Database Management Systems

Database Indexing

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
and
Centre for Artificial Intelligence and Machine Learning
Indian Statistical Institute, Kolkata

March, 2022

Motivation

Table: Bhatnagar

ID	Year	Category	Recipients
1	2017	Mathematical Sciences	0
2	2017	Chemical Sciences	1
3	2017	Engineering Sciences	2
4	2018	Mathematical Sciences	1
5	2018	Chemical Sciences	1
6	2018	Engineering Sciences	1
7	2019	Mathematical Sciences	2
8	2019	Chemical Sciences	2
9	2019	Engineering Sciences	1

Motivation

$\text{Bhatnagar.ID} = 7$

Motivation

Bhatnagar.ID = 7

Bhatnagar.Category = 'Mathematical Sciences'

Motivation

Bhatnagar.ID = 7

Bhatnagar.Category = ‘Mathematical Sciences’

Bhatnagar.Recipients >= 1

Motivation

Bhatnagar.ID = 7

Bhatnagar.Category = ‘Mathematical Sciences’

Bhatnagar.Recipients >= 1

Bhatnagar.ID = 7 and Bhatnagar.Category = ‘Mathematical Sciences’ and Bhatnagar.Recipients >= 1

Motivation

Bhatnagar.ID = 7

Bhatnagar.Category = ‘Mathematical Sciences’

Bhatnagar.Recipients >= 1

Bhatnagar.ID = 7 and Bhatnagar.Category = ‘Mathematical Sciences’ and Bhatnagar.Recipients >= 1

Note: Indexing helps if we can differentiate between full table scans and immediate location of tuple.

Database functionalities

These are the functionalities we need to support for a database access.

- Scan
- Point Search (Searching for equality of elements)
- Range Search (Searching for elements within a range)
- Insert
- Delete

Access methods

1. **Heap file:** Unordered, typically implemented as a linked list of pages.
2. **Sorted file:** ordered records, expensive to maintain.
3. **Index file:** data + additional structures around to quickly access data.
   - It might contain data (primary index)
   - It might contain pointers to the data (often stored in a heap file or secondary index)
   - It might be clustered (data sorted in the same order of the field as a clustered index)

Type of indexes:

- Search Trees
- Hash Table

Search trees

A majority of the tree operations (search, insert, delete, etc.) will require $O(\log_2 n)$ disk accesses where n is the number of data items in the search tree.

The main challenge is to reduce the number of disk accesses for processing per data item.

An m-ary search tree allows m-way branching. As branching increases, the depth decreases. A complete binary tree has a height of $\lceil \log_2 n \rceil$ but a complete m-ary tree has a height of $\lceil log_m n \rceil$.

Characteristics of B-Trees

B-Tree is a low-depth self-balancing tree. The height of a B-Tree is kept low by putting maximum possible keys in a B-Tree node.

The B-Trees have a higher branching factor (also termed as the order) to reduce the depth.

Characteristics of B-Trees

B-Tree is a low-depth self-balancing tree. The height of a B-Tree is kept low by putting maximum possible keys in a B-Tree node.

The B-Trees have a higher branching factor (also termed as the order) to reduce the depth.

Note: Generally, the node size of a B-Tree is kept equal to the disk block size.

B-Trees

Definition (B-Tree)

A B-Tree of order m is an m-ary tree with the following properties:

1. The data items are stored at leaves.
2. The non-leaf nodes store up to $m-1$ keys to guide the searching; The key i represents the smallest key in subtree $i+1$.
3. The root is either a leaf or has between 2 and m children.
4. All non-leaf nodes (except the root) have between $\lceil m/2 \rceil$ and m children.
5. All leaves are at the same depth and have between $\lceil k/2 \rceil$ and k data items, for some k.

B-Trees

Definition (B-Tree)

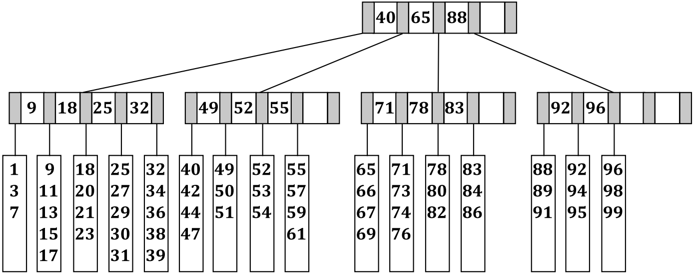
A B-Tree of order m is an m-ary tree with the following properties:

1. The data items are stored at leaves.
2. The non-leaf nodes store up to $m - 1$ keys to guide the searching; The key i represents the smallest key in subtree $i + 1$.
3. The root is either a leaf or has between 2 and m children.
4. All non-leaf nodes (except the root) have between $\lceil m/2 \rceil$ and m children.
5. All leaves are at the same depth and have between $\lceil k/2 \rceil$ and k data items, for some k.

