Tree Indexes
–Part I
ADMINISTRIVIA

Project #1 is due Fri Sept 27th @ 11:59pm

Homework #2 is due Mon Sept 30th @ 11:59pm
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 attributes that are organized and/or sorted for efficient access using a subset of those attributes.

The DBMS ensures that the contents of the table and the index are logically in sync.
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
There is a specific data structure called a **B-Tree**.

People also use the term to generally refer to a class of balanced tree data structures:

- **B-Tree** (1971)
- **B+Tree** (1973)
- **B*Tree** (1977?)
- **B^link-Tree** (1981)
B-TREE FAMILY

There is a specific data structure called a B-Tree.
People also use the term to generally refer to a class of balanced tree data structures:

→ B-Tree (1971)
→ B+Tree (1973)
→ B*Tree (1977?)
→ B-link-Tree (1981)
**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.
B+TREE PROPERTIES

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 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

Sibling Pointers

Leaf Nodes
**B+TREE EXAMPLE**

- **Leaf Nodes**
  - <value> | <key>
  - <node*> | <key>

- **Inner Node**
  - <key>
  - <value>
  - <node*> | <key>
  - <node*> | <key>

- **Sibling Pointers**
  - <key>
  - <node*> | <key>

- **Leaf Nodes**
  - 1 3
  - 6 7
  - 9 13

- **Inner Node**
  - 5 9

- **Keys and Values**
  - <key>
  - <value>
  - <value>
  - <value>
  - <value>
  - <value>

- **Comparisons**
  - <5
  - <9
  - ≥9
Every B+Tree node is comprised of an array of key/value pairs.

→ The keys are derived from the attributes(s) that the index is based 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.
B+TREE LEAF NODES

B+Tree Leaf Node

Prev: $K_1 V_1 \cdots K_n V_n$

Next:
**B+TREE LEAF NODES**

**B+Tree Leaf Node**

- **PageID**: (Prev) **K1** **V1** • • • **Kn** **Vn** (Next)
- **PageID**

Diagram showing the structure of a B+ Tree leaf node with key-value pairs and page IDs.
B+TREE LEAF NODES

B+Tree Leaf Node

Key+Value

PageID

Prev

Next

K1

Kn

CMU 15-445/645 (Fall 2019)
B+Tree Leaf Nodes

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

<table>
<thead>
<tr>
<th>K1</th>
<th>K2</th>
<th>K3</th>
<th>K4</th>
<th>K5</th>
<th>...</th>
<th>Kn</th>
</tr>
</thead>
</table>

Values

| •  | •  | •  | •  | •  | ... | •  |
B+TREE LEAF NODES

B+Tree Leaf Node

Sorted Keys

Values

Level  Slots  Prev  Next

K1  K2  K3  K4  K5  \cdots  Kn

\begin{itemize}
  \item K1
  \item K2
  \item K3
  \item K4
  \item K5
  \item \cdots
  \item Kn
\end{itemize}
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.
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 node \( L \). Put data entry into \( L \) in sorted order. If \( L \) has enough space, done!

Otherwise, split \( L \) keys 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 $M/2-1$ entries, 
→ Try to re-distribute, borrowing from sibling (adjacent node with same parent as $L$). 
→ If re-distribution fails, merge $L$ and sibling. 

If merge occurred, must delete entry (pointing to $L$ or sibling) from parent of $L$. 

Source: Chris Re
B+TREES IN PRACTICE

Typical Fill-Factor: 67%.

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 contain a primary key, the DBMS will automatically make a hidden row id primary key.

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.
Find Key=(A,B)
SELECTION CONDITIONS

Find Key=(A,B)
Find Key=(A,*)

A ≤ A
SELECTION CONDITIONS

Find Key=(A,B)
Find Key=(A,*)

A ≤ A
A ≤ B
SELECT CONDITIONS

Find Key=(A,B)
Find Key=(A,*)
Find Key=(*,B)
**SELECTION CONDITIONS**

Find Key=(A,B)
Find Key=(A,*)
Find Key=(*,B)
SELECTION CONDITIONS

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

Node Size
Merge Threshold
Variable Length Keys
Non-Unique Indexes
Intra-Node Search
The slower the storage device, 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 do not always merge nodes when it is half full.

Delaying a merge operation may reduce the amount of reorganization.

It may also 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 index can vary.
→ Requires careful memory management.

Approach #3: Padding
→ Always pad the key to be max length of the key type.

Approach #4: Key Map / Indirection
→ Embed an array of pointers that map to the key + value list within the node.
KEY MAP / INDIRECTION

B+Tree Leaf Node

Sorted Key Map

Key+Values

Prashanth  V4
Lin     V2  Andy  V1  Obama  V3
**KEY MAP / INDIREDICTION**

**B+Tree Leaf Node**

- **Level**
  - #

- **Slots**
  - #

- **Prev**
  - □

- **Next**
  - □

**Sorted Key Map**

- **Key/Values**

  - Prashanth V4
  - Lin V2
  - Andy V1
  - Obama V3
KEY MAP / INDIRECTION

B+Tree Leaf Node

Sorted Key Map

Key+Values

Prashanth    V4
Lin    V2
Andy    V1
Obama    V3
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  \ldots  Kn

Values

\ldots
**NON-UNIQUE: DUPLICATE KEYS**

**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
- K1
- K1
- K2
- K2
- ...  
- Kn

**Values**

- [ ]
- [ ]
- [ ]
- [ ]
- [ ]
- [ ]
- [ ]
- [ ]
- [ ]
- [ ]
- ...
NON-UNIQUE: VALUE LISTS

B+Tree Leaf Node

Level Slots Prev Next
# # ● ●

Sorted Keys
K1 K2 K3 K4 K5 ● ● ● Kn

Values
● ● ●
● ●
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

\[
\begin{array}{cccccccc}
4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array}
\]
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
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

Offset: 7-(10-8)=5
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.
SUFFIX TRUNCATION

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

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.

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

**Keys:** 3, 7, 9, 13, 6, 1
Nodes use page ids to reference other nodes in the index. The DBMS must 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. This avoids address lookups from the page table.
Nodes use page ids to reference other nodes in the index. The DBMS must 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. This avoids address lookups from the page table.
POINTER SWIZZLING

Nodes use page ids to reference other nodes in the index. The DBMS must 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. This avoids address lookups from the page table.
Nodes use page ids to reference other nodes in the index. The DBMS must 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. This avoids address lookups from the page table.
CONCLUSION

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

More B+Trees
Tries / Radix Trees
Inverted Indexes