B-Tree

Bùi Tiến Lên

2021

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

Contents

B-tree

Introduction

Assumption. So far, search trees were limited to main memory structures $\rightarrow$ the dataset organized in a search tree fits in main memory (**internal memory**)

Problem. Transaction data of a bank > 1 TB per day $\rightarrow$ use secondary storage media (HDD, SSD, etc.) (**external memory**)

Goal. Make a search tree structure secondary-storage-enabled

File system model

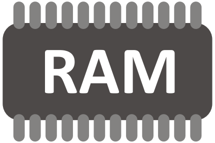
Page. Contiguous block of data (e.g., a file or 4,096-byte chunk).

Probe. First access to a page (e.g., from disk to memory).

Property. Time required for a probe is much larger than time to access data within a page.

Cost. Number of probes.

Goal. Access data using minimum number of probes.



B-tree

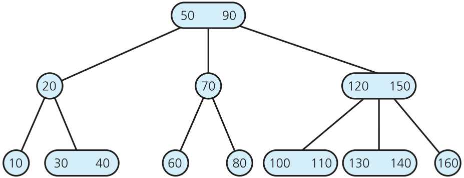
Concept 1

A B-tree of order M ($M > 2$) invented by Rudolf Bayer and Edward M.McCreight is an M-ary tree with the following properties:

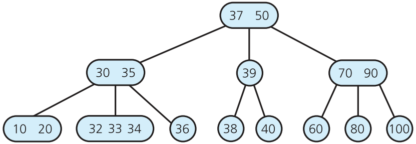
1. The root is either a leaf or has between 2 and M **children**.
2. All nonleaf nodes (except the root) have between $\lfloor \frac{M+1}{2} \rfloor$ and M **children**.
   - **minimum degree** $m = \lfloor \frac{M+1}{2} \rfloor$
   - **maximum degree** M
3. All leaves are at the same depth h.
4. All nodes store **keys** to guide the searching.

B-tree (cont.)

- 2-3 tree



- 2-3-4 tree



The height of a B-tree

Theorem 1

If $n \geq 1$, then for any n-key B-tree T of height h and minimum degree m,

$$h \leq \log_m \frac{n+1}{2} \tag{1}$$

In practice. Number of probes is at most 4.

Optimization. Always keep root page in memory.

Structure of Node

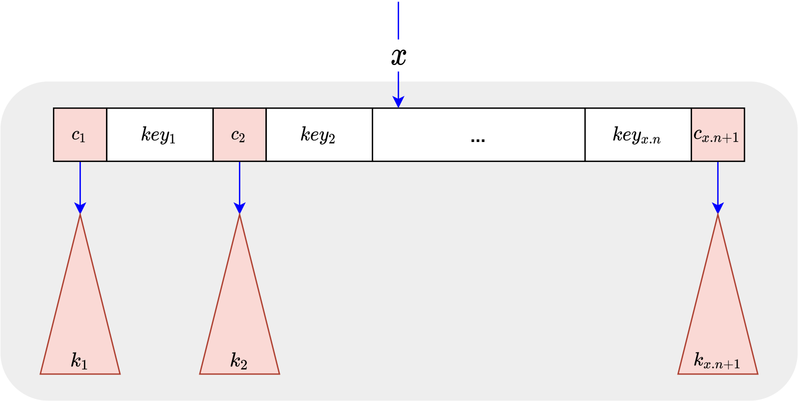
1. Every node x has the following fields:
   - $x.n$, the number of keys currently stored in node x,
   - the $x.n$ keys themselves, $x.key_1, x.key_2, \ldots, x.key_{x.n}$ stored in nondecreasing order, so that

$$x.key_1 \leq x.key_2 \leq \ldots \leq x.key_{x.n} \tag{2}$$

   - $x.leaf$, a boolean value that is **true** if x is a leaf and **false** if x is an internal node.
2. Each internal node x also contains $x.n+1$ pointers $x.c_1, x.c_2, \ldots, x_{c_{x.n+1}}$ to its children
3. The keys $x.key_i$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then

$$k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \ldots \leq x.key_{x.n} \leq k_{x.n+1} \tag{3}$$

Structure of Node (cont.)



