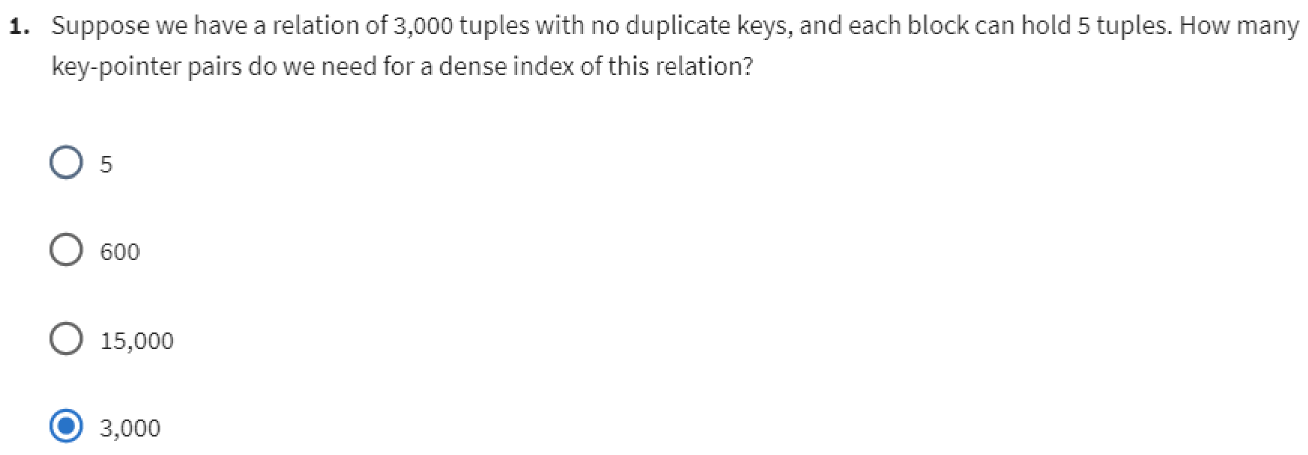
Week 15 Accessing and Indexing Data (3)

Assignment Solutions

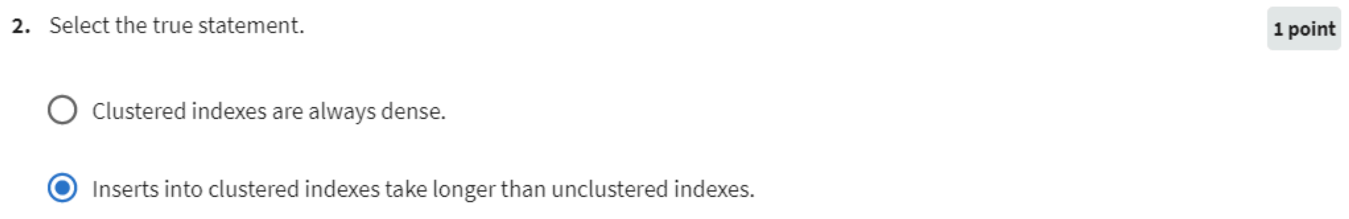


1. Suppose we have a relation of 3,000 tuples with no duplicate keys, and each block can hold 5 tuples. How many key-pointer pairs do we need for a dense index of this relation?

   - 5
   - 600
   - 15,000
   - [✓ 3,000]

Solution: D

Explanation: Since this is a dense index, we need a key-pointer pair for each tuple in the relation. Therefore we need 3000 pairs.



2. Select the true statement. 1 point

○ Clustered indexes are always dense.

[✓ Inserts into clustered indexes take longer than unclustered indexes.]

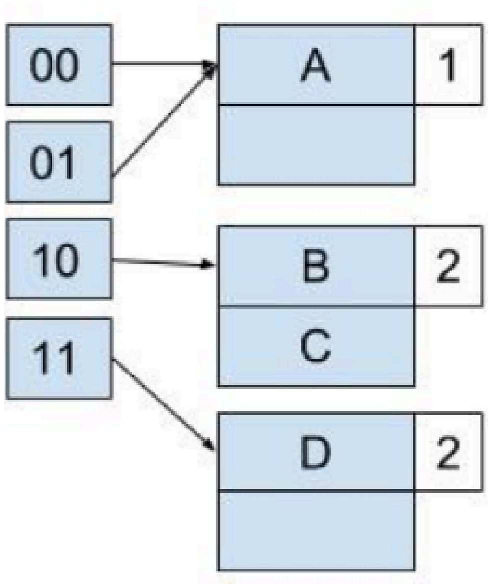
Solution: B

Explanation:

Option A: Clustered indexes can be either dense or sparse, depending on the attribute we index on. In fact, we can use a sparse index only if the index is clustered-- which means the physical order of tuples on the disk is the same as the order of index keys.

