01:XXX:XXX - Homework n

Pranav Tikkawar

December 19, 2025

Contents

$\boxed{\text{Things to review: Extenible Hashing, Linear Hashing}}$

1 Index

Definition (Indexing)**.** Indexing mechanism used to speed up access to desired data

Definition (Search Key)**.** Attribute to set of attributes used to look up records in a file.

Definition (Index File)**.** File containing index entries that map search key values to record locations in a data file. In the form of [search key value, pointer to record]

Remark. Two basic kinda of indices:

- Ordered Index: Search keys are stored in sorted Order
- Hash Index: Search keys are distributed uniformly across buckets using a hash function

Remark (Index Evaluation Metrics)**.**

- Access types suppeorted efficiently
- Access Time
- Insertion Time
- Deletion Time
- Storage Overhead

Definition (Ordered Index)**.** Index entries are stored sorted on the search ket value

Definition (Clustered Index)**.** In a sequentially ordered file he index whos search key specifies the sequential order of the data file. Primary index.

Definition (Secondary Index)**.** An index whose search key does not specify the sequential order of the data file. Unclustered index.

Definition (Dense Index)**.** An index in which there is an index entry for every search key value in the data file. EX Index on ID attribute of instructor relation

Definition (Sparse Index)**.** An index in which index entries are created only for some of the search key values in the data file. EX Index on department attribute of instructor relation

Remark (Clustering Vs Non Clustering)**.** Indices provide benefits when searching for records, but they impose overhead on db modifications like insertions, deletions and updates. Sequential Scan using cluseting index is efficent but a sequential scan using a non clustering index is expensive on magnetic disks since it requires many random I/O operations to fetch the data records.

Remark (Multilevel Indices)**.** If index does not fit in memory, access becomes expensive. To solve this problem, treat index kept on disk as a sequential file and build a sparse index on it. Outer index is built on the first level index, and inner index is built on the data file. This can be repeated to form multiple levels of indices until the top level index fits in memory.

2 Hash

Definition (Hash Table Index)**.** Index entries are stored in a hash table structure using a hash function on the search key value to determine the location of each index entry. Tradeoff between fast vs collision rate.

Definition (Hash Function)**.** For any input key k, a hash function h(k) computes an integer in the range 0 to N-1, where N is the number of buckets in the hash table. We dont want a cryptographic hash function, just a uniform distribution.

Remark (Hash Index Variants)**.**

- Static Hashing: Number of buckets is fixed
  - Linear probing
  - Cuckoo Hashing
  - Chained Hashing
- Dynamic Hashing: Number of buckets can grow and shrink dynamically as the number of records in the file grows and shrinks
  - Extendible Hashing
  - Linear Hashing

Definition (Linear Probing)**.** Single giant table of slots. On collision, search sequentially for next free slot.

Definition (Chained Hashing)**.** Each bucket contains a pointer to a linked list of entries that hash to the same bucket. on collision, add to linked list. Potential for infinite growth.

Definition (Cuckoo Hashing)**.** Use multiple hash tables with different hash functions. On insert, check every table and pick anyone empty. If none empty, evict existing key and reinsert it into its alternative location. Repeat until all keys placed. O(1) lookup time and deletions. If we get into infinite loop, rehash all keys into larger tables, with new hash functions.

Definition (Extendible Hashing)**.** Use a directory of pointers to buckets. Directory size is a power of 2. Each bucket has a local depth, directory has global depth. On collision, if local depth < global depth, split bucket and update directory pointers. If local depth = global depth, double directory size, increment global depth, then split bucket. $\boxed{\text{Look into Later}}$

Definition (Linear Hashing)**.** Maintain a pointer that tracs the next bucket to split, when any bucker overflow split the bucket the pointer location, overflow criterion based on implimentation. Use a family of hash functions, N inital buckets, h_0 is some hash function mod N, h_i is h mod 2^i * N. Processes in rounds, current round number is level. there are 2^{level} * N buckets. When splitting a bucket, rehash all its entries using $h_{level+1}$. After splitting, increment pointer. If pointer reaches 2^{level} * N, reset pointer to 0 and increment level. $\boxed{\text{Look into Later}}$

3 B+ Trees

Definition (Tree)**.** A data strucutre that represents heierarchical relationships between items of data. Consists of nodes, with a root node at the top, internal nodes in the middle and leaf nodes at the bottom.

