Indexing and Hashing

- Cost estimation
- Basic Concepts
- B⁺-Tree Index Files
- Hashing
- Comparison of Ordered Indexing and Hashing

by Paul L. Bergstein @ UMass Dartmouth

Estimating Costs

- For simplicity we estimate the cost of an operation by counting the number of blocks that are read or written to disk.
- We ignore the possibility of blocked access which could significantly lower the cost of I/O.
- We assume that each relation is stored in a separate file with B blocks and R records per block.

Basic Concepts

- Indexing is used to speed up access to desired data.
  - E.g. author catalog in library
- A search key is an attribute or set of attributes used to look up records in a file. Unrelated to keys in the db schema.
- An index file consists of records called index entries.
- An index entry for key k may consist of
  - An actual data record (with search key value k)
  - A pair (k, rid) where rid is a pointer to the actual data record
  - A pair (k, bid) where bid is a pointer to a bucket of record pointers
- Index files are typically much smaller than the original file if the actual data records are in a separate file.
- If the index contains the data records, there is a single file with a special organization.

Index Evaluation Metrics

- Access time for:
  - Equality searches – records with a specified value in an attribute
  - Range searches – records with an attribute value falling within a specified range.
- Insertion time
- Deletion time
- Space overhead

Types of Indices

- The records in a file may be unordered or ordered sequentially by some search key.
- A file whose records are unordered is called a heap file.
- If an index contains the actual data records or the records are sorted by search key in a separate file, the index is called clustering (otherwise non-clustering).
- In an ordered index, index entries are sorted on the search key value. Other index structures include trees and hash tables.
- A primary index is an index on a set of fields that includes the primary key. Any other index is a secondary index.

B-Trees

- B-Trees are balanced search trees
  - They are designed to be stored on external storage, e.g., magnetic disks
  - They are designed to minimize disk accesses
  - They are widely used in database systems
- B⁺-Tree is a popular variant of the original B-Tree
  - The keys are stored in the leaves
B⁺-Tree Index Files

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 where \( n \) is the maximum number of pointers per node.
- 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 \( n \) values.

B⁺-Tree Node Structure

- A typical node has tens to hundreds of elements

\[
P_1 \quad K_1 \quad P_2 \quad \ldots \quad P_{n-1} \quad K_n \quad P_n
\]

- \( K_i \) are the search-key values.
- \( P_i \) are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

\[
K_1 < K_2 < K_3 < \ldots < K_n
\]

Leaf Nodes in B⁺-Trees

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 \( K_i \). If \( L_i \) and \( L_j \) are leaf nodes and \( i < j \), \( L_i \)'s search-key values are less than \( L_j \)'s search-key values.
- \( P_n \) points to next leaf node in search-key order.

Non-Leaf Nodes in B⁺-Trees

- Non-leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with \( m \) pointers:

\[
P_1 \quad K_1 \quad P_2 \quad \ldots \quad P_{m-1} \quad K_m \quad P_m
\]

- All the search-keys in the subtree to which \( P_i \) points are less than \( K_i \).
- All the search-keys in the subtree to which \( P_i \) points are greater than or equal to \( K_i \).

Examples of a B⁺-tree

- B⁺-tree for account file (n=3)

- B⁺-tree for account file (n=5)

Examples of a B⁺-tree

- Leaf nodes must have between 2 and 4 values (\( (n-1)/2 \) and \( n-1 \), with \( n=5 \)).
- Non-leaf nodes other than root must have between 3 and 5 children (\( n/2 \) and \( n \) with \( n=5 \)).
- Root must have at least 2 children.
Observations about $B^+$-trees

- Since the inter-node connections are done by pointers, there is no assumption that in the $B^+$-tree, the "logically" close blocks are "physically" close.
- The $B^+$-tree contains a relatively small number of levels (logarithmic in the size of the main file), thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see).

Queries on $B^+$-Trees

- Find all records with a search-key value of $k$.
  » Start with the root node
  » Examine the node for the smallest search-key value $> k$.
  » If such a value exists, assume it is $K_m$. Then follow $P_m$ to the child node
  » Otherwise $k = K_m$, where there are $m$ pointers in the node, then follow $P_m$ to the child node.
  » If the node reached by following the pointer above is not a leaf node, repeat the above procedure on the node, and follow the corresponding pointer.
  » Eventually reach a leaf node. If key $K_k = k$, follow pointer $P$ to the desired record or bucket. Else no record with search-key value $k$ exists.

Queries on $B^+$-Trees (Cont.)