Basic operations on B-trees

- Searching
- Creation
- Single-pass Insertion
- Insertion
- Deletion

Introduction

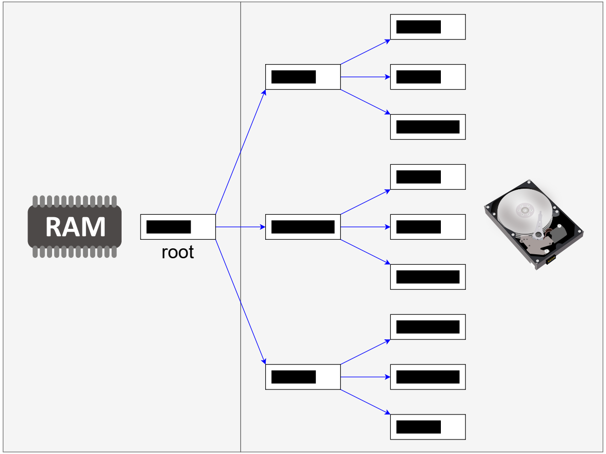
We adopt two conventions:

- The root of the B-tree is always in main memory, so that a DISK-READ on the root is never required; a DISK-WRITE of the root is required, however, whenever the root node is changed.
- Any nodes that are passed as parameters must already have had a DISK-READ operation performed on them.

There are two kinds of algorithms:

- "Single pass down" algorithms that proceed downward from the root of the tree, without having to back up (**pre-processing**)
- "Two-pass" algorithms (**post-processing**)

Introduction (cont.)



Searching a B-tree

```
B-Tree-Search(x, k)
    if x.leaf return null
    i ← 1
    while i ≤ x.n and k > x.key_i
        i ← i + 1
    if i ≤ x.n and k = x.key_i
        return (x, i)
    else
        Disk-Read(x.c_i)
        return B-Tree-Search(x.c_i, k)
```

- **Challenge**: Can we make any improvement?

Creating an empty B-tree

$\textsc{B-Tree-Create}(T)$

$\quad x \leftarrow \textsc{Allocate-Node}()$

$\quad x.leaf \leftarrow \textbf{true}$

$\quad x.n \leftarrow 0$

$\quad \textsc{Disk-Write}(x)$

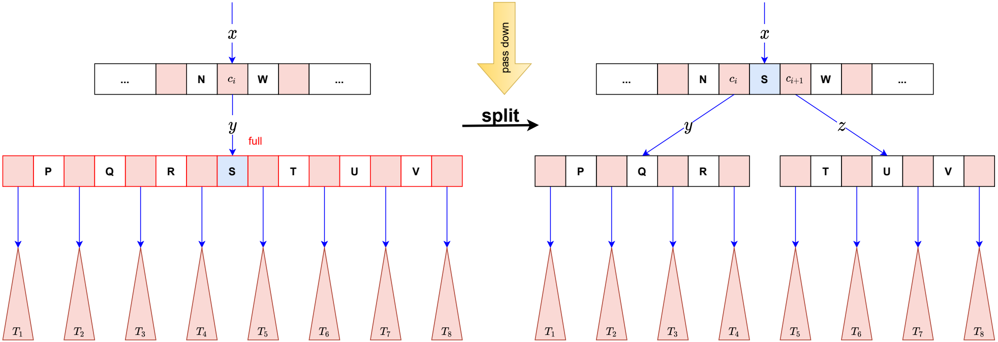
$\quad T.root \leftarrow x$

Splitting a node in a B-tree

```
B-Tree-Split-Child(x, i)
    z ← Allocate-Node()
    y ← x.c_i
    z.leaf ← y.leaf
    z.n ← m − 1
    for j ← [1, ..., m − 1]
        z.key_j ← y.key_{j+m}
    if not y.leaf
        for j ← [1, ..., m]
            z.c_j ← y.c_{j+m}
    y.n ← m − 1
    for j ← [x.n + 1, ..., i + 1]
        x.c_{j+1} ← x.c_j
    x.c_{i+1} ← z
    for j ← [x.n, ..., i]
        x.key_{j+1} ← x.key_j
    x.key_i ← y.key_m
    x.n ← x.n + 1
    Disk-Write(y)
    Disk-Write(z)
    Disk-Write(x)
```

Splitting a node in a B-tree (cont.)