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

3. 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?

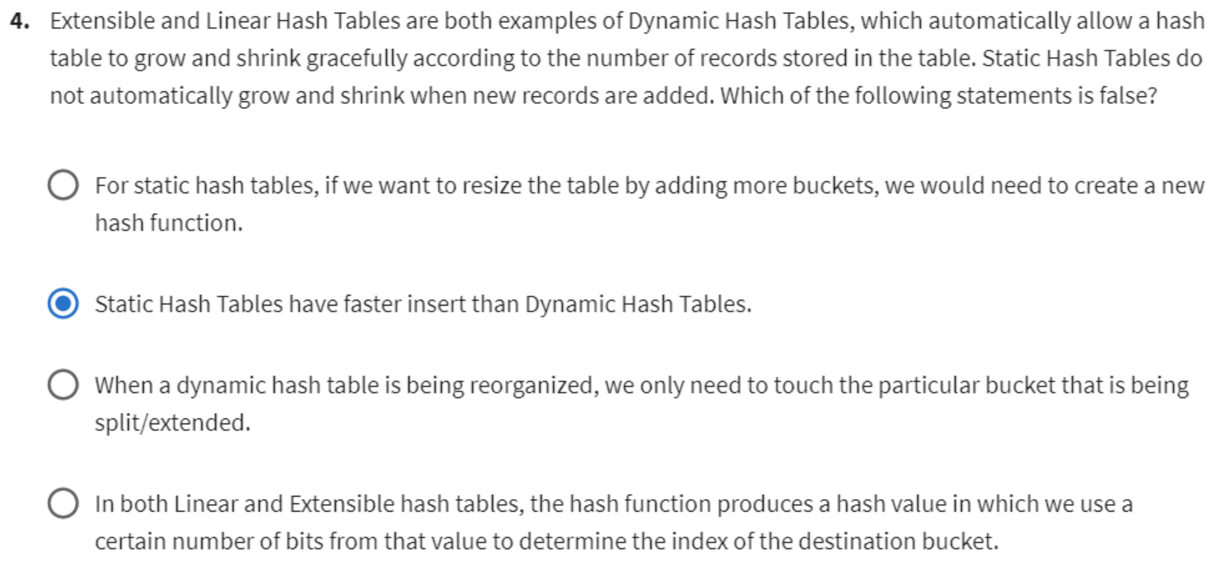


- A = 22 (1000), B = 35 (0111), C = 69 (1101), D = 81 (1011)
- [✓ A = 35 (0111), B = 22 (1000), C = 81 (1011), D = 69 (1101)]
- A = 69 (1101), B = 35 (0111), C = 35 (0111), D = 22 (1000)
- A = 81 (1011), B = 69 (1101), C = 35 (0111), D = 22 (1000)

Solution: B
Explanation:
22 mod 14 = 8 (1000) -> starts with [10]
69 mod 14 = 13 (1101) -> starters with [11]

35 mod 14 = 7 (0111) -> starts with [0]
81 mod 14 = 11 (1011) -> starts with [10]

According to the figure, B and C should both start with 2 bits of 10, D should start with 2 bits of 11, and A should start with 1 bit of 0. This option is the only one satisfying these conditions.



4. 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?

- ○ For static hash tables, if we want to resize the table by adding more buckets, we would need to create a new hash function.
- [✓ Static Hash Tables have faster insert than Dynamic Hash Tables.]
- ○ When a dynamic hash table is being reorganized, we only need to touch the particular bucket that is being split/extended.
- ○ 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.