- In processing a query, a path is traversed in the tree from the root to some leaf node.
- If there are $K$ search-key values in the file, the path is no longer than $\log_{\log_2} K$.
- A node is generally the same size as a disk block, typically 4 kilobytes, and $n$ is typically around 200 (20 bytes per index entry).
- With 1 million search key values and $n = 200$, at most $\log_{\log_2}(1,000,000) = 3$ nodes are accessed in a lookup.
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup.
  » above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds!

Updates on $B^+$-Trees: Insertion (Cont.)

- Splitting a node:
  » take the $n$ (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $n/2$ in the original node, and the rest in a new node.
  » Let the new node be $p$, 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.
- The splitting of nodes proceeds upwards till a node that is not full is found. In the worst case the root node may be split increasing the height of the tree by 1.
Updates on B+–Trees : Deletion

- 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 don’t fit into a single node, then
  - 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 (Kv, P), where P is the pointer to the deleted node, from its parent, recursively using the above procedure.

Examples of B+–Tree Deletion

Result after deleting “Downtown” from account

- The removal of the leaf node containing “Downtown” did not result in its parent having too little pointers. So the cascaded deletions stopped with the deleted leaf node’s parent.

B+–Tree File Organization

- Index file degradation problem is solved by using B+–Tree indices, Data file degradation problem is solved by using B– Tree File Organization.
- The leaf nodes in a B+–tree file organization store records, instead of pointers.
- Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Leaf nodes are still required to be half full.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B+–tree index.
- Good space utilization is important since records use more space than pointers. To improve space utilization, involve more sibling nodes in redistribution during splits and mergers.

Updates on B+–Trees : Deletion

- Otherwise, if the node has too few entries due to the removal, and the entries in the node and a sibling don’t fit into a single node, then
  - Redistribute the pointers between the node and a sibling such that both have at least the minimum number of entries
  - Update the corresponding search-key value in the parent of the node.
- The node deletions may cascade upwards till a node which has \[\lfloor n/2 \rfloor\] 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.

Examples of B+–Tree Deletion (Cont.)

Deletion of “Perryridge” instead of “Downtown”

- The deleted “Perryridge” node’s parent became too small, but its sibling did not have space to accept one more pointer, so redistribution is performed. Observe that the root node’s search-key value changes as a result.

Static Hashing

- A bucket is a unit of storage containing one or more records (a bucket is typically a disk block). In a hash file organization we obtain the bucket of a record directly from its search-key value using a hash function.
- Hash function h is a function from the set of all search-key values K to the set of all bucket addresses B.
- Hash function is used to locate records for access, insertion, and deletion.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.
Hash Functions

- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is uniform, i.e., each bucket is assigned the same number of search-key values from the set of all possible values.
- Ideal hash function is random, so each bucket will have the same number of records assigned to it irrespective of the actual distribution of search-key values in the file.
- Typical hash functions perform computation on the internal binary representation of the search-key. For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo number of buckets could be returned.

Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation. A hash index organizes the search keys, with their associated record pointers, into a hash file structure.
- Hash indices are always secondary indices — if the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary. However, the term hash index is used to refer to both secondary index structures and hash organized files.

Example of Hash Index

Deficiencies of Static Hashing

- In static hashing, function h maps search-key values to a fixed set B of bucket addresses.
  - Databases grow with time. If initial number of buckets is too small, performance will degrade due to too much overflows.
  - If file size at some point in the future is anticipated, and number of buckets allocated accordingly, significant amount of space will be wasted initially.
  - If database shrinks, again space will be wasted.
  - One option is periodic, re-organization of the file with a new hash function, but it is very expensive.
- These problems can be avoided by using techniques that allow the number of buckets to be modified dynamically.

Dynamic Hashing

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically

  **Extendable hashing** — one form of dynamic hashing
  - Hash function generates values over a large range — typically
  - 32-bit integers, with \( b = 32 \).
  - At any time use only the last \( k \) bits of the hash function to index into a table of bucket addresses, where: \( 0 \leq i < 2^k \)
  - Initially \( i = 0 \)
  - Value of \( i \) grows and shrinks as the size of the database grows and shrinks.
  - Actual number of buckets is \( 2^i \), and this also changes dynamically due to merging and splitting of buckets.

General Extendable Hash Structure

In this structure, \( i_2 = i \), whereas \( i_1 = i - 1 \).
Index Definition in SQL

- Create an index
  
  ```sql
  create index <index-name> on <relation-name>
  (<attribute-list>)
  ```
  
  E.g.: `create index b-index on branch(branch-name)`

- Use `create unique index` to indirectly specify and enforce the condition that the search key is a candidate key.

- To drop an index
  
  ```sql
  drop index <index-name>
  ```