Definition (Binary Tree)**.** A tree where each ndoe has at most two children. and a BST (Binary Search Tree) is a binary tree where for each node, all values in the left subtree are less than the node's value, and all values in the right subtree are greater than the node's value.

Remark (Height and Complexity)**.** N is the number of nodes in the tree, h is the height of the tree. Best case height: $log_2(N)$ (perfectly balanced tree) Worst case height: N (completely unbalanced tree) Solution is a balanced BST, with special rotation operations to maintain balance during insertions and deletions. Examples: AVL Trees, Red Black Trees.

Remark (RAM vs Disk speed for Tree)**.** For acessesing data in RAM, binary search tree is fine. For accessing data on disk, we want to minimize number of disk I/O operations.

Definition (M-Way Tree)**.** We can have $M-1$ keys and M children per node. This reduces height of tree, and thus number of disk I/O operations.

Definition (B+ Tree)**.** A balanced M-Way tree with a parameter n (order of the tree) that satisfies the following properties:

- All paths from root to leaf are of the same length
- Non leaf nodes (except root) have between $\lceil n/2 \rceil$ and n children
- Leaf nodes contain between $\lceil (n-1)/2 \rceil$ and $n-1$ keys
  - If a root is not a leaf, it has at least 2 children
  - if the root is a leaf, it can have between 1 and $n-1$ keys
- For a non leaf node with k children, it contains $k-1$ keys that act as separation values to direct the search
- All leaf nodes are linked together in a linked list for efficient range queries

Remark (Insertion in B+ Tree)**.** Always insert in leaf node. If leaf node has space, insert key in sorted order. If leaf node is full, split the node into two nodes, each with $\lceil n/2 \rceil$ keys, and promote the middle key to the parent node. If parent node is full, repeat the process up to the root.

Remark (Deletion in B+ Tree)**.** Always delete from leaf node. If leaf node has enough keys after deletion, done. If leaf node has too few keys, try to borrow a key from a sibling node. If sibling node has enough keys, move a key from parent to leaf node and move a key from sibling to parent. If sibling node also has too few keys, merge leaf node with sibling and move a key from parent to the merged node. If parent node has too few keys, repeat the process up to the root.

4 Bloom Filters

Definition (Bloom Filter)**.** A space-efficient probabilistic data structure used to test whether an element is a member of a set. It can yield false positives but not false negatives. ie it may return true even if the element is not in the set, but if it returns false, the element is definitely not in the set. Represented by a bitmap. It has m bits, k hash functions, and n inserted elements.

Remark (Bloom Filter Formula)**.** Have a Bit array of m items, all set to 0. To add an item, compute k independent hash functions, each producing a number between 0 and m-1. Set the bits at all k positions to 1. To check if an item is in the set, compute the k hash functions and check the bits at those positions. If any of the bits is 0, the item is definitely not in the set. If all bits are 1, the item may be in the set.

