DATABASE MANAGEMENT SYSTEMS

FALL 2023

CS – 623 PROJECT

Professor: Mr. Buchi Okoli Okoli

Manikanta Reddy Yerolla

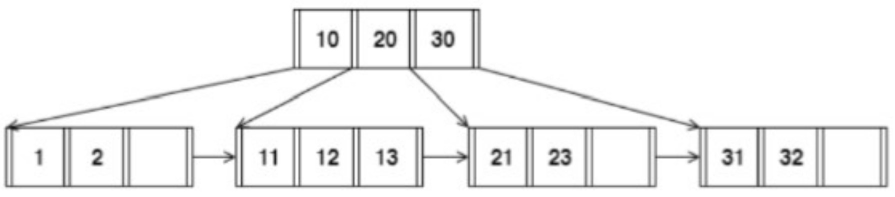
1. We have a file with a million pages (N = 1,000,000 pages), and we want to sort it using external merge sort. Assume the simplest algorithm, that is, no double buffering, no blocked I/O, and quicksort for in-memory sorting. Let B denote the number of buffers.
   How many passes are needed to sort the file with N = 1,000,000 pages with 6 buffers?

To perform an external merge sort on 1,000,000 pages with 6 available buffers, we first execute an initial pass to generate sorted runs. During this pass, we utilize 5 of the buffers for input (since one buffer is reserved for output). Each buffer is responsible for sorting a portion of the file, leading to the creation of 200,000 sorted runs.

In the subsequent phases, we merge 5 runs together at a time while using one buffer for output. The determination of the total number of passes required depends on how many times we can merge 5 runs into a single run, repeating this process until only one sorted run remains. This entire procedure encompasses a total of 9 passes, encompassing the initial sorting pass and the subsequent merging passes.

2. 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∗ ?

Consider the following B+tree.



Begin at the root node, compare 9 with the keys in the root node and decide which path to follow. Since 9 is less than 10, we would typically follow the leftmost pointer. However, we are looking for keys greater than or equal to 9, so we take the path that leads us to keys greater than or equal to 10, which is the pointer between 10 and 20.
At the child node with keys 11, 12, and 13, all these keys fall within the range 9 to 19. So, we would access all these keys. Since this is a B+ tree, the leaf nodes are linked. Therefore, we would follow the sibling pointer to the next leaf node to check for keys up to 19.

The next leaf node contains keys 21 and 23, which are greater than 19.
So, one parent to child and one sibling to sibling that gives us **2** pointers to chase all the 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.

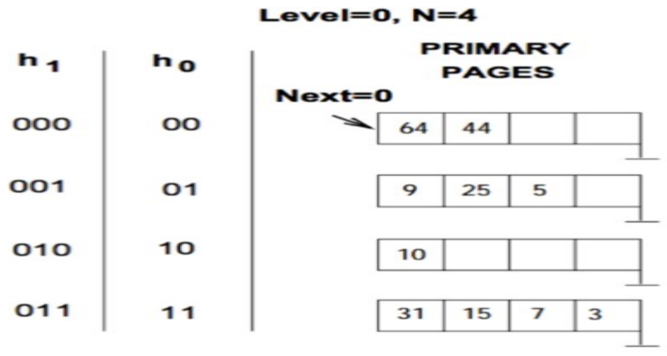


Figure 2: Linear Hashing

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

In the given linear hashing structure, Level=0 employs a hash function denoted as h0(x), which takes into account the rightmost 2 bits of the key x. Let's consider the binary representations of two keys: 24 ('11000') and 23 ('10111'). For 24, the rightmost 2 bits are '00,' and for 23, they are '11.'

When inserting the key 24, which ends in '00,' into the bucket indexed by '00,' there is no need for a split if there is still available space in the '00' bucket. In this case, the '00' bucket only contains the keys 64 and 44, so there's room for more.

However, inserting the key 23, which ends in '11,' into the bucket indexed by '11' would indeed trigger a split if the '11' bucket is already full. Assuming a bucket capacity of 3 keys and an overflow page created when a fourth key is inserted, the '11' bucket with keys 31, 15, 7, and 3 is already at its maximum capacity.

Given the current state of the hash table and the provided rules, inserting key 23 would indeed cause a split because the '11' bucket is unable to accommodate another key. Therefore, the largest key less than 25 that, when inserted, will cause a split is indeed 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?

