Lecture #08

B+Tree Index
UPCOMING EVENTS

**PostgresML** (ML⇄DB Seminar)  
→ Monday Sept 25\(^{th}\) @ 4:30pm

**Weaviate** (ML⇄DB Seminar)  
→ Monday Oct 2\(^{nd}\) @ 4:30pm

**FeatureForm** (ML⇄DB Seminar)  
→ Monday Oct 9\(^{th}\) @ 4:30pm
LAST CLASS

Hash tables are important data structures that are used all throughout a DBMS.
→ Space Complexity: $O(n)$
→ Average Time Complexity: $O(1)$

Static vs. Dynamic Hashing schemes

DBMSs use mostly hash tables for their internal data structures.
TODAY'S AGENDA

B+Tree Overview
Design Choices
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)
- **Bε-Tree** (2003)
- **Bw-Tree** (2013)
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)
- Bε-Tree (2003)
- Bw-Tree (2013)
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**\text{link}-**Tree** (1981)
- **Bε-Tree** (2003)
- **Bw-Tree** (2013)
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), Bε-Tree (2003), Bw-Tree (2013).

This directory contains a correct implementation of Lehman and Yao's high-concurrency B-tree management algorithm (P. Lehman and S. Yao, Efficient Locking for Concurrent Operations on B-Trees, ACM Transactions on Database Systems, Vol 6, No. 4, December 1981, pp 650-670). We also use a simplified version of the deletion logic described in Lanin and Shasha (V. Lanin and D. Shasha, A Symmetric Concurrent B-Tree Algorithm, Proceedings of 1986 Fall Joint Computer Conference, pp 380-389).

The basic Lehman & Yao Algorithm
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)
- **Bε-Tree** (2003)
- **Bw-Tree** (2013)
A **B+Tree** is a self-balancing, ordered tree data structure that allows searches, sequential access, insertions, and deletions in $\mathcal{O}(\log n)$.

→ Generalization of a binary search tree, since 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 in the tree)

→ 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

- Root Node
  - node*: key
  - 20

- Sibling Pointers
  - <20
  - ≥20

- Inner Nodes
  - 10
  - ≥10
  - <35
  - ≥35
  - 10
  - 20
  - 31
  - 38
  - 44

- Leaf Nodes
  - 6
  - 10
  - 20
  - 38

- <value>|<key>

- CMU-DB
15-445/645 (Fall 2023)
Every B+Tree node is comprised of an array of key/value pairs.
→ The keys are derived from the attribute(s) that the index is based on.
→ The values will differ based on whether the node is classified as an **inner node** or a **leaf node**.

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

Store all **NULL** keys at either first or last leaf nodes.
B+TREE LEAF NODES

B+Tree Leaf Node

PageID

Prev

K1

V1

... 

Kn

Vn

Next

PageID
B+TREE LEAF NODES

B+Tree Leaf Node

Key + Value

Prev K1 V1 ... Kn Vn Next

PageID

PageID

PageID
B+TREE LEAF NODES

B+Tree Leaf Node

PageID

Prev

\[ K1 \]

\[ \cdots \]

Key+Value

Next

PageID

Key

Value

K1

Kn

\[ \cdots \]

PageID
**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_

• • •
LEAF NODE VALUES

Approach #1: Record IDs
→ A pointer to the location of the tuple to which the index entry corresponds.

Approach #2: Tuple Data
→ AKA Index-Organized Storage
→ The leaf nodes store the actual contents of the tuple.
→ Secondary indexes must store the Record ID as their values.
**B-TREE VS. B+TREE**

The original **B-Tree** from 1972 stored keys and 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 \( L \).
Insert 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.
B+TREE VISUALIZATION

https://cmudb.io/btree

Source: David Gales (Univ. of San Francisco)
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,
→ 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
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 \((a=1 \ AND \ b=2 \ AND \ c=3)\)
- Supported: \((a=1 \ AND \ b=2)\)
- Supported: \((b=2), (c=3)\)

Not all DBMSs support this.
For a hash index, we must have all attributes in search key.
SELECTION CONDITIONS

Find Key=(1,2)

1 ≤ 1
2 ≤ 3

1,3  2,2  3,3

1,1  1,2
1,3  2,1
2,2  2,3
3,3  3,4  4,1
SELECTION CONDITIONS

Find Key=(1,2)
Find Key=(1,∗)
SELECTION CONDITIONS

Find Key=(1,2)
Find Key=(1,*)
SELECTION CONDITIONS

Find Key=(1,2)
Find Key=(1,*)
SELECTION CONDITIONS

Find Key=(1,2)
Find Key=(1,*)
Find Key=(*,1)
Find Key=(1,2)
Find Key=(1,*)
Find Key=(*,1)
Approach #1: Append Record ID
→ Add the tuple's unique Record ID as part of the key to ensure that all keys are unique.
→ The DBMS can still use partial keys to find tuples.

