Order-Preserving Trees
ADMINISTRIVIA

Project #1 is due Monday
October 2\textsuperscript{nd} @ 11:59pm

Homework #3 is due Wednesday
October 4\textsuperscript{th} @ 11:59pm
STATUS

We are now going to talk about how to support the DBMS's execution engine to read/write data from pages.

Two types of data structures:
→ Hash Tables
→ Trees
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 columns that are organized and/or sorted for efficient access using a subset of those columns.

The DBMS ensures that the contents of the table and the index are always in sync.
TABLE INDEXES

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
Skip List
Radix Tree
Extra Index Stuff
B-TREE FAMILY

There is a specific data structure called a B-Tree, but then people also use the term to generally refer to a class of data structures.

→ B-Tree
→ B+Tree
→ B\text{link}-Tree
→ 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 inner 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 OVERVIEW

Inner Node

Sibling Pointers

Leaf Nodes
B+TREE OVERVIEW

Inner Node

Sibling Pointers

Leaf Nodes

<value>/<key>
**B+Tree Nodes**

Every node in the B+Tree contains an array of key/value pairs.
→ The keys will always be the column or columns that you built your index on
→ The values will differ based on whether the node is classified as **inner nodes** or **leaf nodes**.

The arrays are always kept in sorted order.
**B+TREE: 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.
B+TREE LEAF NODES

B+Tree Leaf Node

Prev K1 V1 • • • Kn Vn Next

PageID

PageID
B+TREE LEAF NODES

B+Tree Leaf Node

Prev

K1

• • •

Kn

Next

Key+Value

Prev

Next

PageID

PageID
B+TREE LEAF NODES

B+Tree Leaf Node

Level Slots Prev Next
# # • • •

Sorted Keys
K1 K2 K3 K4 K5 • • • Kn

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.
**B+TREE: INSERT**

Find correct leaf \( L \).

Put data entry into \( L \) in sorted order.

→ If \( L \) has enough space, done!
→ Else, must split \( L \) 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

http://cmudb.io/btree

https://www.cs.usfca.edu/~gall es/visualization/BPlusTree.html
**B+TREE: DELETE**

Start at root, find leaf $L$ where entry belongs.

Remove the entry.

$\rightarrow$ If $L$ is at least half-full, done!

$\rightarrow$ 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
**B+ TREES IN PRACTICE**

Typical Fill-Factor: 67%.
→ Average Fanout = 2*100*0.67 = 134

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
B+TREE DESIGN CHOICES

Merge Threshold
Non-Unique Indexes
Variable Length Keys
Prefix Compression
B+TREE: MERGE THRESHOLD

Some DBMSs don't always merge nodes when it is half full.

Delaying a merge operation may reduce the amount of reorganization.
B+TREE: 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.
B+TREE: DUPLICATE KEYS

B+Tree Leaf Node

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

**Sorted Keys**
- K1
- K1
- K1
- K2
- K2
- Kn

**Values**
- φ
- φ
- φ
- φ
- φ
- φ
- φ
- φ
- φ
B+TREE: VALUE LISTS

B+Tree Leaf Node

Level | Slots | Prev  | Next
---    | ---    | ---   | ---
#      | #      | ⊗     | ⊗

Sorted Keys

K1     | K2     | K3     | K4     | K5     | … | Kn

Values

•     | •     | •
•     | •
•     |
B+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 B+Tree can vary.
→ Requires careful memory management.

Approach #3: Key Map
→ Embed an array of pointers that map to the key + value list within the node.
B+TREE: PREFIX COMPRESSION

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

Store a minimum prefix that is needed to correctly route probes into the index.
B+TREE: PREFIX COMPRESSION

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

Store a minimum prefix that is needed to correctly route probes into the index.
B+TREE BULK INSERT

The fastest/best way to build a B+Tree is to first sort the keys and then build the index from the bottom up.
B+TREE BULK INSERT

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
**B+TREE BULK INSERT**

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
B+TREE BULK INSERT

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
**OBSERVATION**

The easiest way to implement a **dynamic** order-preserving index is to use a sorted linked list.

All operations have to linear search.

