HQ_8_1

Select all of the true statements regarding B+ Trees and ISAM.

✓ B+ Tree is better in tables that grow at a very fast pace compared to an ISAM structure

- Tables that grow quickly may cause overflow in ISAM (for example, situations where there are ever-increasing keys). Though both data structures have the concept of overflow, B+ Trees perform automatic re-balancing. The lookup time for ISAM can become even worse in special cases where overflow pages are not deleted.

✓ ISAM, on average, requires fewer disk operations to visit a data page than a B+ tree if we consider that all of the ISAM data fits into a structure in which we do not have to access any overflow pages.

- ISAM trees are normally fully packed and appear fully optimized with the shortest possible height. In B+ trees, they are rarely fully packed and are usually only more than 50% full.

✗ ISAM will automatically sort its overflow pages in order to minimize lookup time.

- ISAM does not sort overflow pages, and this makes lookup slow, especially if there are long chains of overflow pages. There are implementations in which you can sort overflow pages, however, these implementations will make inserts slower.

✓ Overflow pages in ISAM do not get deleted unless all records in the overflow pages are deleted.

- This may cause very long overflow page chains, which quickly degrade the performance of ISAM. Another reason why B+ Trees seem more attractive than ISAM.

HQ_8_2

Consider a B+ tree index with n = 50, where n is the maximum number of keys fitting in a block. The B+ tree indexes 100,000 records. What is the minimum number of nodes in the tree that we need to examine when searching for a record?

✗ 1

✗ 2

✓ 3

- For a B+ Tree of n = 50, the maximum number of records we can store for a tree with 3 levels is 50^3 - 50^(3 - 1) = 122500. If we have 2 levels, it is 50^2 - 50^(2 - 1), which is 2450, clearly not enough to store our 100,000 records.

✗ 4

HQ_8_3

Consider the following B+ tree in the figure. Each index node can hold at most 4 keys. A student from our CS411 class listed two solutions for inserting key 24 into the original tree: Solution 1: Split the 5th leaf into [20, 22, 23] and [24, 26]. Add a key and a pointer to the rightmost node at level 2, which then has 3 keys [24, 29, 39] and 4 pointers. Solution 2: Split the 5th leaf into [20, 22] and [23, 24, 26]. Add a key and a pointer to the rightmost node at level 2, which then has 3 keys [23, 29, 39] and 4 pointers. Which of the above two solutions is correct?

✗ Both answers are incorrect

✗ Solution 1

✗ Solution 2

✓ Both answers are correct

HQ_8_4

Consider the following B+ tree in the figure, each index node can hold 3 keys. What should be the resulting tree after deleting key 6 from the tree?

✓ Both trees are possible

- When we remove key 6 from the tree, key 7 becomes the only key in that node. However, for each node, we must have at least 2 keys, so we merge key 7 into another node. It does not matter which node we merge key 7 into, so both solutions will work.

✗ Tree A

✗ Tree B

✗ Neither tree is possible

HQ_8_5

Suppose we have a relation of 3,000 tuples with no duplicate keys, and each block can hold 5 tuples. How many key-pointer pairs do we need for a dense index of this relation?

✗ 5

✗ 600

✓ 3,000

- Since this is a dense index, we need a pair for each tuple in the relation. Therefore, we need 3,000 pairs.

✗ 15,000

HQ_8_6

Select the true statement.

✓ Inserts into clustered indexes take longer than unclustered indexes.

- Inserts and updates take longer because we need to put the data back into a sorted order in clustered indexes.

✗ Clustered indexes are always dense

- They can be either dense or sparse, depending on the attribute we index on. If the attribute is unique, it will be dense since we will need to give every value an entry in the index table. If it is not unique, then it can be a sparse index.

HQ_8_7

In this problem, we use an extensible hash table to index the following search key values by inserting them one by one: 22, 69, 35, 81. The hash function h(n) for an integer valued search key n is h(n) = n mod 14. Each data block can hold 2 data items. The result index is shown in the figure. Which of the following are possible according to the figure?

✗ A = 22 (1000), B = 35 (0111), C = 69 (1101), D = 81 (1011)

✗ A = 81 (1011), B = 69 (1101), C = 35 (0111), D = 22 (1000)

✗ A = 69 (1101), B = 35 (0111), C = 35 (0111), D = 22 (1000)

✓ A = 35 (0111), B = 22 (1000), C = 81 (1011), D = 69 (1101)

HQ_8_8

Extensible and Linear Hash Tables are both examples of Dynamic Hash Tables, which automatically allow a hash table to grow and shrink gracefully according to the number of records stored in the table. Static Hash Tables do not automatically grow and shrink when new records are added. Which of the following statements is false?

✓ Static Hash Tables have faster insert than Dynamic Hash Tables.

- This is not necessarily true. Consider the case where a static hash table is currently full, and we want to insert another record. If the bucket is already full, we must add an overflow bucket and chain them to an already existing bucket, which takes time. Whereas for a dynamic hash table, we can simply split the bucket in which we are adding the record. Static hash tables have faster insert when there is space in the table for the new record, while a dynamic hash table needs to split.

❌ For static hash tables, if we want to resize the table by adding more buckets, we would need to create a new hash function.

- In a static hash table, the hash function computes a bucket for a given key using the fixed number of buckets in the hash table. If this number of buckets changes, the hash function must also change.

❌ When a dynamic hash table is being reorganized, we only need to touch the particular bucket that is being split/extended.

- When we redistribute items in a particular bucket, all the elements in other buckets remain untouched. In contrast, with a static hash table, when we redistribute into a hash table with different buckets, every record needs to be redistributed.

❌ In both Linear and Extensible hash tables, the hash function produces a hash value in which we use a certain number of bits from that value to determine the index of the destination bucket.

- This statement is true.

HQ_8_9

Select the true statement.

❌ Extensible Hash Tables do not have overflow chains like Linear Hash Tables, which make them a better choice.

- Though they do not have overflow chains, extensions on the table can be costly and take a long time.

✗ Linear Hash Tables use less memory than Extensible Hash Tables.

- Extensible Hash Tables require more memory because there are no overflow blocks. This means that the entire directory needs to fit into main memory, and after an extension, it may no longer even fit in main memory. When memory is a problem, Linear Hash Tables are a better option, since overflow chains allow for less memory usage.

✓ The search cost for a Linear Hash Table can vary significantly.

- Depending on the number of overflow chains, a Linear Hash Table can have very low search costs or very high search costs. Since a Linear Hash Table has no control over the length of its overflow chains, the search cost can suddenly become very high.

HQ_8_10

Select all of the correct statements.

✓ Hash Tables are faster on average than B+ Trees for 'equality' based lookups (i.e., checking if an attribute is equal to a certain value).

- If there is a hash created on the attribute we want to view, then we will likely have an amortized O(1) lookup cost.

✗ Hash Tables are better than B+ Trees for a range scan (i.e., looking for all keys in a certain range)

- B+ Trees are better, because Hash Tables do not store the keys in any certain order, while B+ Trees do.

✗ Hash Tables are better than B+ Trees for prefix matching of a key (matching a prefix of a key, rather than the entire key).

- Lookup is based on a hash computation of the entire key, so they cannot use just the prefix.

✓ Hash Table performance can degrade quicker than a B+ Tree if we do not carefully manage the structure/scalability of the Table.

- If there are lots of collisions in the Hash Table, we will have to have long overflow chains, which will quickly reduce the performance of the Hash Table. B+ Trees do not have to worry about this, as they will always balance themselves out.