Approach #2: Overflow Leaf Nodes
→ Allow leaf nodes to spill into overflow nodes that contain the duplicate keys.
→ This is more complex to maintain and modify.
B+TREE – APPEND RECORD ID

Insert 6

<Key,RecordId>
B+Tree – Append Record ID

Insert <6, (Page, Slot)>

<Key, RecordId>
B+TREE – APPEND RECORD ID

Insert \(<6,(\text{Page,Slot})>\)
B+TREE – APPEND RECORD ID

Insert <6,(Page,Slot)>

<Key,RecordId>
B+TREE – OVERFLOW LEAF NODES

Insert 6

![Diagram showing the process of inserting 6 into a B+ tree]

- The node contains values 5 and 9.
- The new value 6 is inserted into the node.
- The node now contains values 1, 3, 6, 7, 8, and 9.
- The node is split into two nodes, with the new node containing values 1, 3, 6, 7, 8, and the original node containing values 5 and 9.

The diagram illustrates the insertion process and the resulting structure of the B+ tree node.
B+TREE – OVERFLOW LEAF NODES

Insert 6
B+TREE – OVERFLOW LEAF NODES

Insert 6

Insert 7
Insert 6
Insert 7
Insert 6
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 does not contain a primary key, the DBMS will automatically make a hidden primary key.

Other DBMSs cannot use them at all.
CLUSTERED B+TREE

Traverse to the left-most leaf page and then retrieve tuples from all leaf pages.

This will always be better than sorting data for each query.
Retrieving tuples in the order they appear in a non-clustered index is inefficient due to redundant reads.

A better approach is to find all the tuples that the query needs and then sort them based on their page ID. The DBMS retrieves each page once.
B+TREE DESIGN CHOICES

Node Size
Merge Threshold
Variable-Length Keys
Intra-Node Search
NODE SIZE

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 they are half full.
→ Average occupancy rate for B+Tree nodes is 69%.

Delaying a merge operation may reduce the amount of reorganization.
It may also be better to just let smaller nodes exist and then periodically rebuild entire tree.

This is why PostgreSQL calls their B+Tree a "non-balanced" B+Tree (nbtree).
**VARIABLE-LENGTH KEYS**

**Approach #1: Pointers**
→ Store the keys as pointers to the tuple’s attribute.
→ Also called **T-Trees** (in-memory DBMSs)

**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.
INTRA-NODE SEARCH

Approach #1: Linear
→ Scan node keys from beginning to end.
→ Use SIMD to vectorize comparisons.

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.
→ Use SIMD to vectorize comparisons.

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.
→ Use SIMD to vectorize comparisons.

_\texttt{mm\_cmpeq\_epi32\_mask(a, b)}
INTRA-NODE SEARCH

Approach #1: Linear
→ Scan node keys from beginning to end.
→ Use SIMD to vectorize comparisons.
INTRA-NODE SEARCH

Approach #1: Linear
→ Scan node keys from beginning to end.
→ Use SIMD to vectorize comparisons.

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

Approach #1: Linear
→ Scan node keys from beginning to end.
→ Use SIMD to vectorize comparisons.

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

Approach #1: Linear
→ Scan node keys from beginning to end.
→ Use SIMD to vectorize comparisons.

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

Approach #1: Linear
→ Scan node keys from beginning to end.
→ Use SIMD to vectorize comparisons.

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.

Offset: \((8-4)*7/(10-4)=4\)
OPTIMIZATIONS

Prefix Compression
Deduplication
Suffix Truncation
Pointer Swizzling
Bulk Insert
Buffered Updates
Many more…
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: rob

<table>
<thead>
<tr>
<th>robbed</th>
<th>robbing</th>
<th>robot</th>
</tr>
</thead>
<tbody>
<tr>
<td>bed</td>
<td>bing</td>
<td>ot</td>
</tr>
</tbody>
</table>
DEDUPLICATION

Non-unique indexes can end up storing multiple copies of the same key in leaf nodes.

The leaf node can store the key once and then maintain a "posting list" of tuples with that key (similar to what we discussed for hash tables).
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.
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.
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.
The fastest way to build a new B+Tree for an existing table 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 way to build a new B+Tree for an existing table 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
OBSERVATION

Modifying a B+tree is expensive when the DBMS has to split/merge nodes.
→ Worst case is when DBMS reorganizes the entire tree.
→ The worker that causes a split/merge is responsible for doing the work.

What if there was a way to delay updates and then apply multiple changes together in a batch?
Instead of immediately applying updates, store changes to key/value entries in log buffers at inner nodes. → Also known as \textit{Bε-trees}.

Updates cascade down to lower nodes incrementally when buffers get full.
Instead of immediately applying updates, store changes to key/value entries in log buffers at inner nodes. → Also known as $B\varepsilon$-trees.

Updates cascade down to lower nodes incrementally when buffers get full.
Instead of immediately applying updates, store changes to key/value entries in log buffers at inner nodes. → Also known as Bε-trees.

Updates cascade down to lower nodes incrementally when buffers get full.
WRITE-OPTIMIZED B+TREE

Instead of immediately applying updates, store changes to key/value entries in log buffers at inner nodes.
→ Also known as Bε-trees.

Updates cascade down to lower nodes incrementally when buffers get full.

Find 10

Insert 7   Delete 10

20

10

35

6  10  20  38

10

20
Instead of immediately applying updates, store changes to key/value entries in log buffers at inner nodes. → Also known as Bε-trees.

Updates cascade down to lower nodes incrementally when buffers get full.
DEMO

B+Tree vs. Hash Indexes
Table Clustering
CONCLUSION

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

Index Concurrency Control