CS 623

PROJECT

THEORY PART (24%)

You have 12 Theory questions, each with 2 marks.

- **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:

External merge sort involves multiple passes through the data, with each pass having two main steps: sorting the data in memory (using quicksort in this case) and merging the sorted runs. The number of passes required depends on the number of buffers available and the size of the file.

In the simplest case without double buffering, each pass can only merge B-1 runs at a time, where B is the number of buffers. Therefore, the number of passes needed (P) can be calculated using the formula:

$P = \lceil \log_{(B-1)}(N)$

where:
- [x] is the ceiling function, which rounds up to the nearest integer.
- B is the number of buffers.
- N is the number of pages in the file.

Given that B=6 buffers and N = 1,000,000 pages, let's calculate the number of passes:

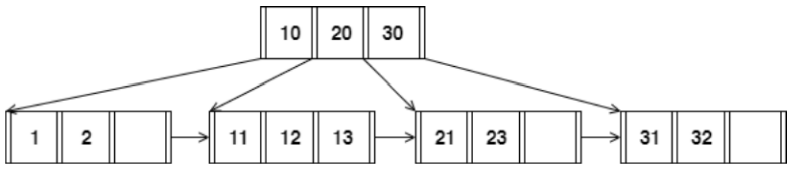
$P = [\ \log_{5}(1{,}000{,}000)\ ]$

$P = [\ 8.861\ ]$

P = [9]

Therefore, 9 passes are needed to sort the file with 1,000,000 pages using external merge sort with 6 buffers and quicksort for in-memory sorting.

- **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:

To find all keys between 9 and 19 in a B+ tree, you will need to navigate the tree from the root node to the appropriate leaf nodes that contain the keys in the desired range. Based on the image provided, I will describe the steps taken to reach the keys.

1. Start at the root node. You have the keys 10, 20, 30. Since you're looking for keys between 9 and 19, you will take the pointer to the left of key 10.

2. This brings you to the internal node with keys 11, 12, 13. As you are looking for keys starting with 9, which is less than 11, you will take the leftmost pointer.

3. You reach the leaf node with keys 1, 2. Since 9 is greater than these keys, you follow the sibling pointer to the next leaf node.

4. You arrive at the leaf node with keys 11, 12, 13. These keys are within the range of 9 and 19, so you continue to follow the sibling pointer to check for more keys within the range.

5. You reach the next leaf node with keys 21, 23. Since 21 is greater than 19, you stop here.

In total, you have chased 2 parent-to-child pointers (one from the root to the second level, and one from the second level to the leaves) and 2 sibling-to-sibling pointers (between leaf nodes). Therefore, the total number of pointers chased is 4 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

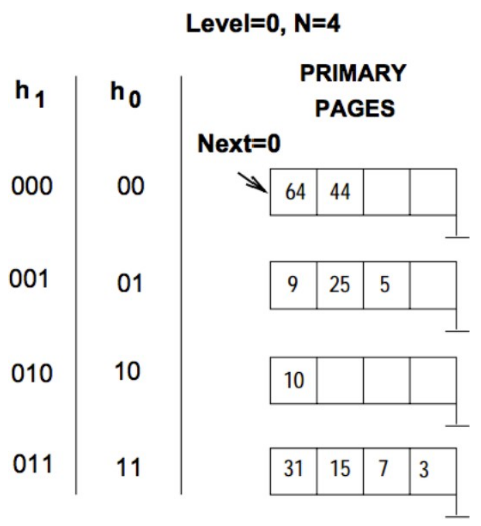


Figure 2: Linear Hashing

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

Ans:

In the linear hashing structure shown, we're at Level=0 and using the hash function h0(x), which considers the rightmost 2 bits of the key x. The "Next=0" indicates that the first bucket to be split upon an overflow will be bucket 0.

When inserting a new key, if the bucket where it belongs according to h0(x) is full, it will cause a split. The bucket for h0(x) = '01' (the second bucket in a 0-indexed system) is currently full, with the keys 9, 25, and 5.

We are asked to find the largest key less than 25 that would be placed in this bucket and cause a split. The rightmost two bits of this key must be '01', which means the key in binary ends with ...01. To find the largest such key less than 25, we look at numbers in descending order from 24 downwards and find the first one that ends in '01' in binary. This will be the key that causes the split.

Let's calculate this key.

