Tree Indexes
ADMINISTRIVIA

Project #1 is due Sunday, Sept 26\textsuperscript{th} @11:59pm

Homework #2 is due Sunday, Oct 3\textsuperscript{rd} @11:59pm
DATA STRUCTURES

Internal Meta-data
Core Data Storage
Temporary Data Structures
Table Indexes
DATA STRUCTURES

Internal Meta-data
Core Data Storage
Temporary Data Structures
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 those attributes.

The DBMS ensures that the contents of the table and the index are logically synchronized.
It is the DBMS's job to figure out the best index(es) to use to execute each query.

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

B+Tree Overview
Use in a DBMS
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)
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)
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)
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}\text{-Tree}** (1981)
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, since 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 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
B+TREE EXAMPLE

Leaf Nodes
B+Tree Example

Inner Node

Leaf Nodes

1 3
5 9
6 7
9 13
B+TREE EXAMPLE

**Inner Node**

**Sibling Pointers**

**Leaf Nodes**
B+TREE EXAMPLE

Leaf Nodes

Inner Node

Sibling Pointers

Leaf Nodes
B+TREE EXAMPLE

Inner Node

Leaf Nodes

Sibling Pointers

<node*> | <key>

<5 <9 ≥9

1 3 6 7 9 13
B+TREE EXAMPLE

Leaf Nodes

Inner Node

Sibling Pointers

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.
B+TREE LEAF NODES
B+TREE LEAF NODES

B+Tree Leaf Node

Prev

K1 V1

• • •

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

Key + Value

Prev K1 V1 \( \cdots \) Kn Vn Next

PageID

CMU-DB

15-445/645 (Fall 2021)
B+TREE LEAF NODES

B+Tree Leaf Node

PageID

Prev

Key+Value

Next

PageID
**B+TREE LEAF NODES**

**B+Tree Leaf Node**

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

**Sorted Keys**

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

**Values**

- •, •, •, •, •, ..., •
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
→ The leaf nodes store the actual contents of the tuple.
→ Secondary indexes must store the Record ID as their 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**
- The leaf nodes store the actual contents of the tuple.
- Secondary indexes must store the Record ID as their values.
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.
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 a hash index, we must have all attributes in search key.
SELECTION CONDITIONS
SELECTION CONDITIONS

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

A ≤ A
B ≤ C

A, A
A, B
A, C
B, A
B, B
B, C
B, B
B, C
C, C
C, C
C, D
Find Key=(A,B)

A ≤ A
B ≤ C

A,C
B,B
C,C

A,A
A,B

A,C
B,A

B,B
B,C

C,C
C,D
Find Key=(A,B)
Find Key=(A,B)
Find Key=(A,∗)
**SELECTION CONDITIONS**

Find Key=(A,B)

Find Key=(A,*)

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

(A,*) ≤ (B,*)
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 $L_2$
\[ \rightarrow \] Redistribute entries evenly, copy up middle key.
\[ \rightarrow \] 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 – 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 $L$. 
**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

Diagram showing a B+ tree with keys 1, 3, 5, 6, 7, 8, 9, and 13. The tree has branches with keys less than 5, less than 9, and greater than or equal to 9.
B+TREE – APPEND RECORD ID

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

Insert 6

<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 – 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 insertion of 6 into a B+Tree with overflow conditions for keys less than 5, less than 7, and greater than or equal to 9.
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
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.
Traverse to the left-most leaf page and then retrieve tuples from all leaf pages.

This will always be better than external sorting.
Traverse to the left-most leaf page and then retrieve tuples from all leaf pages.

This will always be better than external sorting.
Traverse to the left-most leaf page and then retrieve tuples from all leaf pages.

This will always be better than external sorting.
Retrieving tuples in the order they appear in a non-clustered index can be very inefficient.

The DBMS can first figure out all the tuples that it needs and then sort them based on their Page ID.
Retrieving tuples in the order they appear in a non-clustered index can be very inefficient.

The DBMS can first figure out all the tuples that it needs and then sort them based on their Page ID.
Retrieving tuples in the order they appear in a non-clustered index can be very inefficient.

The DBMS can first figure out all the tuples that it needs and then sort them based on their Page ID.
Retrieving tuples in the order they appear in a non-clustered index can be very inefficient.

The DBMS can first figure out all the tuples that it needs and then sort them based on their Page ID.
Retrieving tuples in the order they appear in a non-clustered index can be very inefficient.

The DBMS can first figure out all the tuples that it needs and then sort them based on their Page ID.
Retrieving tuples in the order they appear in a non-clustered index can be very inefficient.

The DBMS can first figure out all the tuples that it needs and then sort them based on their Page ID.
B+TREE DESIGN CHOICES

Node Size
Merge Threshold
Variable-Length Keys
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 they are half full.

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.
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.
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

Find Key=8

4 5 6 7 8 9 10

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
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
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

4 5 6 7 **8** 9 10

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**

Offset: $(8-4)*7/(10-4)=4$
**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 |

Offset: \((8-4)\times7/(10-4)=4\)
OPTIMIZATIONS

Prefix Compression
Deduplication
Bulk Insert
Many more…
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.
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.
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 list of tuples with that key (similar to what we discussed for hash tables).
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 list of tuples with that key (similar to what we discussed for hash tables).
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 list of tuples with that key (similar to what we discussed for hash tables).
BULK INSERT

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

Index Concurrency Control