DATABASE MANAGEMENT SYSTEMS

FALL 2023

CS – 623 PROJECT

Professor: Mr. Buchi Okoli Okoli

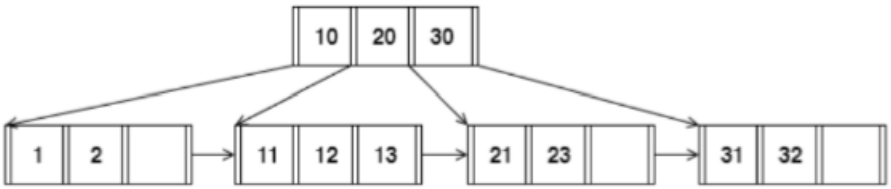
Vishnu Vardhan Reddy Bijjam

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 sort 1,000,000 pages using external merge sort with 6 buffers, we perform an initial pass to create sorted runs using 5 buffers (since one is for output. In each pass, one buffer is used for output, leaving B−1=5 buffers for input). Each buffer sorts a portion of the file, resulting in 200,000 runs. In subsequent passes, 5 runs are merged at a time (one output buffer). The total passes needed are determined by how many times you can merge 5 runs into 1, repeating this until only one sorted run remains. This process takes a total of 9 passes, including the initial sorting pass and the 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.

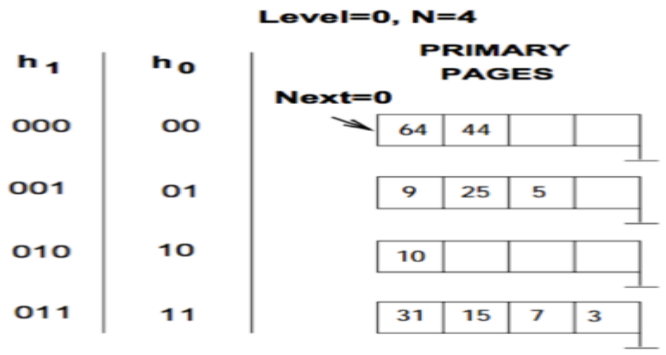


Figure 2: Linear Hashing

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

In the given linear hashing structure, when inserting a key, the hash function being used at Level=0 is h0 (x), which considers the rightmost 2 bits of the key x. If we consider the binary representations of 24 ('11000') and 23 ('10111'), the rightmost 2 bits are '00' and '11', respectively.
Inserting 24, which ends in '00', into the bucket indexed by '00' would not cause a split if there's still space left in the '00' bucket, which in this case there is, as it only contains the keys 64 and 44.
On the other hand, inserting 23, which ends in '11', into the bucket indexed by '11' would indeed cause a split if the '11' bucket is already full, as the keys 31, 15, 7, and 3 are already occupying the slots, assuming the buckets have a capacity of 3 keys and that an overflow page is created when a fourth key is inserted.
Given the current state of the hash table and the rules provided, inserting key 23 would cause a split because the bucket '11' is already at its maximum capacity and cannot accommodate another key.
Therefore, the largest key less than 25 whose insertion 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. πa(σc=3(R ⋈b=b (S)))
II. πa(R⋈b=b σc=3(S)))
Which plan is more efficient than the other?

I.πa(σe=6(R⋈b=b(S⋈b=bU))) This plan first performs a join between S and U on their common attribute b and then filters the results where U.e equals 6. After that, it joins this result with R on their common attribute b and then projects a attribute.

II.πa(R⋈b=bσe=6(S⋈b=bU)) This plan first joins S and U on their common attribute b, filters the results where U.e equals 6, and then immediately joins the filtered result with R on their common attribute b. After the join, it projects a attribute.

The more efficient plan is typically the one that reduces the size of intermediate results as early as possible in the query execution. Filtering as early as possible (as soon as the necessary columns are available) generally reduces the number of tuples involved in subsequent joins.
In this case, Plan II is likely to be more efficient because it applies the selection condition σe=6 on the join between S and U before joining with R. This means that R is joined with a potentially smaller subset of the S and U join, leading to fewer operations than joining first and then applying the selection, which is what Plan I does.