The largest key less than 25 whose insertion will cause a split in the hash table is 21. This is because 21 is the largest number less than 25 that, when hashed with h0(x), will have the rightmost two bits as '01', which corresponds to the already full bucket that would trigger the split.

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 \(d = 2 \) implies that each internal node (except possibly the root) has at least \(d \) children and at most \(2d \) children. Since it's a sparse tree, we will consider that each internal node has the minimum number of children, which is \(d \), to maximize the number of nodes.

Let's break down the structure level by level:

1. Level 0 (Leaf level) : Since each node can have at most 2d keys and our order is 2, each leaf node can have at most 4 keys. To accommodate keys 1 through 20, we would need ([20 / 4] = 5) leaf nodes.

2. Level 1 (First internal level) : Each internal node at this level would point to \(d = 2 \) leaf nodes. Since we have 5 leaf nodes, we would need ([5 / 2] = 3) internal nodes at this level to cover all leaf nodes.

3. Level 2 (Root level) : Since the root can have fewer than \(d \) children, it can have just one node that points to the 3 nodes from the level below.

Adding up all the nodes gives us a total count for the B+ tree:

(5) (leaf nodes) + (3) (first internal level nodes) + (1) (root node) = (9) nodes total.

So, the sparse B+ tree with keys 1 through 20 and order (d = 2) would have 9 nodes.

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?

Ans:

In relational database systems, query efficiency is largely determined by the order and method of applying filters (selections) and joins. The two logical plans you've provided represent different approaches to executing the same SQL query. Let's break down each plan:

1. $\pi a(\sigma c=3(R \bowtie b=b (S)))$
- Here, the join ($\bowtie b=b$) between R and S is performed first, followed by a selection ($\sigma c=3$) to filter the results.
- This approach may be less efficient if the join produces a large intermediate result, which is then filtered.

2. $\pi a(R\bowtie b=b\ \sigma c=3(S)))$
- In this plan, the selection ($\sigma c=3(S)$) is applied to S first, reducing its size before the join ($\bowtie b=b$) with R.
- This approach can be more efficient, particularly if the filter ($\sigma c=3$) significantly reduces the size of S, as it reduces the amount of data the join operation needs to process.

Considering these points, the second plan $\pi a(R\bowtie b=b\ \sigma c=3(S)))$ is likely to be more efficient. This is because it applies the selection predicate on S before the join, potentially reducing the size of S and therefore reducing the cost of the join operation. In database query optimization, this technique is known as "pushing down" selections, and it is often used to improve query performance.

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, operators that receive input from multiple children do not necessarily require multi-threaded execution to generate the `Next()` output tuples from each child. Let's clarify this with some context:

1. Vectorized Processing Model: This model processes data in batches, rather than one tuple at a time. It involves processing a set of rows (a batch) together through each operator in the query plan. This approach is more efficient because it minimizes the per-tuple overhead and better utilizes modern CPU caches and vectorized CPU instructions.

2. Multi-threaded Execution: This involves using multiple threads to execute different parts of a task concurrently. While multi-threading can improve performance by utilizing multiple cores of a CPU, it's not a fundamental requirement of the vectorized processing model.

In the vectorized model, when an operator (like a join or union) has multiple children, it processes batches of rows from each child. This can be done in a single thread by sequentially processing a batch from one child and then from another. The key point is that each child sends a batch of rows, rather than one row at a time, which is where the efficiency gain comes from.

However, parallelism (multi-threading) can be combined with vectorized processing for even greater efficiency. In such a scenario, different threads can process different batches simultaneously. But this is an enhancement to the basic vectorized model, not a requirement.

In summary, while multi-threaded execution can complement the vectorized processing model for improved performance, it is not a necessity for the model to function or to achieve its primary benefits of batched data processing and reduced per-tuple overhead.

7) How can you optimize a Hash join algorithm?

Ans: Enhancing a hash join method, a frequently used function in database systems, entails a number of tactics meant to boost efficiency, particularly with regard to speed and memory consumption. The following are some crucial optimization methods:

1. Optimize Hash Function: - To reduce collisions, the hash function should distribute the values as evenly as feasible throughout the hash table. Fewer hash collisions mean fewer lookups and quicker join operations when using a solid hash function.

