CS562: Database Management Systems II Notes

Steven DeFalco

Spring 2024

Contents

1 Indexing and Hashing

Indexing methods are used to speed up access to desired data. A ***search key*** is an attribute used to look up records in a file. An ***index file*** consists of records (called **index entries**) of the form. Index files are typically much smaller than the original file. There are two basic kinds of indices:

- **Ordered indices**: search keys are stored in sorted order
- **Hash indices**: search keys are distributed uniformly across *buckets* using a *hash function*

Remark When *optimizing* in database queries, the goal is to have a small search space.

In an ***ordered index***, index entries are stored sorted on the search key value. In a sequentially ordered file, the ***primary index*** is the index whose search key specifies the sequential order of the file; the search key of a primary index is usually but not necessarily the primary key. The ***secondary index*** is an index whose key specifies an order different from the sequential order of the file (also called non-clustering index). An ***index-sequential file*** is an ordered sequential file with a primary index.
A ***dense index*** is where an index record appears for every search-key value in the file. ***Sparse index*** contains index records for only some search-key values. To locate a record with search-key value K we:

1. Find index record with largest search-key value $< K$
2. Search file sequentially starting at the record to which the index record points

Remark Sparse index must be primary index

Compared to dense indices, sparse indices have less space and less maintenance overhead for insertions and deletions. However, they are generally slower than dense index for locating records. a record with search-key value K we:

1. Find index record with largest search-key value $< K$
2. Search file sequentially starting at the record to which the index record points

Remark Sparse index must be primary index

Compared to dense indices, sparse indices have less space and less maintenance overhead for insertions and deletions. However, they are generally slower than dense index for locating records.

If primary index does not fit in memory, access becomes expensive. A solution to this is to treat primary index kept on disk as a sequential file and construct

a sparse index on it. The outer index is a sparse index of primary index. The inner index is the primary index file. If even out index is too large to fit in main memory, yet another level of index can be created and so on. Indices at all levels must be updated on insertion or deletion from the file.

1.1 Index Updates

1.1.1 Deletion

If the deleted record was the only record in the file with its particular search-key value, the seark-key is deleted from the index also. For singl e-level index deletion.

- Dense indices—deleetion of search-key
- Sparse indices—
  - If an entry for the search key exists in the index, it is deleted by replacing the entry in teh index with the next search-key value in the file
  - If the next search-key value already has an index entry, the entry is deleted instead of being replaced

1.1.2 Insertion

In single-level index insertion:

- Perform a lookup using the searh-key value appearing in the record to be inserted.
- Dense indices—if the search-key value does not appear in the index, insert it
- sparse indices—if index stores an entry for each block of the file, no chnage needs to be made to the index unless a new block is created. If a new block is created, the first search-key value appearing in the new block is inserted into the index.

In multilevel insertion, algorithms are simple extensiosn of the single-level algorithms.
Indices offer substantial benefits when searching for records, but updating indices imposes overhead on database modification—when a file is modified, every index on the file must be updated. Sequential scan using primary index is efficient, but a sequential scan using a second index is expensive.

2 B+ Tree Index Files

B^+-tree indices are an alternative to indexed-sequential files. Some disadvantages of indexed-sequential files are that performance degrades as the file grows (since many overflow blocks get created) and that periodic reorganization of the entire file is required. Advantages of B^+-tree index files are:

- automatically reorganizes itself with small, local changes in the face of insertions and deletions
- Reorganization of entire file is not required to maintain performance

One minor disadvantage of B^+-trees is the extra insertion and deletion overhead (space overhead); however, the advantages outweigh the disadvantages and thus this structure is used extensively.

A B^+-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children
- A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values
- Special cases...
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n-1)$ values.