- B-tree($m = 4, M = 8$): split

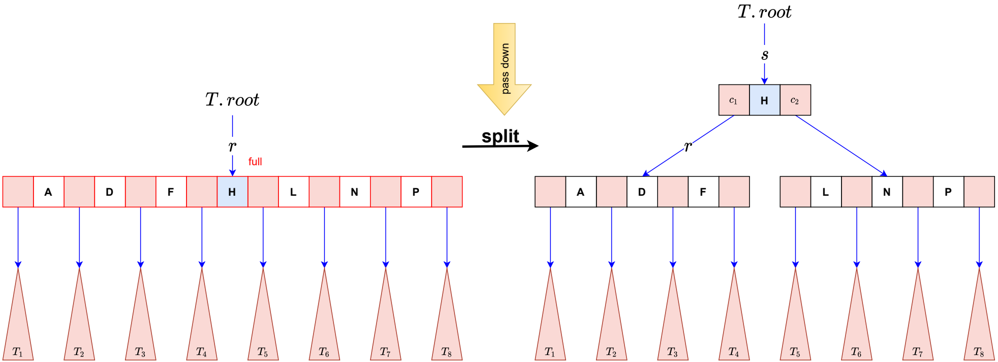


Inserting a key into a B-tree

```
B-TREE-INSERT(T, k)
    r ← T.root
    if r.n = 2m − 1
        s ← ALLOCATE-NODE()
        T.root ← s
        s.leaf ← false
        s.n ← 0
        s.c_1 ← r
        B-TREE-SPLIT-CHILD(s, 1)
        B-TREE-INSERT-NONFULL(s, k)
    else
        B-TREE-INSERT-NONFULL(r, k)
```

Splitting a node in a B-tree (cont.)

- B-tree($m = 4, M = 8$): split at root



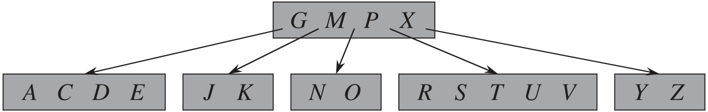
Inserting a key into a B-tree

```
B-TREE-INSERT-NONFULL(x, k)
    i ← x.n
    if x.leaf
        while i ≥ 1 and k < x.key_i
            x.key_{i+1} ← x.key_i
            i ← i − 1
        x.key_{i+1} ← k
        x.n ← x.n + 1
        DISK-WRITE(x)
    else
        while i ≥ 1 and k < x.key_i
            i ← i − 1
        i ← i + 1
        DISK-READ(x.c_i)
        if x.c_i.n = 2m − 1
            B-TREE-SPLIT-CHILD(x, i)
            if k > x.key_i
                i ← i + 1
        B-TREE-INSERT-NONFULL(x.c_i, k)
```

Example of Insertion

Consider B-tree($m = 3, M = 6$)

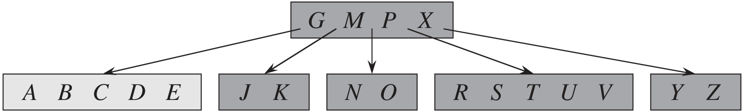
1 Initial tree



Example of Insertion (cont.)

Consider B-tree($m = 3, M = 6$)

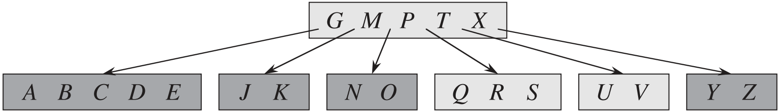
2 B inserted



Example of Insertion (cont.)

