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) To sort a file of N pages using external merge sort, we divide the file into chunks of size B pages each, sort each chunk in memory using quicksort, and then merge the sorted chunks using B-1 buffers.

To calculate the number of passes needed to sort a file of N pages with B buffers, we can use the formula:

Passes = ceil(log_base_(B-1) N)

where ceil(x) is the smallest integer greater than or equal to x.

Using B = 6 and N = 1,000,000, we get:

Passes = ceil(log_base_(6-1) 1,000,000) = ceil(log_base_5 1,000,000) = ceil(8.51719319142) = 9

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

2.) Consider the following B+tree.

3

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) To find all keys between 9* and 19* in the given B+tree, we need to traverse the tree from the root to the leaf nodes that contain the keys in that range. Along the way, we need to follow the appropriate pointers to navigate between parent and child nodes, and between sibling nodes at the same level.

Starting at the root node with key 3, we compare 9* to the keys in the node and find that 9* is greater than all keys in the node. Therefore, we follow the right pointer to the next node with key 7.

In the node with key 7, we again compare 9* to the keys in the node and find that 9* is greater than all keys in the node. Therefore, we follow the right pointer to the next node with key 12.

In the node with key 12, we find that 9* is less than the first key and greater than the second key. Therefore, we follow the left pointer to the previous node with key 7 and then follow the right pointer to the next node with key 14.

In the node with key 14, we find that 9* is less than the first key and greater than the second key. Therefore, we follow the left pointer to the previous node with key 12 and then follow the right pointer to the leaf node with keys 15, 16, and 19*.

In the leaf node with keys 15, 16, and 19*, we find all the keys between 9* and 19*.

Therefore, we chased a total of 4 pointers (parent-to-child and sibling-to-sibling) to find all keys between 9* and 19*.

3.) 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) To determine the largest key less than 25 whose insertion will cause a split, we need to consider the hash functions and the number of keys in each bucket.

According to the hash function h0(x), the hash value of a key x is determined by taking the rightmost 2 bits of x. Therefore, any key whose rightmost 2 bits match those of another key in the same bucket will cause a split.

Looking at the table in Figure 2, we can see that the bucket for hash value 1 currently contains keys 1 and 17. The next key that would cause a split in this bucket is the largest key with a hash value of 1 and a value less than 25. To determine this key, we can examine the binary representation of 25 and find the rightmost 2 bits:

25 = 11001

The rightmost 2 bits of 25 are 01. Therefore, the largest key less than 25 that would cause a split in the bucket with hash value 1 is the largest key whose rightmost 2 bits are 01. This key is:

23 = 10111

Therefore, the largest key less than 25 whose insertion will cause a split is 23.

4.) 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) A sparse B+ tree of order d = 2 is a B+ tree where each internal node has at most 2 children, and each leaf node has at most 2 keys. In other words, the tree is very unbalanced and has many levels.

To determine the number of nodes in the B+ tree containing the keys 1 through 20, we can start by drawing the tree. Since the tree is very unbalanced, we will draw it in a horizontal fashion for clarity:

```
                     +------------------+
                     |        11        |
                     +---------+--------+
                               |
         +----------------+-----------+----------------+
         |                |                    |
    +-------+------+    +-+-----+            +---+---+
    |      4     |     | 11    |            | 17   20
    +-------+------+    +------+           +-------+
         |                 |               |
      +----+----+         +----+----+       +-----+-----+
      |       |          |      |       |        |
    +--+--+  +---+--+     +--+--+  +---+--+    +-+--+    +-+-+
    | 1  2 |  | 5  7 |     |10 11|  |13 15|    |18 19|   | 20 |
    +------+------+       +-----+  +------+    +-----+   +----+
```

We can see that the tree has a height of 5, with 1 root node, 2 internal nodes at level 1, 4 leaf nodes at level 2, 6 leaf nodes at level 3, and 10 leaf nodes at level 4.

Therefore, the B+ tree containing the keys 1 through 20 has a total of 23 nodes (1 root node + 2 internal nodes + 4 leaf nodes at level 2 + 6 leaf nodes at level 3 + 10 leaf nodes at level 4).

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

4

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(R⋈b=b σc=3(S)))

Which plan is more efficient than the other?

