

Therefore, the I/O cost of the block nested loop join with R as the outer relation and S as the inner relation is given by the calculated total I/O cost.

1. Given a full binary tree with 2n internal nodes, how many leaf nodes does it have?

Ans) A full binary tree is a binary tree in which each node has either 0 or 2 children. If the tree has ‘n’ internal nodes, it means there are ‘n’ nodes with two children, and each internal node has two outgoing edges, contributing to a total of ‘2n’ edges in the tree.

For a full binary tree, the number of leaf nodes (*L*) can be determined using the following relationship:

*L*=*I*+1

where ‘*I’* is the number of internal nodes. In this case, *I*=2n, so:

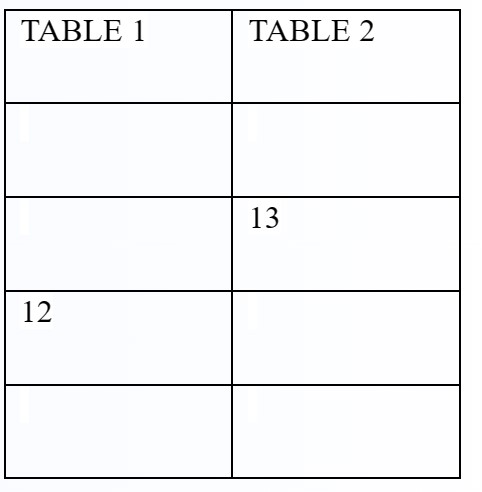
*L*=2n+1

Therefore, a full binary tree with ‘2n’ internal nodes has ‘2n+1’ leaf nodes.

1. Consider the following cuckoo hashing schema below:

Both tables have a size of 4.The hashing function of the first table returns the fourth and third least significant bits: h1(x) = (x >> 2) & 0b11.The hashing function of the second table returns the least significant two bits: h2(x) = x & 0b11.

When inserting, try table 1 first. When replacement is necessary, first select an element in the second table. The original entries in the table are shown in the figure below.



What sequence will the above sequence produce? Choose the appropriate option below:

* + 1. Insert 12, Insert 13
    2. Insert 13, Insert 12
    3. None of the above. You cannot have more than 1 Hash table in Cuckoo hashing
    4. I don’t know

Ans) Based on the information provided in the image and your query, the correct sequence for inserting 12 and 13 into the cuckoo hashing schema is:

a.) Insert 12, Insert 13

Here's why:

Inserting 12:

h1(12) = (12 >> 2) & 0b11 = 0b10 (2 in binary)

Since 2 is already present in Table 1, we need to perform replacement.

We choose an element from Table 2 for replacement. Since both elements (3 and 11) have the same hash value (h2(3) = h2(11) = 0b11), we can choose either. Let's choose 3.

We place 12 in the position previously occupied by 3 in Table 1.

We place 3 in the empty position in Table 2.

Inserting 13:

h1(13) = (13 >> 2) & 0b11 = 0b11 (3 in binary)

3 is not present in Table 1, so we can directly insert 13 into the empty position.

Therefore, the sequence "Insert 12, Insert 13" successfully places both elements in the cuckoo hashing schema without any further collisions.