Tree Indexes (Part I)
DATA STRUCTURES

Internal Meta-data
Core Data Storage
Temporary Data Structures
Table Indexes
TABLE INDEXES

A table index is a replica of a subset of a table's columns that are organized and/or sorted for efficient access using a subset of those columns.

The DBMS ensures that the contents of the table and the index are logically in sync.
TABLE INDEXES

It is the DBMS's job to figure out the best index(es) to use to execute each query.

There is a trade-off on the number of indexes to create per database.
→ Storage Overhead
→ Maintenance Overhead
TODAY'S AGENDA

B+Tree Overview
Design Decisions
Optimizations
B-TREE FAMILY

There is a specific data structure called a B-Tree, but then people also use the term to generally refer to a class of data structures.

→ B-Tree
→ B+Tree
→ B^{link}-Tree
→ B^{*}Tree
A B+Tree is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in $O(\log n)$. → Generalization of a binary search tree in that a node can have more than two children. → Optimized for systems that read and write large blocks of data.
A B+tree is an $M$-way search tree with the following properties:

→ It is perfectly balanced (i.e., every leaf node is at the same depth).
→ Every inner node other than the root, is at least half-full
   \[ \frac{M}{2} - 1 \leq \text{#keys} \leq M - 1 \]
→ Every inner node with $k$ keys has $k+1$ non-null children
B+TREE EXAMPLE

Inner Node

Leaf Nodes

Sibling Pointers
B+TREE EXAMPLE

Inner Node

Leaf Nodes

Sibling Pointers

<value> | <key>
Every node in the B+Tree contains an array of key/value pairs.

→ The keys will always be the column or columns that you built your index on
→ The values will differ based on whether the node is classified as **inner nodes** or **leaf nodes**.

The arrays are (usually) kept in sorted key order.
LEAF NODE VALUES

**Approach #1: Record Ids**
→ A pointer to the location of the tuple that the index entry corresponds to.

**Approach #2: Tuple Data**
→ The actual contents of the tuple is stored in the leaf node.
→ Secondary indexes have to store the record id as their values.
B+TREE LEAF NODES

B+Tree Leaf Node

Prev: K1 V1 • • • Kn Vn
Next: PagenID

PageID
B+TREE LEAF NODES

B+Tree Leaf Node

- Key + Value
- Prev
- Next
- PageID
**B+TREE LEAF NODES**

![B+Tree Leaf Node Diagram]

- **Level**
  - #
- **Slots**
  - #
- **Prev**
  - •
- **Next**
  - •

**Sorted Keys**

- K1
- K2
- K3
- K4
- K5
- ... Kn

**Values**

- •
- •
- •
- •
- •
- ... •
B+TREE LEAF NODES

B+Tree Leaf Node

- Level
  - #
  - #
  - Prev
  - Next
- Sorted Keys
  - K1
  - K2
  - K3
  - K4
  - K5
  - ... Kn
- Values
  - • • •
  - • • •
  - • • •
  - • • •
  - • • •
  - • • •
  - • • •
  - • • •
  - • • •
  - • • •
  - ...
B-TREE VS. B+TREE

The original B-Tree from 1972 stored keys + values in all nodes in the tree.
→ More space efficient since each key only appears once in the tree.

A B+Tree only stores values in leaf nodes. Inner nodes only guide the search process.
B+TREE INSERT

Find correct leaf $L$.
Put data entry into $L$ in sorted order.
If $L$ has enough space, done!
Else, must split $L$ into $L$ and a new node $L_2$
   → Redistribute entries evenly, copy up middle key.
   → Insert index entry pointing to $L_2$ into parent of $L$.

To split inner node, redistribute entries evenly, but push up middle key.

Source: Chris Re
B+TREE VISUALIZATION

https://cmudb.io/btree

Source: David Gales (Univ. of San Francisco)
B+TREE DELETE

Start at root, find leaf \( L \) where entry belongs.
Remove the entry.
If \( L \) is at least half-full, done!
If \( L \) has only \( \frac{M}{2}-1 \) entries,
\( \rightarrow \) Try to re-distribute, borrowing from sibling (adjacent node with same parent as \( L \)).
\( \rightarrow \) If re-distribution fails, merge \( L \) and sibling.