Ans) Assuming that the given SQL query is correct (i.e., there is no typo in the query), the two equivalent logical plans in relational algebra are:

I. πa(σc=3(R ⋈b=b (S)))

II. πa(R⋈b=b σc=3(S)))

Both plans involve a join between R and S on the attribute b, followed by a selection on the attribute c. However, the order in which the operations are performed is different in the two plans.

Plan I first performs the join between R and S on the attribute b, and then applies the selection on the attribute c. This means that all tuples in the join result are considered, regardless of whether they satisfy the selection condition or not. Therefore, plan I may involve more intermediate tuples than plan II.

Plan II, on the other hand, applies the selection on the attribute c before performing the join between R and S. This means that only tuples in S that satisfy the selection condition are joined with tuples in R, potentially reducing the size of the intermediate result. Therefore, plan II is likely to be more efficient than plan I.

In general, it is a good idea to perform selections and projections as early as possible in a query plan, to reduce the size of intermediate results and improve efficiency. However, the actual efficiency of a query plan also depends on the size and distribution of the data, as well as the available indexing and optimization techniques in the underlying database system.

6.) 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, each operator is designed to process a batch of input tuples in a vectorized fashion, i.e., by applying the same operation to multiple tuples at once. This allows for better utilization of the CPU cache and processor pipelines, and can lead to significant performance improvements compared to row-by-row processing.

When an operator receives input from multiple children, it may need to merge and sort the input tuples before applying the operation. However, this can still be done in a vectorized fashion by processing multiple batches of input tuples from each child in parallel. The output tuples can also be generated in a vectorized fashion by applying the operation to multiple tuples at once.

In summary, while multi-threaded execution can be used to parallelize the processing of input batches from multiple children, it is not strictly required for vectorized processing in the presence of multiple inputs. Vectorized processing can still be achieved by processing multiple batches of input tuples in a parallel and vectorized fashion, and generating output tuples in a vectorized fashion as well.

7.) How can you optimize a Hash join algorithm?

Ans) The hash join algorithm is a popular join algorithm used in database systems to efficiently join large datasets. Here are some techniques that can be used to optimize the performance of the hash join algorithm:

1. Hash function selection: Choosing an appropriate hash function that can evenly distribute the data among the available buckets can significantly improve the performance of the hash join algorithm. A poor hash function can result in uneven distribution of data, causing some buckets to be overloaded, which can lead to poor performance.

2. Bucket size selection: Choosing an appropriate bucket size can also impact the performance of the hash join algorithm. A large bucket size can lead to more collisions and result in a degraded performance, while a small bucket size can increase the number of buckets and result in higher memory consumption. Finding the right balance between bucket size and number of buckets is important for optimizing performance.

3. Join order: Changing the order in which tables are joined can also impact the performance of the hash join algorithm. Joining smaller tables first can lead to a smaller intermediate result that can fit in memory, reducing the need for disk I/O and improving performance.

4. Join algorithm selection: Depending on the characteristics of the datasets being joined, different join algorithms may be more appropriate than the hash join algorithm. For example, if one or both of the datasets being joined are small enough to fit in memory, a nested loop join or a sort-merge join algorithm may be more appropriate and can result in better performance.

5. Memory allocation: Efficient memory allocation strategies can also improve the performance of the hash join algorithm. For example, pre-allocating memory for hash tables and buffers can reduce the overhead of memory allocation during the join operation.

6. Parallel processing: Parallelizing the hash join operation across multiple threads or cores can improve the performance of the algorithm by allowing multiple buckets to be processed simultaneously.

Overall, optimizing the performance of the hash join algorithm requires a combination of careful selection of parameters, efficient memory management, and algorithmic and architectural optimizations.

8.) 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.

5

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 cost of a query plan is typically measured by the number of I/O operations required to execute the plan. In general, the more I/O operations required, the higher the cost of the plan.

To estimate the cost of a query plan, you can follow these steps:

1. Identify the tables and indexes involved in the query and estimate the number of pages that need to be read from each.
2. Estimate the number of join operations required and the size of the intermediate results. This can be done by estimating the number of rows returned by each table and the selectivity of the join predicates.
3. Estimate the cost of any sorting or grouping operations required by the query.
4. Add up the estimated I/O operations for each step in the plan to arrive at a total cost estimate.