Consider B-tree($m = 3, M = 6$)

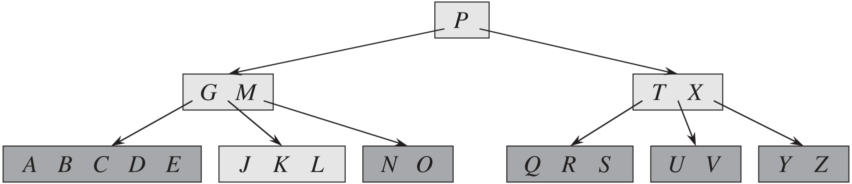
3 Q inserted



Example of Insertion (cont.)

Consider B-tree($m = 3, M = 6$)

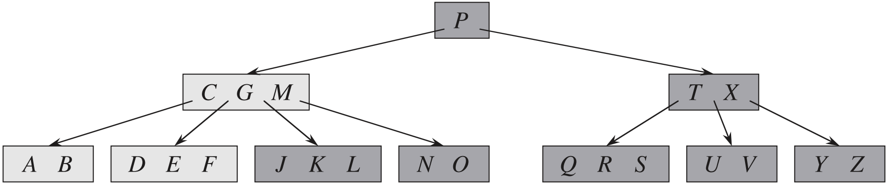
4 L inserted



Example of Insertion (cont.)

Consider B-tree($m = 3, M = 6$)

5 F inserted

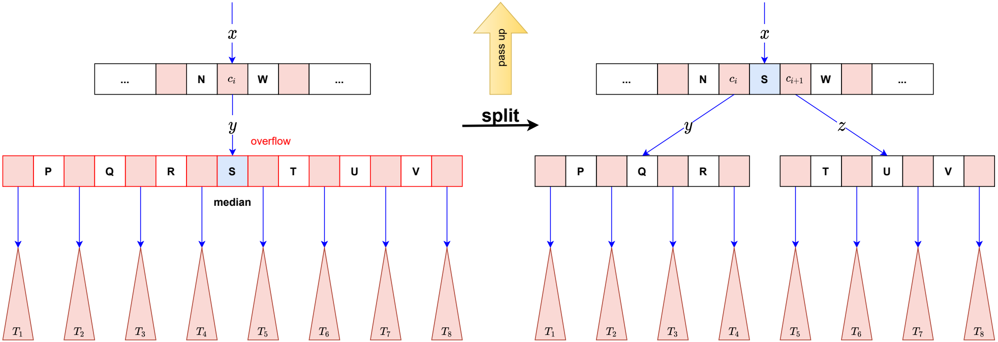


Bottom-up Insertion

1. All insertions start at a **leaf node**.
2. **If** the node contains more than the maximum allowed number of keys, **then** the node is overflow:
   - A **single median key** is chosen from among the node's keys
   - Keys less than the median are put in **the new left node** and keys greater than the median are put in **the new right node**.
   - The median is inserted in **the node's parent**, which may cause it to be split, and so on. If the node has no parent (i.e., the node was the root), create a new root above this node.

Bottom-up Insertion (cont.)

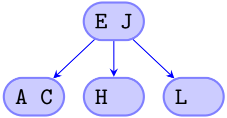
- B-tree$(m = 4, M = 7)$



Example of Insertion

Consider B-tree($m = 2, M = 3$)

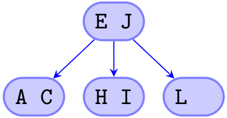
1 Initial tree



Example of Insertion (cont.)

Consider B-tree($m = 2, M = 3$)

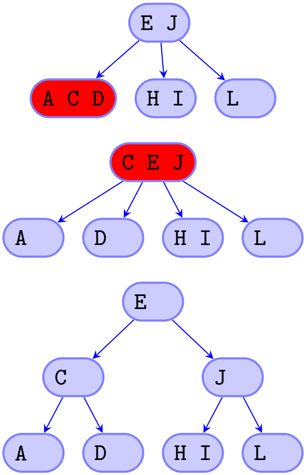
2 `I` inserted



Example of Insertion (cont.)

