More B-trees, Hash Tables, etc.

CS157B
Chris Pollett
Feb 21, 2005.

Outline

- B-tree Domain of Application
- B-tree Operations
- Hash Tables on Disk
- Hash Table Operations
- Extensible Hash Tables
- Multidimensional Indexes

Domain of Application of B-trees

- Can be used for search keys which are primary keys for the table whether or not the table is sorted on this key.
- Can be used on non-primary key fields.
- Can allow multiple occurrences of search key at the leaves.

Lookup- in B-Trees

For simplicity will assume no duplicate keys.

To search for a record with key K:

- If at a leaf, look among the keys, follow pointer K if exists to record.
- For an interior node, if K is less than the first key or greater than the last, follow the pointer to the left/right. If K_i <= K < K_{i+1} then follow the pointer to the right of K_i.

Range Queries

- B-Trees not only support single-valued queries, but can also support range queries like: SELECT * FROM R WHERE R.a > 10;
- To look up keys in a range [a,b], we look up a, then cycle through entries until we get a value larger than b.

Insertion into B-Trees

- First try to put key into the appropriate leaf, if there is room.
- If there is no room, we split the leaf node among two blocks, so that each is half full. We put a new (key, pointer) pair into the parent that selects between these two halves.
- If this propagates to the root and it is full, split the root into two and make a new root with a key to select between these two halves.

Deletion from B-Trees

- Locate the record and its key pointer pair in a leaf in B-tree.
- If the leaf node is now less than half full and either sibling would be more than half full after moving a key, pointer pair. More the appropriate end value pair from the sibling into the current node. Adjust parent node.
- If neither, sibling can do this then combine this leaf with one of its two siblings and propagate the results up to parent.

It should be noted some DBs just use tombstones rather than bother with deletion.

Efficiency of B-Trees

- Suppose a block can store 340 key-pointer pairs and on average is around 3/4 full. So around 255 pointer in any block index in use.
- In a three level tree the root has 255 children, 255^2 grandchildren and 255^3 records pointer to at the leaves. That is, around 16.6 million records.
- Since one typically caches at least the root, and often the next level as well, this means in one or two disk I/Os can find any record among 16.6 million records.

Secondary-Storage Hash Tables

Assume have seen main memory hashing before…

- Set-up is have a hash function h(K) mapping keys into buckets 0, …, B-1 and have some scheme to resolve collisions.
- If hash table is in secondary storage, then the buckets are mapped to blocks.
- If a block has too many records in it then use *overflow blocks* to handle any additional records hashing to this value.

Insertion, Deletion and Efficiency

- To insert we compute h(K). If the labelled bucket h(K) still has room then we add a key pointer pair there. If not, we create an overflow bucket.
- To delete we compute h(K), then search that bucket and any overflow bucket for the record pointer to the record to be deleted. Might consolidate blocks if some blocks becomes empty.
- Ideally, only one disk I/O is needed to find record pointer we want. However, as the file grows we will tend to use overflow blocks more and more.

Extensible Hash Tables

- Want to dynamically increase the number of buckets to avoid having to do more than one I/O.
- To do this we add an indirection level (which might be cached in memory), which is an array of pointers 0 to 2^i -1 for some i. Some of these pointers might point to same block.
- Each block records how many bits of hash used to get to it.
- Hash computes a k bit number for some k >= i. Alternatively, could have a have function h(K,i) where i says number of bits we want.
- To look up K. Compute h(K) using at first i bits. Look up corresponding indirection pointer follow to bucket.

Insertion into Extensible Hash Tables

1. Computer h(K) look at first i bits, say equals x.
2. Look up xth entry in indirection table, follow to bucket. If there is room insert into bucket.
3. If the number of bits of hash value used to look up bucket way less than i, say j, divide entries in block x into two blocks according to the j+1st bit of the hash function.
4. If j=i then we double the number of indirection pointers. That is set i:=i+1. Each old pointer on i bits now becomes two pointers on i+1 bits. Now we can go to 3.

Linear Hash Tables

- Extensible hashing can be inefficient if number of records is small, if have to insert a lot and update pointers, or if after double the indirection layer can no longer fit in memory.
- Another technique, linear hashing, always keeps the number of buckets n so that 80% of each bucket is filled.
  - Overflow blocks are permitted but on average there will be much fewer than 1 such block per bucket.
  - [log n] bits are needed to number entries of bucket array.
  - When we insert we check if adding the record makes the ratio of number of records to blocks is such that it is expected that more than 80% of each block is filled. If yes, we increase the number of blocks, by one by splitting the block that the record would be added to. We use the most significant bit of the hash function to distinguish between these two blocks.
  - If n exceeds to 2^i then use another bit of hash.

Multidimensional Indexes

- All indexes considered so far are on one field. Say salary.
- We now consider indexes on more than one field. Say first name, last name.
- There are several techniques for such indexes: grid-files, partitioned hash functions, multiple key indexes, kd-trees, Quad trees, and R-trees.