

Some Practice Problems on Hardware, File Organization and Indexing

Multiple Choice

State if the following statements are true or false.

1. On average, repeated random IO's are as efficient as repeated sequential IO's because random IO's tend to access different tracks and therefore cause less contention.
2. The LRU ("Least Recently Used") buffer replacement strategy is a good strategy for repeated scans of the same file.
3. The LRU ("Least Recently Used") strategy is a good strategy for B-trees.
4. Consider the relation $R(A,B,\dots)$, where the pair (A,B) is a key of R . Is it possible to build a multi-attribute index for the pair (A,B) without needing indirection from grid entries to buckets?
5. For any data file it is possible to create two separate sparse indices on different keys.
6. There are scenarios where it makes sense to construct a two-level index that has dense first level and a dense second level.
7. Linear hashing uses overflow buckets.
8. Since there is no downside to having indices, we may build an index on every column of every relation to speed up as many queries as possible.
9. Most commercial vendors have preferred to build B+ trees instead of hash indices because they are superior in equality lookups.
10. Consider the cost of fetching a random block from disk. The ratio (access cost/number of bytes) decreases as the block size increases.

Disk Stuff

Consider a disk with the following characteristics:

- Rotation Speed 3600RPM
- Average seek time 15ms
- 1024 sectors per track
- Gaps between sectors take no space
- Sector size 8Kb
- Block size 8Kb (1 sector)

Relation R is stored on the disk as a contiguous sequential file sorted on the key A . R has 2^{20} records of fixed size 1KB, of which attribute A uses 12bytes. We want to build a sequential sparse index file for A . Assume that the index will be placed in a file that will be located on tracks far from the tracks where R is stored. Consider two design choices for the sparse index:

1. Conventional sparse index: One entry for each block of R – as suggested in class.
2. Super-sparse index: One pointer per track. (Assuming, of course, that the OS gives us such detailed access.)

Assume that a pointer to a track requires 2 bytes.

Questions:

1. What is the size of the conventional sparse index?
2. What is the size of the super-sparse index?
3. We need to locate a record r whose A value is c . What is the average time needed to locate c using (a) the conventional sparse index and (b) the super-sparse index? Assume a "cold start", i.e., the buffers are empty.
- 4.

Storage

Sequential Vs Random Access

Consider a table R that is stored in consecutive pages of the disk. We know that reading pages of the disk in the sequential order in which they are stored is at least ten times faster than reading pages in some random order. Give two reasons why we may still prefer to read in random order instead of reading sequentially.

Indexing

Estimating the Cost of Operations on a Dense Index

Consider a relation $R(K,A,B,...)$ where the attribute K is an integer key. The relation is sorted along the key K , i.e., K is a primary key. Then there is a multilevel dense index on K . Assume that the top level of the multilevel index always has just one page.

Each tuple has a block pointer pointing to the next tuple. Also, each index entry has a pointer to the next entry. As usual, the system attempts to "pack" consecutive tuples in contiguous blocks and uses the overflow when this is not possible.

Assume that n tuples are inserted in R .

- The size of a disk page is d .
- The size of a tuple of R is r .
- The size of each K value is k bytes.
- Block pointers are used and each one is p bytes long.
- No use of main memory buffers.
- Records (tuples and index entries) do not span blocks.

Questions

1. Assume that initially the tuples with keys $m*i$, $i=1,...,n$ and the corresponding index entries are stored in consecutive blocks, no overflow is needed and space utilization is 66%. Display the state of the indexes and the relation assuming $p=6$, $r=44$, $k=20$, $d=160$, $m=1000$, $n=30$.

- Assume that at every minute j there's a burst of inserting 100 tuples with keys $m*j+k$, $k=1, \dots, l$ in ascending order. If you were the administrator how often would you reorganize this database?

B+ Trees (1)

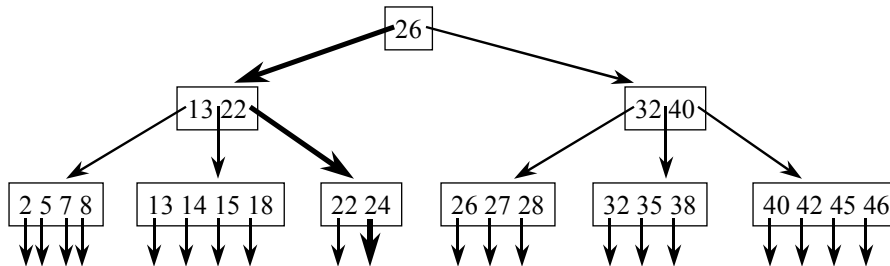
Consider the table $R(A,B,\dots)$. Suggest ways in which a B+ Tree can be used for multi-key indexing of the pair of attributes (A,B) .

B+ Trees (2)

Draw an example of a valid 3-level B+ Tree of order $n=2$ where inserting an entry with key 20 causes a split to a non-leaf node but does not increase the height of the tree.

B+ Trees (3)

Consider the following B+ Tree of order $n=4$.



Show “snapshots” of the B+ Tree as it evolves as a result of the following actions:

- Insert an entry with key 29.
- Insert an entry with key 30.
- Insert 44
- Insert 43
- Insert 41

The Weird Properties of C+ Trees

The C+ trees are a variation of B+ trees. They have the following properties:

- Each page/node of the C+ tree has m to $3m/2$ elements (m is even). The root may have any number of elements.
- $m > 2$
- has more than one levels
- All other properties of the C+ tree are identical with the B+ tree.

Write an $O(\log n)$ procedure for insertion of a new element e into a C+ tree.

Dynamic Hashing

1. Consider an extensible hash table where 200 buckets are actually allocated at this point. What is the size of the smallest possible directory at this point?
2. Consider an extensible hash table whose directory has 1024 entries. What is the largest number of entries that could point to the same bucket?
3. Consider a linear hash table with max used bucket 110 (in binary). Which bucket should a key with hash value 111 (binary) be sent to?

Extensible Hashing

Consider an extensible hash structure where buckets can hold up to three entries. Initially the structure is empty. The keys and their hash values are the following (don't ask what function produces such values☺)

A → 00001
 B → 00011
 C → 00110
 D → 01110
 E → 01111
 F → 10001
 G → 10101
 H → 10111
 I → 11000
 K → 11001
 L → 11101
 M → 11111

1. Suppose we insert the values in the order given above. Show the structure after all the values have been inserted.
2. Now assume we insert in the order F,E,D,C,B,A,M,L,K,I,H,G. Show the structure after all values have been inserted.

Linear Hashing

Consider a linear hash structure where buckets can hold up to two records. Initially the structure is empty. Assume that the utilization threshold value is 100%. The keys and their hash values are given above. Show the structure after all values have been inserted.

No More Indexes Please...

Provide reasons why we should **not** keep an index on an attribute A of some relation R.