To summarize, Plan II applies the filter earlier, which likely results in a smaller set for the final join with R, making it more efficient.

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.

False.
In the vectorized processing model, operators efficiently handle inputs from various sources without the need for multi-threaded execution. This model works by processing data in groups or batches, allowing for quicker data handling compared to processing each item individually. Contrastingly, in the iterator model, the process is more integrated. The Next () function is key here – it not only fetches the next batch of data but also directs the flow of operations. It's akin to getting both the information and instructions on what to do next in one go. This method intertwines data retrieval with the control flow, yet interestingly, it can be effectively managed using just a single thread.
To sum up, while the vectorized model excels in processing large data batches swiftly, the iterator model combines data processing with sequential control, efficiently executable within a single-threaded environment.

7. How can you optimize a Hash join algorithm?

Optimizing a hash join algorithm involves several key strategies: First, manage memory effectively to accommodate the hash table in-memory and minimize disk I/O. Choose the smaller of the two tables for constructing the hash table, reducing its size and the likelihood of overflowing memory. Implement a well-distributed hash function to minimize key collisions. Take advantage of parallel processing to distribute the workload across multiple CPUs. Use bloom filters for rapid checks of row correspondence, and process data in batches to improve cache efficiency. For large datasets, partition the data to ensure it fits in memory. In cases where data must spill to disk, optimize the read/write processes. Finally, tailor the optimization to your specific hardware setup, considering factors like cache size and disk speed.

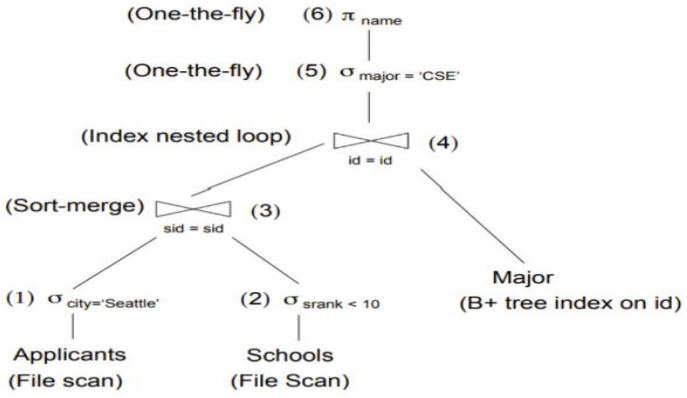
8. Consider the following SQL query that finds all applicants who want to major in CSE, live in Seattle, and go to a school ranked better than 10 (i.e., rank < 10).

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:



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

The query execution plan involves scanning the Applicants table, which consumes 100 I/Os, and scanning the schools table, which uses 10 I/Os. These scans generate 100 and 9 tuples, respectively. A sort-merge join on these results operates in memory, so it doesn't add to the I/O cost, and outputs 9 tuples. Following this, an index-nested loop join on the merged output requires 9 additional I/Os. The final two steps of the plan are executed without incurring further I/O costs. Summing up the I/Os from all these operations gives a total cost of 119 I/Os for the entire query plan.

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, 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 with 2n internal nodes:
The total number of nodes T is the sum of the number of internal nodes I and the number of leaf nodes L.
Each internal node contributes two children, leading to the equation T=I+L.
In full binary trees, the number of leaf nodes is always one more than the number of internal nodes, so L=I+1.
Given I=2n, we find that the number of leaf nodes L is 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.) Insert 12, Insert 13

In cuckoo hashing, when inserting keys into two hash tables, if a key encounters a collision, it displaces the existing key, which then seeks a place in the other table. With keys 12 and 13, using hash functions h1 and h2, both hash to position 3 in Table 1. Inserting 12 first and then 13 causes 12 to be kicked out. 12 then goes to its alternate position, 0 in Table 2, determined by h2, where it fits without issue. This sequence ensures both keys are placed successfully: 12 in Table 2 and 13 in Table 1, making option a (Insert 12, Insert 13) the correct outcome.