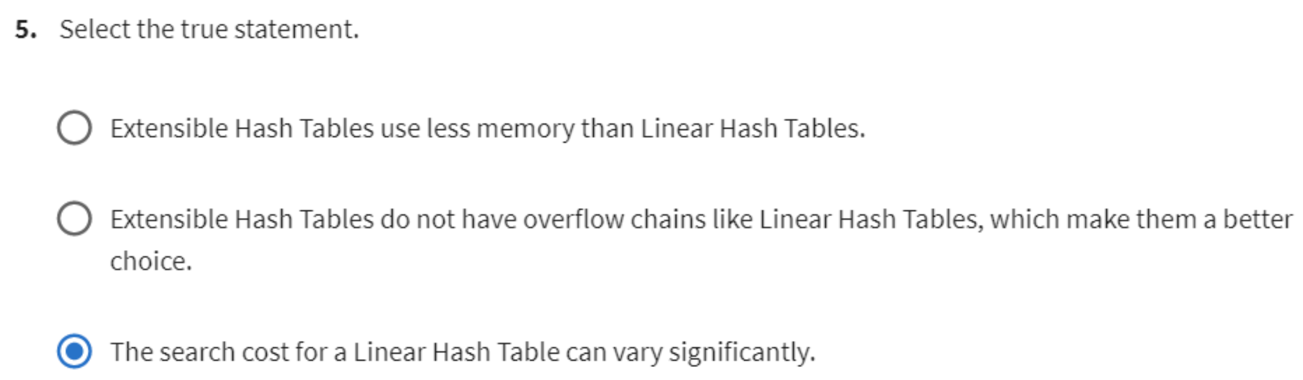
Solution: B
Explanation:

Option A: 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. So, this option is true.

Option B: 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. This option is false and thus the answer.

Option C: 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. So, this option is true.

Option D: In a linear hash table, we use the right (least significant) bits. In an extensible hash table, we use the left (most significant) bits. So, this option is true.



5. Select the true statement.

- ○ Extensible Hash Tables use less memory than Linear Hash Tables.
- ○ Extensible Hash Tables do not have overflow chains like Linear Hash Tables, which make them a better choice.
- [✓ The search cost for a Linear Hash Table can vary significantly.]

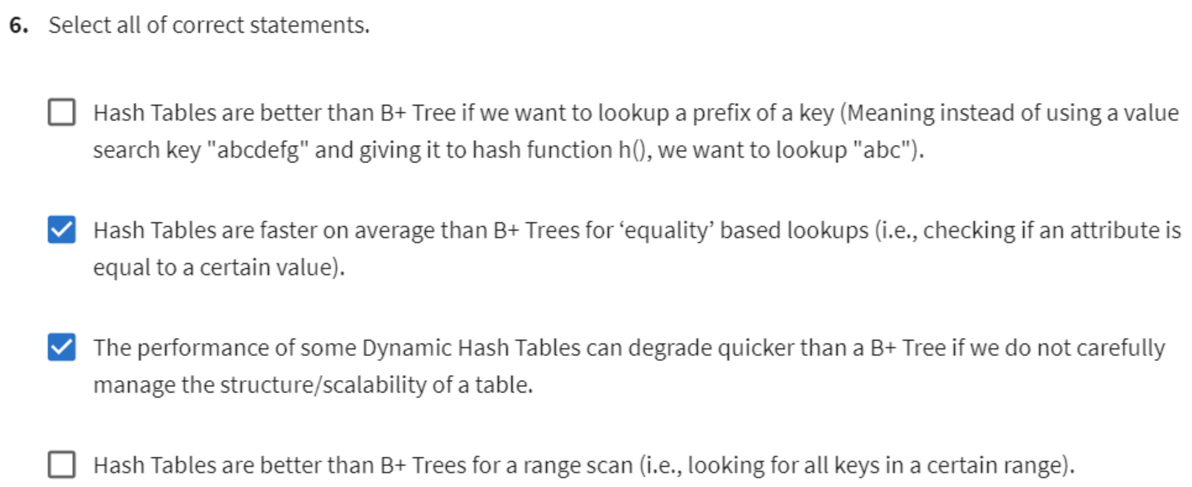
Solution: C

Explanation:

Option A: Extensible Hash Tables require more memory because there are no overflow blocks. This means that the entire directory needs to fit into the main memory, and after an extension, it may no longer even fit into the main memory. When memory is a problem, Linear Hash Tables are a better option, since they do not have a directory table.

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

Option C: 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. This option is correct.



6. Select all of correct statements.

☐ Hash Tables are better than B+ Tree if we want to lookup a prefix of a key (Meaning instead of using a value search key "abcdefg" and giving it to hash function h(), we want to lookup "abc").

[✓ 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).]

[✓ The performance of some Dynamic Hash Tables can degrade quicker than a B+ Tree if we do not carefully manage the structure/scalability of a table.]

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

Solution: BC

Explanation:

Option A: Lookup is based on a hash computation of the entire key, so they cannot use just the prefix. This option is incorrect.

Option B: If there is a hash created on the attribute we want to view, then we will likely have an O(1) lookup cost-- to simply follow the pointer given by the hash function. This option is correct.

Option C: If there are lots of collisions in linear hash tables, 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. This option is correct.

Option D: B+ Trees are better, because hash tables do not store the keys in any certain order, while B+ Trees do. This option is incorrect.