If merge occurred, must delete entry (pointing to \( L \) or sibling) from parent of \( L \).
B+TREES IN PRACTICE

Typical Fill-Factor: 67%.
→ Average Fanout = 2*100*0.67 = 134

Typical Capacities:
→ Height 4: 1334 = 312,900,721 entries
→ Height 3: 1333 = 2,406,104 entries

Pages per level:
→ Level 1 = 1 page = 8 KB
→ Level 2 = 134 pages = 1 MB
→ Level 3 = 17,956 pages = 140 MB
CLUSTERED INDEXES

The table is stored in the sort order specified by the primary key.
→ Can be either heap- or index-organized storage.

Some DBMSs always use a clustered index.
→ If a table doesn’t include a pkey, the DBMS will automatically make a hidden row id pkey.

Other DBMSs cannot use them at all.
The DBMS can use a B+Tree index if the query provides any of the attributes of the search key.

Example: Index on \(<a, b, c>\)
→ Supported: \((a=5 \AND b=3)\)
→ Supported: \((b=3)\).

Not all DBMSs support this.

For hash index, we must have all attributes in search key.
SELECTION CONDITIONS

Find Key=(A,B)
SELECTION CONDITIONS

Find Key=(A,B)
Find Key=(*,B)
Find Key=(A,B)  
Find Key=(*,B)
B+TREE DESIGN CHOICES

Node Size
Merge Threshold
Variable Length Keys
Non-Unique Indexes
Intra-Node Search
The slower the disk, the larger the optimal node size for a B+Tree.

→ HDD ~1MB
→ SSD: ~10KB
→ In-Memory: ~512B

Optimal sizes can vary depending on the workload
→ Leaf Node Scans vs. Root-to-Leaf Traversals
Some DBMSs don't always merge nodes when it is half full.

Delaying a merge operation may reduce the amount of reorganization.

May be better to just let underflows to exist and then periodically rebuild entire tree.
VARIABLE LENGTH KEYS

Approach #1: Pointers
→ Store the keys as pointers to the tuple’s attribute.

Approach #2: Variable Length Nodes
→ The size of each node in the B+Tree can vary.
→ Requires careful memory management.

Approach #3: Key Map
→ Embed an array of pointers that map to the key + value list within the node.
NON-UNIQUE INDEXES

Approach #1: Duplicate Keys
→ Use the same leaf node layout but store duplicate keys multiple times.

Approach #2: Value Lists
→ Store each key only once and maintain a linked list of unique values.
NON-UNIQUE: DUPLICATE KEYS

B+Tree Leaf Node

- Sorted Keys: K1, K1, K1, K2, K2, ..., Kn
- Values: \( \vdots \)

Diagram:

- Tree structure with leaf nodes containing keys and values.
NON-UNIQUE: VALUE LISTS

B+Tree Leaf Node

<table>
<thead>
<tr>
<th>Level</th>
<th>Slots</th>
<th>Prev</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>#</td>
<td>⚫️</td>
<td>⚫️</td>
</tr>
</tbody>
</table>

Sorted Keys

| K1 | K2 | K3 | K4 | K5 | ... | Kn |

Values

<table>
<thead>
<tr>
<th>⚫️</th>
<th>⚫️</th>
<th>⚫️</th>
</tr>
</thead>
<tbody>
<tr>
<td>⚫️</td>
<td>⚫️</td>
<td></td>
</tr>
</tbody>
</table>
INTRA-NODE SEARCH

Approach #1: Linear
→ Scan node keys from beginning to end.

Approach #2: Binary
→ Jump to middle key, pivot left/right depending on comparison.

Approach #3: Interpolation
→ Approximate location of desired key based on known distribution of keys.

Find Key=8

<table>
<thead>
<tr>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
</table>

Find Key=8
INTRA-NODE SEARCH

Approach #1: Linear
→ Scan node keys from beginning to end.

Approach #2: Binary
→ Jump to middle key, pivot left/right depending on comparison.

Approach #3: Interpolation
→ Approximate location of desired key based on known distribution of keys.
INTRA-NODE SEARCH

Approach #1: Linear
→ Scan node keys from beginning to end.