- In a sparse B+ tree of order d=2, each node carries the minimum number of keys, which is 2, except for the root node which can have at least 1 key. With keys ranging from 1 to 20:
- There are 10 leaf nodes since each one holds exactly 2 keys.
- The next level up has 5 nodes, with each parent node connecting to 2 leaf nodes.
- The process of halving continues up the tree levels, resulting in 3 nodes at the next level.
- The root node is a single node, as it points to the 3 nodes below it.
- Adding these nodes across all levels, the sparse B+ tree has **19** nodes in total.

5. 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. $\pi a(\sigma c=3(R \bowtie b=b\ (S)))$
   II. $\pi a(R\bowtie b=b\ \sigma c=3(S)))$
   Which plan is more efficient than the other?

I. In this plan, the first step involves joining S and U based on their shared attribute b. Subsequently, the results are filtered to include only those rows where U.e equals 6. Afterward, this filtered result is joined with R, and finally, the projection operation πa is performed.

II. Conversely, this plan starts by joining S and U based on their common attribute b. Following the join, a filter operation σe=6 is applied to the result, retaining only rows where U.e equals 6. Then, the filtered result is immediately joined with R based on their shared attribute b. Finally, the projection operation πa is applied to the outcome.

The more efficient plan is typically the one that minimizes the size of intermediate results at the earliest stages of query execution. Filtering as early as possible, when the necessary columns are available, generally reduces the number of tuples involved in subsequent joins. In this specific scenario, Plan II is likely to be more efficient because it applies the σe=6 condition on the join between S and U before proceeding to join with R. This means that R is joined with a potentially smaller subset of the S and U join, resulting in fewer operations compared to Plan I, where the join is executed first, followed by the selection condition.

To summarize, Plan II's early filtering likely leads to a smaller dataset for the final join with R, making it a more efficient choice.

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.

True.

In the vectorized processing model, operators are designed to efficiently handle inputs from various sources without the need for multi-threaded execution. This model functions by processing data in groups or batches, which enables faster data processing compared to handling each individual item separately. In contrast, the iterator model operates in a more integrated manner. The crucial element here is the Next() function, which not only retrieves the next batch of data but also manages the flow of operations. It's akin to receiving both data and instructions on what to do next in a single step. This approach intertwines data retrieval with control flow, and interestingly, it can be effectively managed using just a single thread.

In summary, while the vectorized model excels at swiftly processing large data batches, the iterator model combines data processing with sequential control and can be efficiently executed within a single-threaded environment.

7. How can you optimize a Hash join algorithm?

Optimizing a hash join algorithm involves several crucial strategies:

Effective Memory Management: Ensure efficient memory utilization to store the hash table in-memory while minimizing disk I/O.

Choose the Smaller Table: Select the smaller of the two tables to build the hash table, reducing its size and the risk of exceeding available memory.

Well-Distributed Hash Function: Implement a well-designed hash function to minimize key collisions, ensuring even distribution across buckets.

Parallel Processing: Leverage parallel processing capabilities to distribute the workload across multiple CPUs, enhancing overall performance.

Bloom Filters for Efficiency: Employ bloom filters for rapid checks of row correspondence, saving computational resources.

Batch Processing: Process data in batches to enhance cache efficiency and minimize data retrieval latency.

Data Partitioning: For large datasets, partition the data to ensure it fits within available memory, reducing the need for disk access.

Optimize Disk I/O: In cases where data spills to disk, optimize the read/write processes to minimize latency and maximize throughput.

Hardware Considerations: Tailor the optimization strategy to your specific hardware setup, taking into account factors like cache size and disk speed for optimal performance.

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).