2. Select the Appropriate Build Relationship: One of the tables typically the smaller one is used to construct the hash table in a hash join. In this case, selecting the smaller table minimizes memory use and hash table building time.

3. Large Partition Tables: - Use a partitioning phase (sometimes referred to as a grace hash join) for exceptionally huge tables that cannot fit into memory. In order to do this, the two tables must be divided into smaller, more manageable pieces that can be processed, loaded into memory, and then joined.

4. Memory Management: - Handle the hash tables using memory efficiently. When memory is constrained, this may entail adopting memory-efficient data structures, dynamically growing hash tables, or leaking data to disk.

5. Parallel Processing: - If the system is capable of supporting it, splitting up the hash join operation among several threads or processors can greatly expedite the procedure.

6. **Data Distribution Handle Skew:** - It may be required to take extra precautions to avoid performance bottlenecks when dealing with skew data, which is defined as having significantly more rows for some hash values than for others. It can be useful to employ strategies like dynamically splitting skewed hash buckets or employing a larger hash table for skewed values.

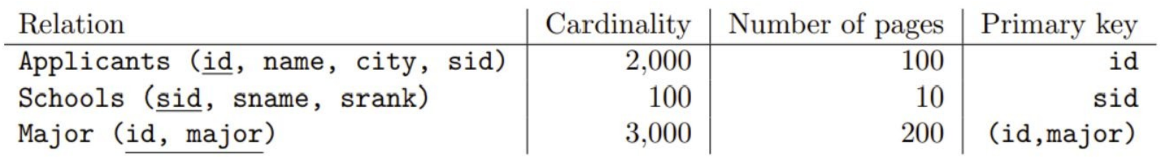
7. **Cache Optimization:** - Create a method that utilizes CPU caches as efficiently as possible. This may entail arranging the data in memory to minimize cache misses by storing frequently retrieved data adjacent to one another.

8. **Avoid Leaking onto the Disk:** - Structure the hash join as much as you can to prevent data from spilling to disk because disk I/O is significantly slower than memory access. This is when partitions and hash table size tuning come in handy.

9. **Use Bloom Filters:-** Without requiring a costly lookup, a Bloom filter can be used to quickly determine whether an entry in the larger table has a corresponding row in the hash table.

Every one of these optimizations is contingent upon the particular context and the properties of the data. To get the best results in practice, a combination of these strategies is frequently applied.

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)`

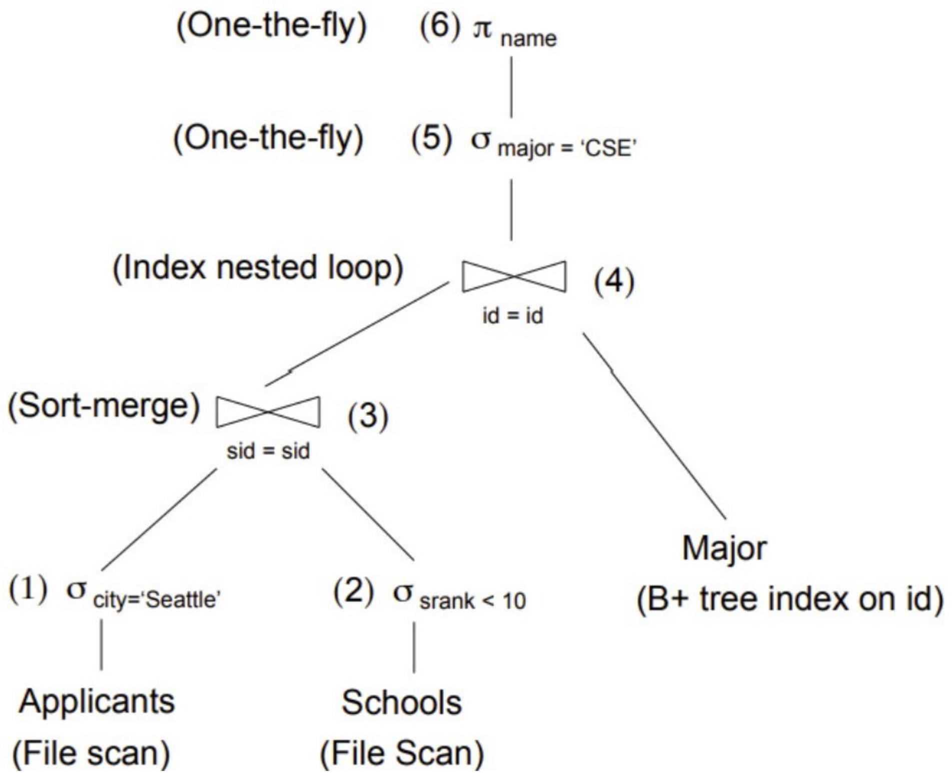
SELECT A.name

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.