Note: The properties 3 and 5 are relaxed for the first k insertions.

B-Trees

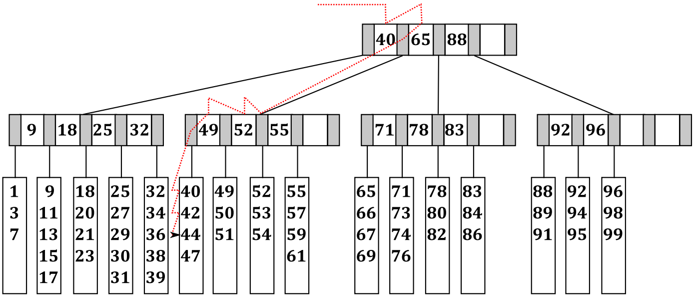
A B-Tree of order 5 and depth 3 that contains 59 data items.



Note: Here, $m = k = 5$.

Searching into B-Trees

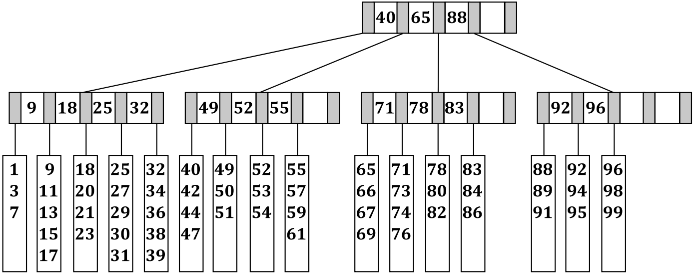
Searching 44 in the following B-Tree:



Note: The lookup (traversal shown in red) is over the disk.

Insertion into B-Trees

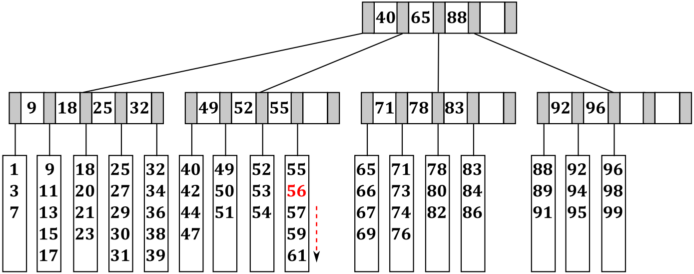
Inserting 56 into the following B-Tree:



Note: Insertion requires shifting of a few data items.

Insertion into B-Trees

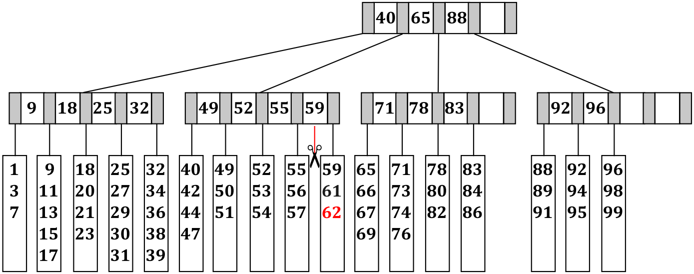
Inserting 62 into the following B-Tree:



Note: Insertion requires breaking a leaf node into a pair of nodes.

Insertion into B-Trees

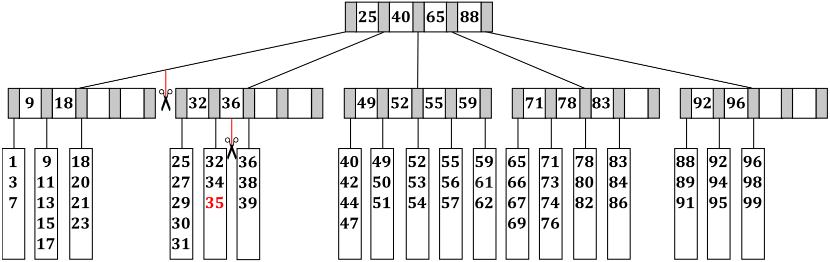
Inserting 35 into the following B-Tree:



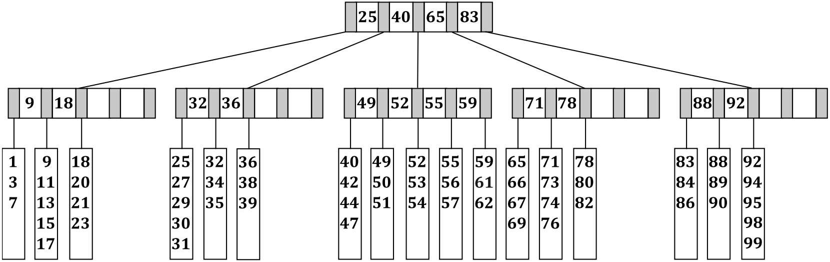
Note: Insertion requires breaking a leaf node into a pair of nodes and the inclusion of a new non-leaf node.

Deletion from B-Trees

Deleting 96 from the following B-Tree:



Deletion from B-Trees



Note: Deletion requires merging of a leaf node with another node.

Characteristics of B^+-Trees

Unlike the B-Trees, a B^+-tree does not have data items in the internal (non-leaf) nodes.

Interestingly, more number of keys can be fit on a page of memory in B^+-Trees (because no data is associated with internal nodes), resulting into fewer cache misses in order to access data that is on a leaf node.

B^+-Trees

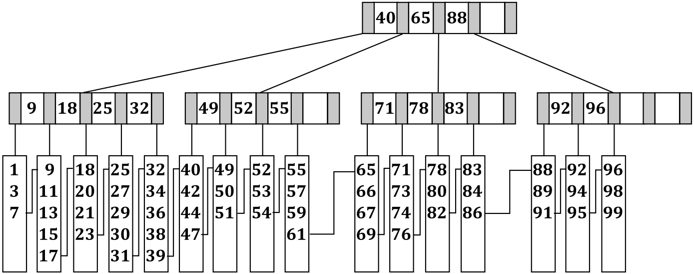
Definition (B^+-Tree)

A B^+-Tree of order m is an m-ary tree with the following properties:

- The data items are stored at leaves.
- The non-leaf nodes store up to $m-1$ keys to guide the searching; The key i represents the smallest key in subtree $i+1$.
- The root is either a leaf or has between 2 and m children.
- All leaves are at the same depth and have up to k data items, for some k.

B$^+$-Trees

A B$^+$-Tree of order 5 and depth 3 that contains 59 data items.



B-Trees vs B^+-Trees

B-Trees	**B^+-Trees**
1. Data are stored on leaf nodes as well as in internal nodes	1. Data are stored on the leaf nodes only
2. Redundant search keys are not allowed	2. Redundant search keys are allowed
3. Leaf nodes are not linked together	3. Leaf nodes are linked together
4. Only direct access is possible	4. Both sequential and direct access are possible
5. Searching is slower	5. Searching is faster
6. Deletion of internal nodes are complicated	6. Deletion of internal nodes are easy

R*-Trees

R*-Trees are tree data structures used for spatial access methods, i.e., for indexing multi-dimensional information such as geographical coordinates, rectangles or polygons.

Database indexing with hash table

`Point Searching` can be done in a hash table by the primary key. This is faster than indexing with a tree algorithm.

Point Searching with hash table takes $O(1)$ time but on trees takes $\log n$ time.

Disadvantages of indexing with hash table

- `Range Searching` is inefficient with hash tables. Tree algorithms support this in $\log n$ time whereas hash indexes can result in a full table scan incurring $O(n)$ time.
- Constant overhead of hash indexes is usually bigger (which is no factor in theta notation, but it still exists).
- Tree algorithms are usually easier to maintain, grow with data, scale, etc. However, hash indexes work with pre-defined hash sizes.

Note: There are scalable hashing algorithms like RUSH (Replication Under Scalable Hashing).

Cost analysis of different access methods

	Scan	Point Search	Range Search	Insert	Delete
Heap	BD	0.5BD	BD	2D	Search + D
Sorted	BD	Dlog B	Dlog B + #M	Search + BD	Search + BD
Clustered	1.5BD	Dlog 1.5B	Dlog 1.5B + #M	Search + D	Search + D
Unclustered tree index	BD(R + 0.15)	D(1 + log 0.15B)	Dlog 0.15B + #M	D(3 + log 0.15B)	Search + 2D
Unclustered hash index	BD(R + 0.125)	2D	BD	4D	Search + 2D

- B denotes the number of data pages
- R denotes the number of records per page
- D denotes the average time to read/write from disk
- C denotes the average time to process a record (e.g., equality check)
- #M denotes the number of matches