→ Average Cost: O(N)
The easiest way to implement a **dynamic** order-preserving index is to use a sorted linked list.

All operations have to linear search.

→ Average Cost: O(N)
Skip Lists: A Probabilistic Alternative to Balanced Trees

Invented in 1990.

Multiple levels of linked lists with extra pointers that skip over intermediate nodes.

Maintains keys in sorted order without requiring global rebalancing.
SKIP LISTS

A collection of lists at different levels
→ Lowest level is a sorted, singly linked list of all keys
→ 2nd level links every other key
→ 3rd level links every fourth key
→ In general, a level has half the keys of one below it

To insert a new key, flip a coin to decide how many levels to add the new key into. Provides approximate $O(\log n)$ search times.
SKIP LISTS: EXAMPLE

Levels

End

P=N

P=N/2

P=N/4

K1 V1

K2 V2

K3 V3

K4 V4

K6 V6

∞

∞

∞

∞
SKIP LISTS: EXAMPLE

Levels

- $P = N/4$
- $P = N/2$
- $P = N$

End

- $\infty$

- $P = N/4$
- $P = N/2$
- $P = N$

$K_1$, $V_1$
$K_2$, $V_2$
$K_3$, $V_3$
$K_4$, $V_4$
$K_6$, $V_6$

$\infty$
**SKIP LISTS: EXAMPLE**

- **Levels**
  - $P = N/4$
  - $P = N/2$
  - $P = N$

- **End**
  - $\infty$

The diagram illustrates a skip list with levels $P = N/4$, $P = N/2$, and $P = N$, leading to the end $\infty$. The keys are labeled as $K1$, $V1$, $K2$, $V2$, $K3$, $V3$, $K4$, $V4$, $K6$, and $V6$.
**SKIP LISTS: EXAMPLE**

Levels

- $P = N/4$
- $P = N/2$
- $P = N$

End

- $\infty$
- $\infty$
- $\infty$

$K_1$  $V_1$  $K_2$  $V_2$  $K_3$  $V_3$  $K_4$  $V_4$  $K_6$  $V_6$
SKIP LISTS: INSERT

Insert K5

Levels

End

P=N/4

P=N/2

P=N

K1

V1

K2

V2

K2

V2

K3

V3

K4

V4

K4

V4

K6

V6

∞

∞

∞

∞
**SKIP LISTS: INSERT**

Insert K5

Levels

- **P=N** (Root node)
  - K1
  - V1

- **P=N/2**
  - K2
  - V2

- **P=N/4**
  - K3
  - V3

- **P=N/2**
  - K4
  - V4

- **P=N/4**
  - K5
  - V5

End

- **∞**
  - K6
  - V6
**SKIP LISTS: INSERT**

**Insert K5**

**Levels**

- **P=N**
  - **K1**
  - **V1**
  - **K2**
  - **V2**
  - **K3**
  - **V3**
  - **K4**
  - **V4**

- **P=N/2**
  - **K2**
  - **V2**

- **P=N/4**
  - **K1**
  - **V1**

**End**

- **K5**
- **V5**
- **K6**
- **V6**

CMU 15-445/645 (Fall 2017)
SKIPPED LISTS: INSERT

Insert K5

Levels

P=N
K1
V1

P=N/2
K2
V2

P=N/4
K3
V3

K4
V4

K5

K6
V6

End
∞
∞
∞
SKIP LISTS: SEARCH

Levels

Find K3

End

P=N

K1
V1

P=N/2

K2
V2

P=N/4

K2

K3
V3

K4
V4

K5

K5

K6
V6

∞

∞

∞

P=N/2

K5

K5

K5

K5

CMU 15-445/645 (Fall 2017)

34
SKIP LISTS: SEARCH

Levels

Find K3

End

K3 < K5

P = N/4

K1
V1

K2
V2

K3
V3

K4
V4

K5
V5

K6
V6

K5

∞
**SKIP LISTS: SEARCH**

**Find K3**

- **Levels**
  - $P = N$
  - $K_1 < V_1$
  - $K_2 < V_2$
  - $K_3 < V_3$
  - $K_4 < V_4$
  - $K_5 < V_5$
  - $K_6 < V_6$