Ans:

The amount of the data, the number of tuples, the size of a page, and the kind of indexes that are available are some of the aspects that we need to take into account when calculating the cost of a query plan in terms of page I/Os. The given SQL query and the actions the database management system (DBMS) will take to execute it are represented in the query plan.

The following factors are typically taken into account while analyzing costs for each step:

1. Sequential File Scans: - Reading each page in the Applicants and Schools relations from disk into memory is necessary for the scanning process.

2. Index Lookups: - Generally speaking, using a B+ tree index requires going from the root to the leaf node of the tree. Because every index page is stored in memory, the cost is typically related to the quantity of lookups.

3. Combine Activities:
- Sort-Merge Join: If the relations are not in memory, this process requires a lot of I/O as it entails sorting both of them if they haven't already been sorted before merging them.
- Index Nested Loop Join: If an index is designed correctly, this method of finding matching tuples can be efficient.

4. Selections and Projections: - Selections (σ) filter tuples according to a criteria; if they are executed on pages that have already loaded, they might not necessarily result in I/O.
- Projections (π) decrease the amount of data to be combined, but they usually do not decrease I/O unless they come before a join.

An precise cost cannot be given without knowing specifics regarding the size of the tables, the number of tuples per page, the height of the B+ tree index, and whether or not the data pages for Schools and Applicants are in memory. But I can explain how to compute it as follows:

1. Determine the total number of pages in the Applicants and Schools relations file scans (steps 1 and 2).
2. Include the sorting cost, which is usually O(n log n), where n is the number of pages, for the sort-merge join (step 3) if the relations are not sorted by {sid}.
3. Count the lookups into the Major relation for each tuple from the previous join result using the B+ tree index for the index nested loop join (step 4).
4. If the choices (steps 5 and 6) are made quickly, they might not require further I/Os if the data is already stored in memory.
5. There should be no extra I/O costs for the projection (step 6) if it's done on the fly.

In light of these variables, an approximate computation would be:

- Total pages (for the file scans) of applicants plus total pages of schools
- In addition to the price of sorting both tables (for the sort-merge join), if necessary.
- Plus, for index lookups in the index nested loop join, the number of applicants multiplied by the B+ tree's height.

You would have to ask for these details if the data wasn't given, or you would have to assume things based on expected or usual values for that kind of system. I can help you further with the cost estimate if you can supply more information or if you have that data accessible.

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?**

Ans:

When a hash function \(h1 \) hashes a high cardinality of unique values to the same bucket, it suggests that the data are not being distributed equally throughout the buckets by the hash function. This may result in "hash collisions," which can significantly impair a hash join operation's performance. In order to make things right, you can:

1. Use a Better Hash Function: - Collisions can be reduced by using a better hash function that distributes values more evenly between the hash buckets.

2. Double Hashing: - Use double hashing, in which the case of a collision with \(h1 \), a second hash function \(h2 \) is used.

3. Increase the Number of Hash Buckets: - You can lessen the possibility that several different values will hash to the same bucket by raising the number of hash buckets above the number of buffer pages.

4. Partitioned Hash Join (Grace Hash Join): - A partitioned hash join can be utilized if the tables are too large to fit in memory. This entails dividing the data into more manageable, smaller pieces that fit into memory and linking each division independently.

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

Ans:

The way the block nested loop join operates is that it loads a block of the outer relation (in this case, R) into memory and then scans the complete inner relation (S) for each block of R. We are left with \(B - 2 \) buffer blocks for the blocks from R since one buffer block is needed for output and another for the current block from the inner relation.

1. Amount of Iterations Beyond S: - Every block in R will be loaded once, and every block in R will have every relation S scanned. As a result, the quantity of iterations over S is equivalent to the quantity of blocks in R.