Approach #2: Binary
→ Jump to middle key, pivot left/right depending on comparison.

Approach #3: Interpolation
→ Approximate location of desired key based on known distribution of keys.

Find Key=8

```
4 5 6 7 8 9 10
```

```
4 5 6 7 8 9 10
```
INTRA-NODE SEARCH

Approach #1: Linear
→ Scan node keys from beginning to end.

Approach #2: Binary
→ Jump to middle key, pivot left/right depending on comparison.

Approach #3: Interpolation
→ Approximate location of desired key based on known distribution of keys.

Find Key=8
INTRA-NODE SEARCH

Approach #1: Linear
→ Scan node keys from beginning to end.

Approach #2: Binary
→ Jump to middle key, pivot left/right depending on comparison.

Approach #3: Interpolation
→ Approximate location of desired key based on known distribution of keys.
INTRA-NODE SEARCH

Approach #1: Linear
→ Scan node keys from beginning to end.

Approach #2: Binary
→ Jump to middle key, pivot left/right depending on comparison.

Approach #3: Interpolation
→ Approximate location of desired key based on known distribution of keys.
INTRA-NODE SEARCH

Approach #1: Linear
→ Scan node keys from beginning to end.

Approach #2: Binary
→ Jump to middle key, pivot left/right depending on comparison.

Approach #3: Interpolation
→ Approximate location of desired key based on known distribution of keys.

Find Key = 8

<table>
<thead>
<tr>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
</table>

Offset: 7 - (10 - 8) = 5

<table>
<thead>
<tr>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
</table>
OPTIMIZATIONS

Prefix Compression
Suffix Truncation
Bulk Insert
Pointer Swizzling
PREFIX COMPRESSION

Sorted keys in the same leaf node are likely to have the same prefix.

Instead of storing the entire key each time, extract common prefix and store only unique suffix for each key.
→ Many variations.
Prefix Compression

Sorted keys in the same leaf node are likely to have the same prefix.

Instead of storing the entire key each time, extract common prefix and store only unique suffix for each key.

→ Many variations.
SUFFIX TRUNCATION

The keys in the inner nodes are only used to "direct traffic".
→ We don't actually need the entire key.

Store a minimum prefix that is needed to correctly route probes into the index.
The keys in the inner nodes are only used to "direct traffic". → We don't actually need the entire key.

Store a minimum prefix that is needed to correctly route probes into the index.
The fastest/best way to build a B+Tree is to first sort the keys and then build the index from the bottom up.
The fastest/best way to build a B+Tree is to first sort the keys and then build the index from the bottom up.

Keys: 3, 7, 9, 13, 6, 1

Sorted Keys: 1, 3, 6, 7, 9, 13
The fastest/best way to build a B+Tree is to first sort the keys and then build the index from the bottom up.
The fastest/best way to build a B+Tree is to first sort the keys and then build the index from the bottom up.

Keys: 3, 7, 9, 13, 6, 1

Sorted Keys: 1, 3, 6, 7, 9, 13
Nodes use page ids to reference other nodes in the index. The DBMS has to get the memory location from the page table during traversal.

If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids, thereby removing the need to get address from the page table.
POINTER SWIZZLING

Nodes use page ids to reference other nodes in the index. The DBMS has to get the memory location from the page table during traversal.

If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids, thereby removing the need to get address from the page table.
Nodes use page ids to reference other nodes in the index. The DBMS has to get the memory location from the page table during traversal.

If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids, thereby removing the need to get address from the page table.
Nodes use page ids to reference other nodes in the index. The DBMS has to get the memory location from the page table during traversal.

If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids, thereby removing the need to get address from the page table.
Nodes use page ids to reference other nodes in the index. The DBMS has to get the memory location from the page table during traversal.

If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids, thereby removing the need to get address from the page table.
Nodes use page ids to reference other nodes in the index. The DBMS has to get the memory location from the page table during traversal.

If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids, thereby removing the need to get address from the page table.
CONCLUSION

The venerable B+Tree is always a good choice for your DBMS.
NEXT CLASS

Skip Lists
Radix Trees
Inverted Indexes