null
Project #1 is due Sunday Sept 27th
Homework #2 is due Sunday Oct 4th
UPCOMING DATABASE TALKS

CockroachDB Query Optimizer
→ Monday Sept 28th @ 5pm ET

Apache Arrow
→ Monday Oct 5th @ 5pm ET

DataBricks Query Optimizer
→ Monday Oct 12th @ 5pm ET
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 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
Using B+Trees in a DBMS
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)
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 in 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

Leaf Nodes
<5
<9
≥9

Inner Node
<value>|<key>

Sibling Pointers
<node*>&<key>

Leaf Nodes
<value>|<key>
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  K1  V1  • • •  Kn  Vn  Next
B+Tree Leaf Nodes

B+Tree Leaf Node

Page ID

Prev

K1 V1 • • • Kn Vn

Next

Page ID
B+TREE LEAF NODES

B+Tree Leaf Node

Key+Value

Prev

Next

PageID

PageID

K1

Kn

15-445/645 (Fall 2020)
B+TREE LEAF NODES

**B+Tree Leaf Node**

- **Level**: 
  - #
  - #
- **Slots**: 
  - #
- **Prev**: 
  - •
- **Next**: 
  - •
- **Sorted Keys**: 
  - K1
  - K2
  - K3
  - K4
  - K5
  - •
- **Values**: 
  - •
  - •
  - •
  - •
  - •
  - •
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 must store the record id as their 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.
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: \((a=5 \ \text{AND} \ b=3)\)
- 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)

\[ A \leq A \]
\[ B \leq C \]
SELECTION CONDITIONS

Find Key=(A,B)

Find Key=(A,*)
SELECTION CONDITIONS

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

Find Key=(A,B)
Find Key=(A, *)
Find Key=(*, A)
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 \( L2 \)
\( \rightarrow \) Redistribute entries evenly, copy up middle key.
\( \rightarrow \) Insert index entry pointing to \( L2 \) 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/btreet

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+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

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

Insert <6, (Page, Slot)>

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

Insert \(<6, (Page, Slot)\)>

\(<\text{Key, RecordId}>\>
B+TREE – OVERFLOW LEAF NODES

Insert 6
**B+TREE – OVERFLOW LEAF NODES**

Insert 6

Insert 7

![Diagram showing B+Tree operations with overflow 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 row id 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 better than external sorting.
HEAP CLUSTERING

Tuples are sorted in the heap's pages using the order specified by a clustering index.

If the query accesses tuples using the clustering index's attributes, then the DBMS can jump directly to the pages that it needs.
HEAP CLUSTERING

Tuples are sorted in the heap's pages using the order specified by a clustering index.

If the query accesses tuples using the clustering index's attributes, then the DBMS can jump directly to the pages that it needs.
INDEX SCAN PAGE SORTING

Retrieving tuples in the order that appear in an unclustered index is inefficient.

The DBMS can first figure out all the tuples that it needs and then sort them based on their page id.
INDEX SCAN PAGE SORTING

Retrieving tuples in the order that appear in an unclustered index is inefficient.

The DBMS can first figure out all the tuples that it needs and then sort them based on their page id.
INDEX SCAN PAGE SORTING

Retrieving tuples in the order that appear in an unclustered index is inefficient.

The DBMS can first figure out all the tuples that it needs and then sort them based on their page id.
DEMO

B+Tree vs. Hash Indexes
Table Clustering
CONCLUSION

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

More B+Trees
Tries / Radix Trees
Inverted Indexes