Consider B-tree($m = 2, M = 3$)

3 `D` inserted



Deletion

Deletion from a leaf node

- Delete it from the node.
- **If** underflow happens, rebalance the tree.

Deletion from an internal node: Each key in an internal node acts as a separation value for two subtrees

- Choose a new separator (either the largest key in the left subtree or the smallest key in the right subtree), remove it from **the leaf node** it is in, and replace the **key** to be deleted with the new separator.

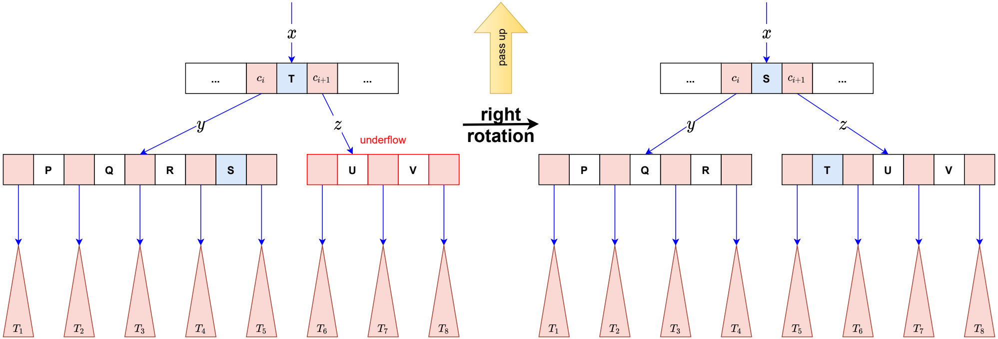
Deletion (cont.)

Rebalancing after deletion

1. **If** the deficient node's right sibling exists and has more than the minimum number of elements, then **rotate left**
2. Otherwise, **if** the deficient node's left sibling exists and has more than the minimum number of elements, then **rotate right**
3. Otherwise, **if** both immediate siblings have only the minimum number of elements, then **merge** with a sibling sandwiching their separator taken off from their parent.

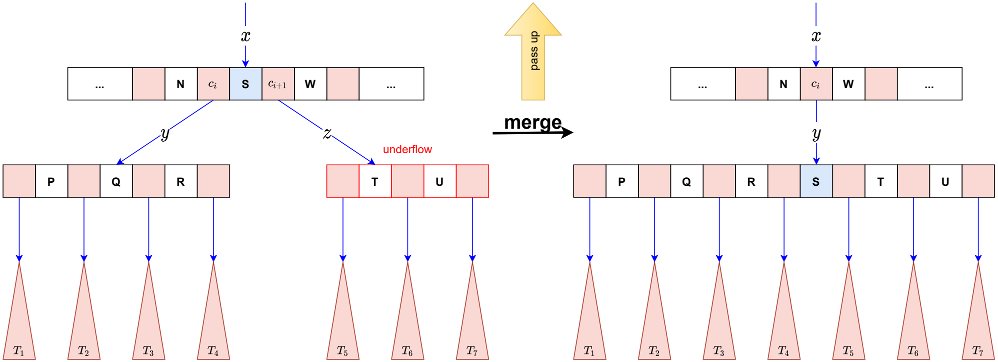
Right rotation

- B-tree($m = 4, M = 7$)



Merge

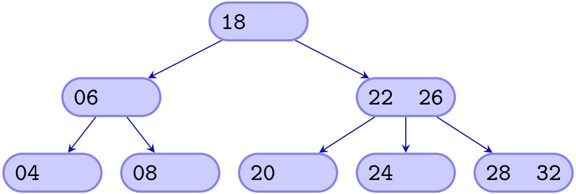
- B-tree$(m = 4, M = 7)$



Example of Deletion

Consider B-tree($m = 2, M = 3$)