In the given scenario, assuming that the B+ tree index on Major.id is used for the join operation between Applicants and Major tables, the cost of the query plan would depend on the number of pages that need to be read from the Applicants and Schools tables based on the selection criteria.

However, without additional information or the actual query plan, it is not possible to estimate the cost of the plan accurately.

9.) 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.

A.) 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?

6

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

Ans) A.) If a high cardinality of distinct values hash to the same bucket using the hash function h1, we can use an alternative hash function h2 to rehash the tuples that collide in the first hash table. This approach is called double hashing and can help to distribute the tuples more evenly among the buckets.

B.) For a block nested loop join with R as the outer relation and S as the inner relation, we need to perform the following steps:

1. Read a block B of R into memory

2. For each block Bi of S:

   a. Read Bi into memory

   b. For each tuple r in B and each tuple s in Bi such that r.a = s.a, output the join result r, s

3. Repeat steps 1 and 2 until all blocks of R and S have been processed

The I/O cost of a block nested loop join with R as the outer relation and S as the inner relation can be computed as follows:

- We need to read all blocks of R, which is M blocks.

- We need to read all blocks of S, which is N blocks.

- For each block Bi of S, we need to read it into memory and scan it once to join with the blocks of R. Since we have B buffer pages, we can only hold B-2 pages in memory at a time (one for the input block of S, one for the output block, and B-2 for the input blocks of R). Therefore, the number of times we need to read Bi is ceil(M/(B-2)) times.

- For each block Bi of S, we need to join it with each block of R, which gives us M*N join operations for each Bi. Since each block contains 100 tuples, each join operation requires us to read 80 tuples from R and 100 tuples from Bi. Therefore, the total number of I/Os for each Bi is ceil(M/(B-2)) * (M * N) * (80 + 100) / (B - 2).

The total I/O cost is the sum of the I/O cost for all blocks of S:

I/O cost = N + M + ceil(M/(B-2)) * (M * N) * (80 + 100) / (B - 2)

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

Ans) A full binary tree has exactly 2 children for each internal node, and every level of the tree is fully filled with nodes except for the last level, which may or may not be completely filled. Let's denote the number of internal nodes as n.

The number of nodes in a full binary tree can be calculated using the formula:

Total nodes = 2^(h+1) - 1,

where h is the height of the tree. For a full binary tree, the number of internal nodes n can be calculated as:

n = 2^h - 1

Given that the tree has 2n internal nodes, we can solve for h as follows:

2n = 2^h - 1

2n + 1 = 2^h

h = log_2(2n + 1) - 1

Since the last level may or may not be completely filled, the number of leaf nodes can be either 2^n or 2^n - 1.

If the last level is completely filled, then there are 2^n leaf nodes. Otherwise, there are 2^n - 1 leaf nodes.

Substituting the value of h in terms of n, we get:

If the last level is completely filled:

number of leaf nodes = 2^n

If the last level is not completely filled:

number of leaf nodes = 2^(n-1) * (2n + 1 - 2^n)

Therefore, the number of leaf nodes depends on whether the last level of the full binary tree is completely filled or not.

11.) 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:

a.) Insert 12, Insert 13

b.) Insert 13, Insert 12

c.) None of the above. You cannot have more than 1 Hash table in Cuckoo hashing

d.) I don't know

Ans) The correct answer is (a) Insert 12, Insert 13.

When we insert 12, it hashes to index 0 in the first table (h1(12) = 0b00). Since this slot is empty, we simply insert 12 there.

Next, when we insert 13, it hashes to index 1 in the first table (h1(13) = 0b01). However, this slot is already occupied by 12. Therefore, we need to replace 12 and try to insert it in the second table. The second table has a size of 4, so we can hash 12 to either index 0 or index 1 in the second table (h2(12) = 0b11). Both these slots are empty, so we can simply insert 12 in either of them.

Now, when we try to insert 13 again, it hashes to index 1 in the first table (h1(13) = 0b01). However, this slot is already occupied by 12. Therefore, we need to replace 12 and try to insert it in the second table. However, both slots in the second table are already occupied. This means that 12 needs to be evicted and replaced with 13 in the first table. We can do this by hashing 12 to index 0 in the first table and 13 to index 1 in the second table, and then inserting them both.

Therefore, the correct sequence is to insert 12 first, then insert 13.