2.1 Types of nodes in B^+-trees

The following are properties of a **leaf node**:

- For $i = 1, 2, \ldots, n-1$, pointer P_i either points to a file record with search-key value K_i, or to a bucket of pointers to file records, each record having search-key value K_i. Only need bucket structure if search-key does not form a primary key.
- If L_i, L_j are leaf nodes and $i < j$, L_j's search-key values are less than L_j's search-key values
- P_n points to next leaf node in search-key order

Nonleaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:

- All the search-keys in the subtree to which P_1 points are less than K_1
- For $2 \leq i \leq n-1$, all the search-keys in the substree to which P_i points have values greater than or equal to K_{i-1} and less than K_i

- All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}

Since the inter-node connections are done by pointers, *logically* close blocks need not be *physically* close. The non-leaf levels of the B$^+$-tree form a hierarchy of sparse indices. The B$^+$-tree contains a relatively small number of levels; level below root has at least $2 \times \lceil n/2 \rceil$ values. If there are K search-key values in the file, the tree height is no more than $\left\lceil \log_{\lceil n/2 \rceil}(K) \right\rceil$. Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time.

2.2 Queries on a B$^+$-tree

Find all records of a search-key value of k.

1. $N =$ root
2. Repeat
   (a) Examine N for the smallest search-key value $> k$.
   (b) If such a value exists, assume it is K_i. THen set $N = P_i$
   (c) Otherwise $k \geq K_{n-1}$. Set $N = P_n$

   Until N is a leaf node
3. If for some i, key $K_i = k$ follow pointer P_i to the desired record or bucket.
4. Else no record with search-key value k exists

2.3 Updates on B$^+$-trees

2.3.1 Insertion

To perform insertion on a B$^+$-tree...

1. Find the leaf node in which the search-key value would appear
2. If the search-key value is already present in the leaf node
   (a) Add record to the file
   (b) If necessary add a pointer to the bucket
3. If the search-key value is not present, then
   (a) Add the record to the main file (and create a bucket if necessary)
   (b) If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
   (c) Otherwise, split the node (along with new (key-value, pointer) entry)

To split a leaf node...

- Take the n (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
- let the new node by p, and let k be the least key value in p. Insert (k, p) in the parent of the node being split.
- If the parent is full, split it and **propagate** the split further up.

Splitting of nodes proceeds upwards until a node that is not full is found. In the wrost case, the root node may be split increasing the height of the tree by 1.

To split a non-leaf node: when inserting (k, p) into an already full internal node N

- Copy N to an in-memory area M with space for $n+1$ pointers and n keys
- Insert (k, p) into M
- Copy from M back into node N
- Copy from M into newly allocated node N'
- Insert $(K_{\lceil n/2 \rceil}, N')$ into parent N

2.3.2 Deletion

To perform a deletion in a B^+-tree...

- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then *merge siblings*:
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node
  - Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.
- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then *redistribute pointers*:
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries

  - Update the corresponding search-key value in the parent of the node

- The node deletions may cascade upwards until a node which has $\lceil n/2 \rceil$ or more pointers is found
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

2.3.3 Indexing

Variable lenght strings are used as the keys: use space utilization as criterion for splitting, not number of pointers. ***Prefix compression*** is that key values at internal nodes can be prefixes of the full key. For this, must keep enough chraracters to distinguish entries in the subtrees separated by the key value. Keys in the leaf nodes can be compressed by sharing common prefixes.

2.3.4 B-Tree Index Files

B-Tree index files are similar to B^+-trees, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys. Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included. Some advantages of B-tree indices are:

- May use less tree nodes than a corresponding B^+-tree
- Sometimes possible to find search-key value before reaching leaf node

Some disadvantages of B-tree indices are:

- Only small fraction of all search-key values are found correctly
- Non-leaf nodes are larger, so fan-out is reduced. Thus, B-trees typically have greater depth than corresponding B^+-tree
- Insertion and deletion are more complicated than in B^+-trees
- Implementation is harder than B^+-trees

Typically, the advantages of B-Trees do not out weigh the disadvantages.

2.4 Multiple-Key Access

Multiple-key access uses multiple indices for certain types of queries.