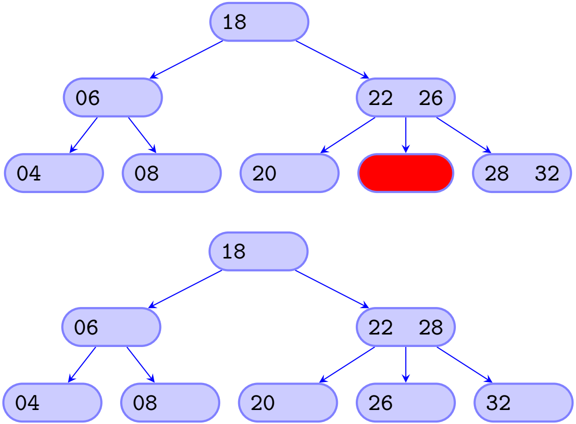
1. Initial tree



Example of Deletion (cont.)

Consider B-tree($m = 2, M = 3$)

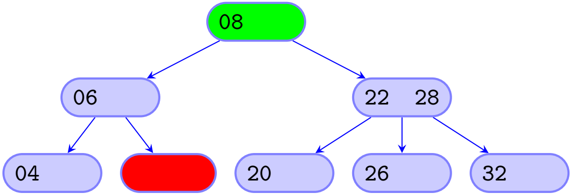
2 24 deleted



Example of Deletion (cont.)

Consider B-tree($m = 2, M = 3$)

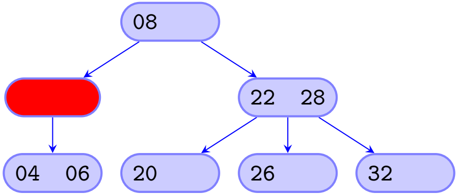
3 18 deleted



Example of Deletion (cont.)

Consider B-tree($m = 2, M = 3$)

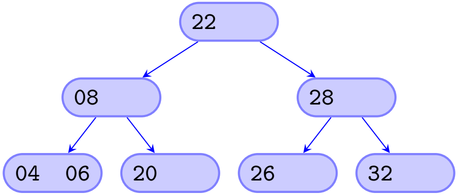
3 18 deleted ...



Example of Deletion (cont.)

Consider B-tree($m = 2, M = 3$)

3 18 deleted ...



Applications

- Indexing
- B-tree variants

Indexing

Concept 2

Indexing is a data structure technique to efficiently retrieve records from the database files based on some attributes on which the indexing has been done.

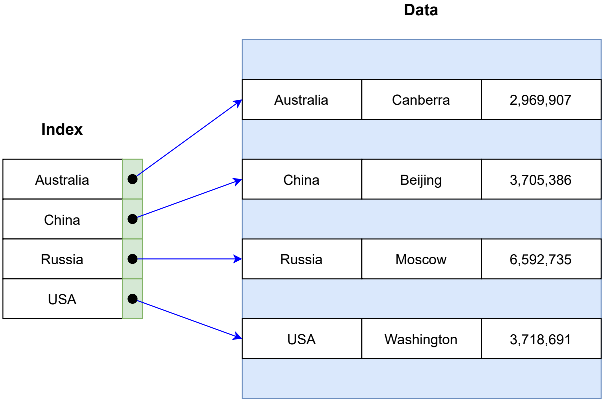
Indexing can be

- Primary Index: primary key
- Secondary Index: candidate key
- Clustering Index: non-key

DepartmentName	BudgetCode	OfficeNumber	Phone
Administration	BC-100-10	BLDG01-300	360-285-8100
Legal	BC-200-10	BLDG01-200	360-285-8200
Accounting	BC-300-10	BLDG01-100	360-285-8300
Finance	BC-400-10	BLDG01-140	360-285-8400
Human Resources	BC-500-10	BLDG01-180	360-285-8500
Production	BC-600-10	BLDG02-100	360-287-8600
Marketing	BC-700-10	BLDG02-200	360-287-8700
InfoSystems	BC-800-10	BLDG02-270	360-287-8800