- **End**
  - $\infty$

- **Conditions**
  - $K_3 < K_5$
  - $K_3 > K_2$

---

CMU 15-445/645 (Fall 2017)
**SKIP LISTS: SEARCH**

Find K3

Levels

- K3 < K5
- K3 > K2
- K3 < K4

End

- P = N
- P = N/2
- P = N/4
**SKIP LISTS: SEARCH**

Find $K3$

Levels

$P=N$

$K3<K4$

$K3>K2$

$K3<K5$

$P=N/2$

$P=N/4$

End

$\infty$

$\infty$

$\infty$

$K6$

$V6$

$K5$

$V5$

$K4$

$V4$

$K2$

$V2$

$K1$

$V1$
SKIP LISTS: DELETE

First **logically** remove a key from the index by setting a flag to tell threads to ignore.

Then **physically** remove the key once we know that no other thread is holding the reference.
SKIP LISTS: DELETE

Delete K5

Levels

P=N

P=N/2

P=N/4

End

K1

V1

Del false

K2

V2

Del false

K3

V3

Del false

K4

V4

Del false

K5

V5

Del false

K6

V6

Del false

∞

∞

∞
SKIP LISTS: DELETE

Delete K5

Levels

P=N/4

P=N/2

P=N

End

K1
V1

Del
false

K2
V2

Del
false

K3
V3

Del
false

K4
V4

Del
false

K5

Del
true

K5

Del
false

K6
V6

Del
false

∞

∞

∞

∞
SKIP LISTS: DELETE

Levels

Delete K5

End

P=N

P=N/4

K1

V1

Del false

K2

V2

Del false

K3

V3

Del false

K4

V4

Del false

K5

V5

Del true

K6

V6

Del false

N

N/4

N/2

∞
Skip Lists: Delete

Delete K5

Levels

- P = N
- P = N/2
- P = N/4

End

K1 V1 Del false
K2 V2 Del false
K3 V3 Del false
K4 V4 Del false
K5 V5 Del true
K6 V6 Del false
K5

∞

Del false

CMU 15-445/645 (Fall 2017)
SKIP LISTS: DELETE

Delete K5

Levels

P=N
K1 V1 Del false
K2 V2 Del false
K3 V3 Del false
K4 V4 Del false
K5
End

P=N/4
K5

P=N/2
K2

Del false

P=N/4
K5

Del false

P=N/2
K5

Del false

P=N
K5

Del true
Skip Lists: Delete

Delete K5

Levels

P=N

K1
V1
Del false

K2
V2
Del false

K3
V3
Del false

K4
V4
Del false

K5

K6
V6
Del false

End

P=N/4

P=N/2

Del false

CMU 15-445/645 (Fall 2017)
SKIP LISTS: ADVANTAGES

Uses less memory than a typical B+Tree if you don’t include reverse pointers.

Insertions and deletions do not require rebalancing.
SKIP LISTS: DISADVANTAGES

Not disk/cache friendly because they do not optimize locality of references.

Invoking random number generator multiple times per insert is slow.

Reverse search is non-trivial.
RADIX TREE

Uses digital representation of keys to examine prefixes one-by-one instead of comparing entire key.
→ The height of the tree depends on the length of keys.
→ Does not require rebalancing
→ The path to a leaf node represents the key of the leaf
→ Keys are stored implicitly and can be reconstructed from paths.
TRIE VS. RADIUS TREE

Keys: HELLO, HAT, HAVE
TRIE VS. RADIX TREE

Trie

Keys: HELLO

HAT, HAVE
TRIE VS. RADIUS TREE

Trie

Keys: HELLO, HAT, HAVE
TRIE VS. RADIX TREE

Trie

Radix Tree

Keys: HELLO, HAT, HAVE
TRIE VS. RADIX TREE

Trie

Radix Tree

Keys: \textbf{HELLO}, HAT, HAVE
RADIX TREE: MODIFICATIONS
RADIX TREE: MODIFICATIONS

Insert HAIR
RADIX TREE: MODIFICATIONS

Insert HAIR
Delete HAT, HAVE
RADIX TREE: MODIFICATIONS

Insert HAIR
Delete HAT, HAVE
RADIUS TREE: MODIFICATIONS

Insert HAIR
Delete HAT, HAVE
RADIX TREE: MODIFICATIONS

Insert HAIR
Delete HAT, HAVE
RADIX TREE: BINARY COMPARABLE KEYS

Not all attribute types can be decomposed into binary comparable digits for a radix tree.