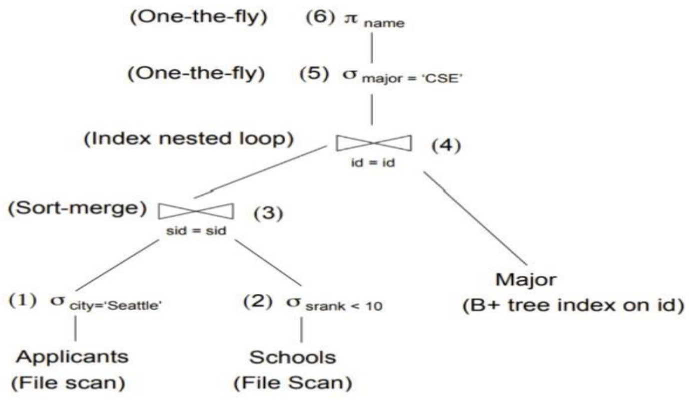
Relation	Cardinality	Number of pages	Primary key
Applicants (id, name, city, sid)	2,000	100	id
Schools (sid, sname, srank)	100	10	sid
Major (id, major)	3,000	200	(id,major)

```
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:


(One-the-fly) (6) π name
(One-the-fly) (5) σ major = 'CSE'
(Index nested loop) ⋈ id = id (4)
(Sort-merge) ⋈ sid = sid (3)
(1) σ city='Seattle'
(2) σ srank < 10
Applicants (File scan)
Schools (File Scan)
Major (B+ tree index on id)

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

Scanning the Applicants table, which results in 100 I/Os and generates 100 tuples.

Scanning the Schools table, which consumes 10 I/Os and generates 9 tuples.

A sort-merge join on these results is performed in memory, incurring no additional I/O cost, and it produces 9 tuples.

Subsequently, an index-nested loop join is executed on the merged output, requiring 9 additional I/Os.

The final two steps of the plan are carried out without incurring any further I/O costs.

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?

To address the issue of high cardinality values leading to hash collisions in join operations, consider three main strategies. Firstly, refine the hash function to ensure a more uniform distribution of keys, decreasing the likelihood of collisions. Secondly, use a partition-based hash join like the Grace Hash Join, which breaks down the data into smaller, manageable chunks that fit in memory, thus cutting down on I/O costs. Lastly, Optimize the Hash Function: Enhance the hash function to achieve a more balanced distribution of keys, thereby reducing the chances of collisions.

Utilize Partition-Based Hash Join: Implement a partition-based hash join approach, such as the Grace Hash Join. This method involves breaking down the data into smaller, manageable segments that can comfortably fit in memory, effectively decreasing I/O costs and the likelihood of collisions.

Increase Memory Buffer Size: If feasible, expand the size of the memory buffer to accommodate a larger portion of the data. This enlargement of the buffer can further mitigate collision issues by allowing more data to be processed in memory, minimizing the need for disk access.if feasible, expand the buffer size to hold more data in memory, which can further reduce collisions.

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

The I/O cost for a Block Nested Loop Join (BNLJ) with R as the outer relation and S as the inner relation can be estimated under the simplest cost model by considering the full scan of the outer table and repeated scans of the inner table for each block of the outer table that fits in the buffer. Given B = 75 buffer pages, M = 2,400 pages for R, and N = 1,200 pages for S, and considering that we use one buffer page for output and one for the current input block from S, we are left with B - 2 pages for blocks from R. The cost formula is:

Total I/O Cost=M+(M/(B-2)) ×N

Total I/O Cost=2400+(2400/ (75-2)) *1200

The I/O cost for a Block Nested Loop Join with R as the outer relation and S as the inner relation is approximately 41,852 page I/Os.

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

In a full binary tree containing 2n internal nodes, we can analyze the total number of nodes, which is represented as T. This total is the sum of two components: the count of internal nodes, denoted as I, and the count of leaf nodes, denoted as L.

Each internal node in a full binary tree has precisely two children, which leads to the equation T = I + L. In the context of full binary trees, it's a characteristic that the number of leaf nodes is consistently one more than the number of internal nodes, thus giving us the relationship L = I + 1.

Given that we have information that I equals 2n, we can determine the number of leaf nodes, L, as 2n + 1.

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.

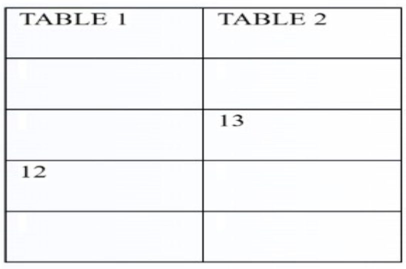


TABLE 1	TABLE 2
	13
12	

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.

a In cuckoo hashing, the process of inserting keys into two hash tables involves handling collisions by displacing the existing key, which then attempts to find a new place in the other table. Let's consider the scenario with keys 12 and 13, utilizing hash functions h1 and h2.

Both keys 12 and 13 initially hash to position 3 in Table 1. When we insert 12 first, it displaces any existing key at position 3. In this case, 12 is the one that gets kicked out. However, 12 has an alternate position, which is determined by h2, leading it to position 0 in Table 2, where it can be accommodated without any issues.

Following this sequence, we can successfully place both keys: 12 in Table 2 and 13 in Table 1. Therefore, the correct outcome in this scenario is option "a" (Insert 12, Insert 13).