Remark (Probability)**.** After inserting n elements, with k hash functions, and m bits, the probability that a specific bit is still 0 is:

$$\begin{aligned} P(bit\ is\ 0) &= \left(1 - \frac{1}{m}\right)^{kn} \\ &\approx e^{-\frac{kn}{m}} \end{aligned}$$

The probability of a false positive is:

$$\begin{aligned} P(false\ positive) &= (1 - P(bit\ is\ 0))^k \\ &\approx \left(1 - e^{-\frac{kn}{m}}\right)^k \end{aligned}$$

Remark (Optimizing k)**.** We want to minimize the false positive rate.

$$\begin{aligned} FPR(k) &= \left(1 - e^{-\frac{kn}{m}}\right)^k \\ \ln(FPR(k)) &= k \ln\left(1 - e^{-\frac{kn}{m}}\right) \\ \frac{d}{dk}\ln(FPR(k)) &= \ln\left(1 - e^{-\frac{kn}{m}}\right) + k \cdot \frac{e^{-\frac{kn}{m}} \cdot \frac{n}{m}}{1 - e^{-\frac{kn}{m}}} \\ \text{Set } \frac{d}{dk}\ln(FPR(k)) &= 0 \\ k &= \frac{m}{n}\ln 2 = .7\frac{m}{n} \end{aligned}$$

5 Sorting And Aggregation

Definition (External Merge Sort)**.** Sorting algorithm used to sort large datasets that do not fit in memory. It works by dividing the dataset into smaller chunks that fit in memory, sorting each chunk, and then merging the sorted chunks together.

Sorting phase: Divide the dataset into chunks that fit in memory, sort each chunk, and write the sorted chunks back to disk.

Merging phase: Read the first element from each sorted chunk, write the smallest element to the output file, and read the next element from the chunk from which the smallest element was read. Repeat until all elements are written to the output file.

Definition (2-way External Merge Sort)**.** Pass 0: Reads every B pages of the table intro memeory, sorts them and writes them back to disk.

Pass 1 2 3 ...: recursivly merges pairs of sorted runs from disk Ueses 3 buffer pages (2 for input, 1 for output)

Requited 1 + log2(N) passes, with 2N * number of passes I/O operations

Definition (General External Merge Sort)**.** Pass 0: USe B buffer Pages, produce N/B sorted runs on disk of size B pages each.

Pass 1,2,3 ...: Merge $B - 1$ sorted runs i K-way merge

required $1 + \log_{B-1}(N/B)$ passes, with 2N * number of passes I/O operations

Definition (K-way Merge)**.** Input K sorted sub arrays, takes $O(N \log_2 K)$ time to merge them into a single sorted array.

Definition (Aggregation)**.** Aggregation is the process of combining multiple records into a single record based on a common attribute. Common aggregation functions include sum, average, count, min, and max.

Definition (Hashing Aggregation)**.** Use a hash function to distribute the records into buckets based on the common attribute. Perform aggregation within each bucket.

Partition Phase: Divide tuples into buckets using a hash function.

Rehash: Fo each parition on disk, dread to memory build in memory hash table on the second hash function, then go through each bucked of this table to bring together the records that have the same hash value.

6 Join Algorithms

Definition (Join)**.** Join is an operation that combines rows from two or more tables based on a related column between them. The result of a join is a new table that contains the combined data from the input tables.

Definition (Nested Loop Join)**.** For each tuple in the first table, scan the second table to find matching tuples. This is a simple but inefficient algorithm for large datasets.

Definition (Index Nested Loop Join)**.** Use an index on the second table to speed up the join operation. This can significantly reduce the number of disk I/O operations required for the join.

Definition (Sort Merge Join)**.** First, sort both tables on the join attribute. Then, use a merge algorithm to combine the sorted tables. This is more efficient than nested loop join for large datasets.

Definition (Hash Join)**.** Use a hash function to distribute the records from both tables into buckets based on the join attribute. Perform a nested loop join within each bucket.

7 Final

1 Index Structures

- a True
- b True
- c True (Remember to look what Secondary indexes are)
- d False (composite index is not efficent to search only on ID)

2 Linear Hashing ***

- a ?
- b Yes
- c Yes
- d No
- e ?

3 Extendible Hashing

- a No
- b Yes
- c No
- d Yes
- e Yes

4 B+ Tree

- a 1
- b 11* 18* 31*
- c [Learn bulk loading]

5 Sorting and Aggregation