→ **Unsigned Integers:** Byte order must be flipped for little endian machines.

→ **Signed Integers:** Flip two’s-complement so that negative numbers are smaller than positive.

→ **Floats:** Classify into group (neg vs. pos, normalized vs. denormalized), then store as unsigned integer.

→ **Compound:** Transform each attribute separately.
RADIX TREE: BINARY COMPARABLE KEYS

Int Key: 168496141

Hex Key: 0A 0B 0C 0D

Big Endian

Little Endian
RADIUS TREE: BINARY COMPARABLE KEYS

Int Key: 168496141
Hex Key: 0A 0B 0C 0D

Big Endian | Little Endian
RADIX TREE: BINARY COMPARABLE KEYS

Int Key: 168496141
Hex Key: 0A 0B 0C 0D

Big Endian

Little Endian
# Single-threaded Performance

*Data Set: 30m Random 64-bit Integers*

<table>
<thead>
<tr>
<th></th>
<th>B+Tree</th>
<th>Masstree</th>
<th>Skip List</th>
<th>Bw-Tree</th>
<th>Radix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Read-only</strong></td>
<td>5.7</td>
<td>5.8</td>
<td>3.2</td>
<td>2.1</td>
<td>1.9</td>
</tr>
<tr>
<td><strong>Insert-only</strong></td>
<td>4.2</td>
<td>2.2</td>
<td>1.5</td>
<td>3.7</td>
<td>0.1</td>
</tr>
<tr>
<td><strong>Read/Write</strong></td>
<td>4.5</td>
<td>5.1</td>
<td>2.7</td>
<td>6.6</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Scan/Insert</strong></td>
<td>5.1</td>
<td>5.5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: Huanchen Zhang
SELECTION CONDITIONS

The DBMS can use a B+Tree index if the query provides all of the attributes in a prefix of the search key.

→ Index on \(<a, b, c>\) matches \((a=5 \text{ AND } b=3)\), but not \(b=3\).

For Hash index, we must have all attributes in search key.
B+TREE PREFIX SEARCH

Find "XY"
B+TREE PREFIX SEARCH

Find "XY"
Find "_Y"
PARTIAL INDEXES

Create an index on a subset of the entire table. This potentially reduces its size and the amount of overhead to maintain it.

```
CREATE INDEX idx_foo
    ON foo (a, b)
    WHERE c = 'WuTang'
```
PARTIAL INDEXES

Create an index on a subset of the entire table. This potentially reduces its size and the amount of overhead to maintain it.

```
CREATE INDEX idx_foo ON foo (a, b)
WHERE c = 'WuTang'
```

```
SELECT b FROM foo
WHERE a = 123
AND c = 'WuTang'
```
COVERING INDEXES

If all of the fields needed to process the query are available in an index, then the DBMS does not need to retrieve the tuple.

```
CREATE INDEX idx_foo
  ON foo (a, b)

SELECT b FROM foo
WHERE a = 123
```
COVERING INDEXES

If all of the fields needed to process the query are available in an index, then the DBMS does not need to retrieve the tuple.

```
CREATE INDEX idx_foo
ON foo (a, b)

SELECT b FROM foo
WHERE a = 123
```
INDEX INCLUDE COLUMNS

Embed additional columns in indexes to support index-only queries.

Not part of the search key.

CREATE INDEX idx_foo
    ON foo (a, b)
    INCLUDE (c)

SELECT b FROM foo
    WHERE a = 123
    AND c = 'WuTang'
INDEX INCLUDE COLUMNS

Embed additional columns in indexes to support index-only queries.
Not part of the search key.
CONCLUSION

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

Skip Lists and Radix Trees have some interesting properties.

We will cover lock free data structures in 15-721.
Query Processing
→ How to use what we've talked about so far to actually execute queries!