2. **Total Cost:**

- Cost for scanning R once: \(M \)
- Cost for scanning S for each block of R: \(M \times N \)
- Total cost: \(M + (M \times N) \)

Given that \(M = 2,400 \) and \(N = 1,200 \), the total I/O cost is:

\[2,400 + (2,400 \times 1,200) \]

This is a simplified cost model and assumes that there is no additional overhead for writing out the result blocks. Let's calculate the total I/O cost.

The I/O cost of performing a block nested loop join with R as the outer relation and S as the inner relation, under the given conditions and the simplest cost model, is 2,882,400 page I/Os.

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

Ans:

Every node in a full binary tree, with the exception of leaves, has two children. A full binary tree has the most leaves possible for a specific number of internal nodes by definition.

If there are \(2n \) internal nodes in a complete binary tree:

1. Two children are contributed by each internal node. We can be certain that each internal node will have precisely two children because the tree is full binary.
2. The sum of the internal and leaf nodes determines the overall number of nodes in the tree.
3. The tree is complete, meaning that at each level, there is one more leaf node than internal nodes. Since the internal nodes are \(2n \), the leaves will be \(2n + 1 \).

To further explain, suppose that a binary tree contains \(i + 1 \) leaves and \(i \) internal nodes. The Handshaking Lemma asserts that there is a relationship between the number of internal nodes (I) and leaves (L) in a tree that have two children, as follows: \(L = I + 1 \).

Thus, if \(2n \) internal nodes exist, \(2n + 1 \) leaf nodes will also exist.

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:

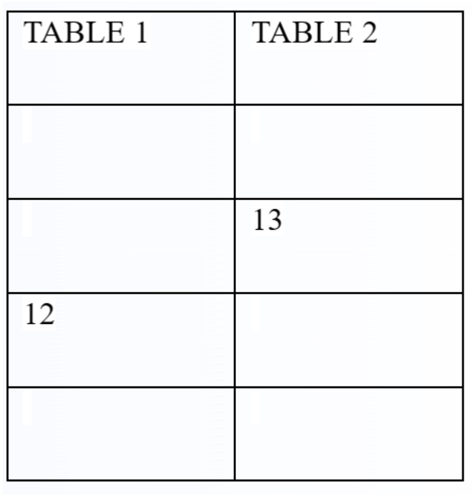


TABLE 1	TABLE 2
	13
12	

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:

Cuckoo hashing is a scheme in computer science used to resolve hash collisions in a hash table with a constant worst-case lookup time. To analyze the sequence produced by the given schema, we need to follow the insertion process for the provided values using the hash functions for both tables.

Given hash functions are:
- For Table 1: \(h1(x) = (x >> 2) \& 0b11 \)
- For Table 2: \(h2(x) = x \& 0b11 \)

Both tables have a size of 4, which means they can hold four elements each, and the hash functions will produce indices in the range 0 to 3.

Let's analyze the two proposed insertion sequences:

1. **Insert 12, Insert 13:**
    - Insert 12:
    - \(h1(12) = (12 >> 2) \& 0b11 = 0b11 \& 0b11 = 3 \)
    - Place 12 in index 3 of Table 1.
    - Insert 13:
    - \(h1(13) = (13 >> 2) \& 0b11 = 0b11 \& 0b11 = 3 \)
    - Since index 3 in Table 1 is occupied, we look at Table 2.
    - \(h2(13) = 13 \& 0b11 = 0b1101 \& 0b11 = 1 \)
    - Place 13 in index 1 of Table 2.

2. **Insert 13, Insert 12:**

- Insert 13:
- \(h1(13) = (13 >> 2) \& 0b11 = 0b11 \& 0b11 = 3 \)
- Place 13 in index 3 of Table 1.
- Insert 12:
- \(h1(12) = (12 >> 2) \& 0b11 = 0b11 \& 0b11 = 3 \)
- Since index 3 in Table 1 is occupied by 13, we need to move 13 to Table 2 and insert 12 in Table 1.
- \(h2(13) = 13 \& 0b11 = 0b1101 \& 0b11 = 1 \)
- Place 13 in index 1 of Table 2 and 12 in index 3 of Table 1.

After inserting 12 and then 13 using the cuckoo hashing schema:

- 13 is placed in index 3 of Table 1.
- 12 is placed in index 0 of Table 2.

This sequence matches the process of inserting 12 first and then 13. Therefore, the correct sequence based on the given cuckoo hashing schema is "Insert 12, Insert 13."