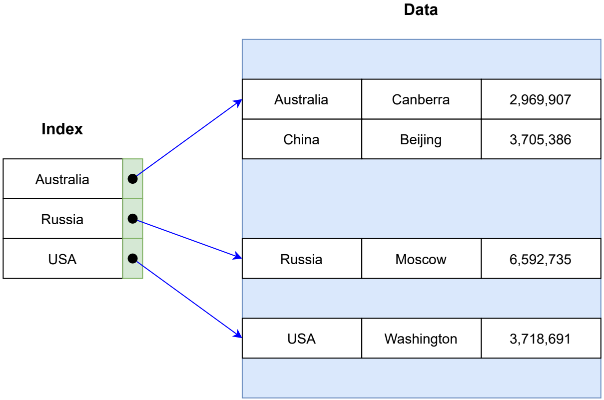
Dense Index

- In dense index, there is an index record for every search key value in the database. This makes searching faster but requires more space to store index records itself. Index records contain search key value and a pointer to the actual record on the disk.



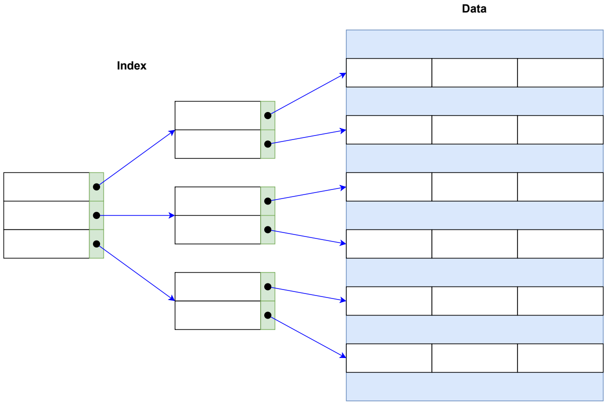
Sparse Index

- In sparse index, index records are not created for every search key.



Multilevel Index

- Multi-level Index helps in breaking down the index into several smaller indices in order to make the top level so small that it can be saved in a single disk block, which can easily be accommodated anywhere in the main memory.



B-tree variants

Concept 3

B+ tree is a B-tree

- Copies of the keys are stored in the internal nodes.
- The keys and records are stored in leaves.
- In addition, a leaf node may include a pointer to the next leaf node to speed sequential access.

Concept 4

B* tree is a B-tree that ensures non-root nodes are at least $2/3$ full instead of $1/2$.

Searching a B+ Tree

Table `people`

ID	name	age
1	Peter	20
2	Mary	30
3	John	25
...	...	...

Queries

```
SELECT name
FROM people
WHERE age = 25
```

Exact key values:

- Start at the root
- Proceed down, to the leaf

```
SELECT name
FROM people
WHERE 20<=age AND
age<=30
```

Range queries:

- As above
- Then sequential traversal

Applications

B-trees (and variants) are widely used for file systems and databases.

- Windows: NTFS.
- Mac: HFS, HFS+.
- Linux: ReiserFS, XFS, Ext3FS, JFS.
- Databases: ORACLE, DB2, INGRES, SQL, PostgreSQL.

Workshop

Quiz

1. What is a B-tree?

. . . .

. . . .

. . . .

2. What is Indexing?

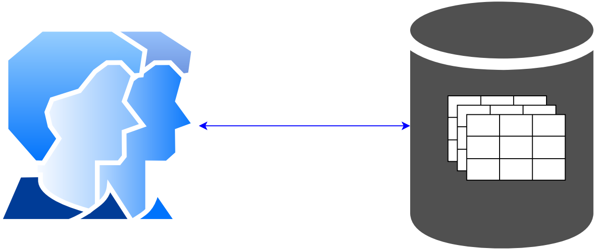
. . . .

. . . .

. . . .

Projects

1. (Big project) Design and implement a tiny relational database project using B+ tree.



References

Cormen, T. H. (2009).
Introduction to algorithms.
MIT press.

Sedgewick, R. (2002).
Algorithms in Java, Parts 1-4, volume 1.
Addison-Wesley Professional.

Walls and Mirrors (2014).
Data Abstraction And Problem Solving with C++.
Pearson.