Lecture #08

B+Tree Index
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)
- **B_{e}-Tree** (2003)
- **B_{w}-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)
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 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 658-679). 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

→ 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)
→ B\varepsilon-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 $O(\log_f n)$.

→ 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.

→ $f$ is the fanout of the tree.
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 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

Inner Nodes

Leaf Nodes
B+TREE EXAMPLE

<node*> | <key>

Root Node

Inner Nodes

Leaf Nodes
B+TREE EXAMPLE

Root Node

Inner Nodes

Leaf Nodes

<node*> | <key>

<value> | <key>
**B+TREE EXAMPLE**

- **Root Node**: 20
- **Inner Nodes**:
  - 10
  - 35
- **Leaf Nodes**:
  - 6
  - 10
  - 20, 31
  - 38, 44

The diagram shows a B+ tree with nodes and keys.
B+TREE EXAMPLE

- **<node*> | <key>**
  - Root Node
  - Sibling Pointers
  - Inner Nodes
  - Leaf Nodes

- **<value> | <key>**

Diagram:
- Root Node: 20
- Inner Nodes: 10, 35
- Leaf Nodes: 6, 10, 20, 31, 38, 44
- Sibling Pointers:
  - <20
  - ≥20
- Key comparisons:
  - <10
  - ≥10
  - <35
  - ≥35
**B+TREE EXAMPLE**

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

**Root Node**

- **<node*> | <key>**
  - Value: 20

**Sibling Pointers**

- **<20**
- **≥20**

**Inner Nodes**

- **<10**
- **≥10**
- **<35**
- **≥35**

**Leaf Nodes**

- **<value> | <key>**
- **<value> | <key>**
- **<value> | <key>**

- **<value> | <key>**
- **<value> | <key>**
- **<value> | <key>**

Also called non-leaf nodes
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

prev  K1  V1  \ldots  Kn  Vn  next
B+TREE LEAF NODES

B+Tree Leaf Node

PageID

Prev  $K1 \ V1$  …  $Kn \ Vn$  Next

PageID
B+TREE LEAF NODES

B+Tree Leaf Node

PageID

Prev

Key + Value

Next

PageID

K1 V1

Kn Vn
B+TREE LEAF NODES

B+Tree Leaf Node

PageID

Prev

Key+Value

Next

K1

Kn

PageID

B+Tree Leaf Node

Key

Value

K1

Kn

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

K1  K2  K3  K4  K5  ⋯  Kn

Values

⋯
B+TREE LEAF NODES

B+Tree Leaf Node

- Sorted Keys: K1, K2, K3, K4, K5, ..., Kn
- Values: corresponding values associated with the keys
- Level: number of levels in the tree
- Slots: number of slots in the node
- Prev: previous node
- Next: next node

Diagram showing the structure of a B+ tree with leaf nodes containing sorted keys and 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 $L_2$
  → Distribute 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 – INSERT
B+TREE – INSERT

Insert 6
B+TREE – INSERT

Insert 6
B+TREE – INSERT

Insert 6

Insert 6.
B+TREE – INSERT

Insert 6
B+TREE – INSERT

Insert 6

Diagram showing the insertion process in a B+ tree.
B+TREE – INSERT

Insert 6
B+TREE – INSERT

Insert 6
**B+TREE – INSERT**

**Insert 6**
B+TREE - INSERT

Insert 6
B+TREE – INSERT

Insert 6
B+TREE - INSERT

Insert 6

Diagram: B+Tree with keys 1, 3, 4, 5, 6, 9, 10, 12, 13, 9, 10, 9, 12. The ranges [4, 9) and [9, 12) are shown, with 6 inserted into the tree.
B+TREE - INSERT

Insert 6
Insert 8
INSERT THE KEY 17

Note: new example/tree.
INSERT THE KEY 17

Note: new example/tree.
NEXT, INSERT THE KEY 16
Next, insert the key 16

No space in the node where the new key “belongs”.

NEXT, INSERT THE KEY 16

Split the node!
Copy the middle key.
Push the key up.
NEXT, INSERT THE KEY 16

New node.
Shuffle keys from the node that triggered the split.
NEXT, INSERT THE KEY 16

New node.
Shuffle keys from the node that triggered the split.
NEXT, INSERT THE KEY 16

But, this is an “orphan” node. No parent node points to it.
NEXT, INSERT THE KEY 16

But, this is an “orphan” node. No parent node points to it.
NEXT, INSERT THE KEY 16

Want to create a key, pointer pair like this. But can’t insert it in the root node, which is full.

But, this is an “orphan” node. No parent node points to it.
Next, insert the key 16

Want to create a key, pointer pair like this. But can’t insert it in the root node, which is full. **Split the root. Grow the tree!**

But, this is an “orphan” node. No parent node points to it.
NEXT, INSERT THE KEY 16

Split the root. Grow the tree!
NEXT, INSERT THE KEY 16

Split the root. Grow the tree!
Next, need to split the “old” root, then point to the split nodes from the new root.
NEXT, INSERT THE KEY 16
NEXT, INSERT THE KEY 16
NEXT, INSERT THE KEY 16
**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
DELETE THE KEY 6
DELETE THE KEY 6

Borrow from a “rich” neighbor.
DELETE THE KEY 6

Borrow from a “rich” neighbor.
Could borrow from either neighbor.
DELETE THE KEY 6

Borrow from a “rich” neighbor.
Could borrow from either neighbor.
DELETE THE KEY 6

Borrow from a “rich” neighbor.
Could borrow from either neighbor.
DELETE THE KEY 6

Borrow from a “rich” neighbor.
Could borrow from either neighbor.
DELETE THE KEY 15

Note: new example/tree.
DELETE THE KEY 15

Note: new example/tree.
DELETE THE KEY 15
DELETE THE KEY 15
NEXT, DELETE THE KEY 19
NEXT, DELETE THE KEY 19
NEXT, DELETE THE KEY 19

Under-filled.
No “rich” neighbors to borrow.
Merge with a sibling
NEXT, DELETE THE KEY 19

This node is under-filled! Pull-down.
NEXT, DELETE THE KEY 19

This node is under-filled!
Pull-down.
NEXT, DELETE THE KEY 19

The tree has shrunk in height.
Composite Index: The key is composed of multiple attributes.

```
CREATE INDEX LFM_name ON artist
    (last_name, first_name, middle_names NULLS FIRST);
```

Can use a B+Tree index if the query provides a “prefix” of composite key. Example: Index on \(<a, b, c>\)

→ Supported: \((a=1 \text{ AND } b=2 \text{ AND } c=3)\)
→ Supported: \((a=1 \text{ AND } b=2)\)
→ NOT (generally) supported: \((b=2), (c=3)\)

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

Find Key=(1,2)
SELECTION CONDITIONS

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

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

Diagram: Tree structure with keys 1,1, 1,2, 1,3, 2,1, 2,2, 2,3, 3,1, 3,2, 3,3, 3,4, 4,1.
SELECTION CONDITIONS

Find Key=(1,2)

Find Key=(1,*)
SELECTION CONDITIONS

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

1 ≤ 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)
SELECTION CONDITIONS

Find Key=(1,2)
Find Key=(1,*)
Find Key=(*,1)
B+TREE – DUPLICATE KEYS

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**
- **Insert <6, (Page,Slot)>**
B+TREE – APPEND RECORD ID

Insert 6

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

Insert <6,(Page,Slot)>

<Key,RecordId>
Insert \((6, (\text{Page}, \text{Slot}))\)

\text{B+TREE} – APPEND RECORD ID
B+TREE – APPEND RECORD ID

Insert <6,(Page,Slot)>

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

Insert <6, (Page, Slot)>

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

Insert <6,(Page,Slot)>

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

Insert 6

Diagram showing a B+tree with nodes containing keys 1, 3, 5, 9, 6, 7, 8, and 9, 13. The diagram illustrates the insertion of the key 6, which causes overflow in the leaf node.
B+TREE – OVERFLOW LEAF NODES

Insert 6
B+TREE – OVERFLOW LEAF NODES

Insert 6
B+TREE – OVERFLOW LEAF NODES

Insert 6

Insert 7
B+TREE - OVERFLOW LEAF NODES

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.

Table Pages

101 102 103 104
**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.
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.
INDEX SCAN PAGE SORTING

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.
INDEX SCAN PAGE SORTING

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.
INDEX SCAN PAGE SORTING

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.
INDEX SCAN PAGE SORTING

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.
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
MERGE THRESHOLD

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).
VAR\text{I}A\text{L}-\text{LEN}\text{T}H\text{E}H\text{E}K\text{Y}S

\textbf{Approach \#1: Pointers}
\begin{itemize}
  \item Store the keys as pointers to the tuple’s attribute.
  \item Also called \textit{T-Trees} (in-memory DBMSs)
\end{itemize}

\textbf{Approach \#2: Variable-Length Nodes}
\begin{itemize}
  \item The size of each node in the index can vary.
  \item Requires careful memory management.
\end{itemize}

\textbf{Approach \#3: Padding}
\begin{itemize}
  \item Always pad the key to be max length of the key type.
\end{itemize}

\textbf{Approach \#4: Key Map / Indirection}
\begin{itemize}
  \item 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

4 5 6 7 8 9 10

46
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 |

Find Key=8
INTRA-NODE SEARCH

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

Find Key=8

\[4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10\]
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 |

_mm_cmpeq_e132_mask(a, b)
INTRA-NODE SEARCH

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

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>

_\_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.
INTRA-NODE SEARCH

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

_mm_cmpeq_epi32_mask(a, b)
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.

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

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

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.
→ 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: \((8-4)*7/(10-4)=4\)
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.
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.

Efficiently Searching In-Memory Sorted Arrays:
Revenge of the Interpolation Search?

Peter Van Sandt, Yannis Chronis, Jignesh M. Patel
Department of Computer Sciences, University of Wisconsin-Madison
{van-sandt,chronis,patel}@cs.wisc.edu

ABSTRACT
In this paper, we focus on the problem of searching sorted, in-memory datasets. This is a key data operation, and Binary Search is the de facto algorithm that is used in practice. We consider an alternative, namely Interpolation Search, which can take advantage of hardware trends by using complex calculations to save memory accesses. Historically, Interpolation Search was found to underperform compared to other search algorithms in this setting, despite its superior asymptotic complexity. Also, Interpolation Search is known to perform poorly on non-uniform data. To address these issues, we introduce SSP (Stepped Linear Interpolation), an optimized implementation of Interpolation Search, and TIP (Three point Interpolation), a new search algorithm that uses linear functions to interpolate on non-uniform distributions. We evaluate these two algorithms against a similarly optimized Binary Search method using a variety of real and synthetic datasets. We show that SSP is up to 6 times faster on uniformly distributed data and TIP is 2-3 times faster on non-uniformly distributed data in some cases. We also design a meta algorithm to switch between these different methods to automate picking the higher performing search algorithm, which depends on factors like data distribution.

CCS CONCEPTS
→ Information systems → Point lookups. Main memory expires.

KEYWORDS
→ In-memory search → Interpolation Search → Binary Search

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2023 Copyright by the owner/author. Publication rights licensed to ACM.

ACM 978-1-4503-8061-5/23/06...
https://doi.org/10.1145/3399800.3399797

1 INTRODUCTION
Searching in-memory, sorted datasets is a fundamental data operation [21]. Today, Binary Search is the de facto search method that is used in practice, as it is efficient and asymptotically optimal in the worst-case algorithm. Binary Search is a primitive in many popular data systems and frameworks e.g. LevelDB [20] and Pandas [13]. Designing algorithms around hardware trends can yield significant performance gains. A key technological trend is the diverging CPU and memory speeds, which is illustrated in Figure 1. This trend favors algorithms that can use more computation to reduce memory accesses [4, 6, 11, 27, 56]. The focus of this paper is on exploring the impact of this trend.
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.
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).
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.
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.
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.
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 $\text{Be}$-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.
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.
Instead of immediately applying updates, store changes to key/value entries in log buffers at inner nodes. → Also known as **Be-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.
Instead of immediately applying updates, store changes to key/value entries in log buffers at inner nodes. → Also known as \textbf{Bε-trees}.

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

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

Index Concurrency Control