B-TREES
(LOOSELY BASED ON THE COW BOOK: CH. 10)
Motivation

Consider the following table:

```sql
CREATE TABLE Tweets (  
    uniqueMsgID INTEGER,       -- unique message id  
    tstamp TIMESTAMP,     -- when was the tweet posted  
    uid INTEGER,       -- unique id of the user  
    msg VARCHAR (140), -- the actual message  
    zip INTEGER       -- zipcode when posted  
);  
```

Consider the following query, Q1:

```sql
SELECT * FROM Tweets  
WHERE uid = 145;  
```

And, the following query, Q2:

```sql
SELECT * FROM Tweets  
WHERE zip BETWEEN 53000 AND 54999  
```

Ways to evaluate the queries, efficiently?

1. Store the table as a heapfile, scan the file. I/O Cost?
2. Store the table as a sorted file, binary search the file. I/O Cost?
3. Store the table as a heapfile, build an index, and search using the index.
4. Store the table in an index file. The entire tuple is stored in the index!
Index

• Two main types of indices
  – **Hash** index: good for equality search (e.g. Q1)
  – **B-tree** index: good for both range search (e.g. Q2) and equality search (e.g. Q1)
    • Generally a hash index is faster than a B-tree index for equality search

• Hash indices aim to get $O(1)$ I/O and CPU performance for search and insert

• B-Trees have $O(\log_F N)$ I/O and CPU cost for search, insert and delete.
What is in the index

• Two things: **index key** and **some value**
  – Insert(indexKey, value)
  – Search (indexKey) \(\rightarrow\) value (s)

• What is the index key for Q1 and Q2?

• Consider Q3:

```sql
SELECT * FROM Tweets
WHERE uid = 145 AND
  zip BETWEEN 53000 AND 54999
```

• Value:
  – Record id
  – List of record id
  – The entire tuple!
(Ubiquitous) B+ Tree

• Height-balanced (dynamic) tree structure
• Insert/delete at $\log_F N$ cost ($F =$ fanout, $N =$ # leaf pages)
• Minimum 50% occupancy (except for root).
  Each node contains $d \leq m \leq 2d$ entries.
  The parameter $d$ is called the order of the tree.
• Supports equality and range-searches efficiently.

Index Entries
Entries in the index
(i.e. non-leaf) pages:
  (search key value, pageid)

Data Entries
Entries in the leaf pages:
  (search key value, recordid)
Example B+ Tree

• Search: Starting from root, examine index entries in non-leaf nodes, and traverse down the tree until a leaf node is reached
  – Non-leaf nodes can be searched using a binary or a linear search.

• Search for 5*, 15*, all data entries >=24*

Root

Height = 1
B+-tree Page Format

**Leaf Page**
- **data entries**
  - $P_0$, $R_1$, $K_1$, $R_2$, $K_2$, $\ldots$, $R_n$, $K_n$, $P_{n+1}$
  - $P_0$: Prev Page Pointer
  - $P_{n+1}$: Next Page Pointer
  - $K_1$, $K_2$, $\ldots$, $K_n$:
    - Pointer to a page with values $s.t., K_i \leq \text{Values} < K_{i+1}$
  - $R_1$, $R_2$, $\ldots$, $R_n$:
    - Pointer to record $1$, $2$, $\ldots$, $n$

**Non-leaf Page**
- **index entries**
  - $P_1$, $K_1$, $P_2$, $K_2$, $P_3$, $\ldots$, $P_m$, $K_m$, $P_{m+1}$
  - $P_1$, $P_2$, $P_3$, $\ldots$, $P_m$:
    - Pointer to a page with values $< K_1$
  - $K_1$, $K_2$, $\ldots$, $K_m$:
    - Pointer to a page with values $s.t., K_i \leq \text{Values} < K_{i+1}$
  - $P_{m+1}$:
    - Pointer to a page with values $\geq K_m$
B+ Trees in Practice

• Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133

• Typical capacities:
  - Height 4: $133^4 = 312,900,700$ records
  - Height 3: $133^3 = 2,352,637$ records

• Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes
B+-Tree: Inserting a Data Entry

• Find correct leaf $L$.
• Put data entry onto $L$.
  – 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 $L$.
• This can happen recursively
  – To split non-leaf node, redistribute entries evenly, but pushing up the middle key. (Contrast with leaf splits.)
• Splits “grow” tree; root split increases height.
  – Tree growth: gets wider or one level taller at top.
Inserting $8^*$ into B+ Tree

Entry to be inserted in parent node **Copied** up (and continues to appear in the leaf)
Inserting 8* into B+ Tree

Insert in parent node. **Pushed up** (and only appears once in the index)
Inserting 8* into B+ Tree

- Root was split: height increases by 1
- Could avoid split by re-distributing entries with a sibling
  - Sibling: immediately to left or right, and same parent
Inserting 8* into B+ Tree

- Re-distributing entries with a sibling
  - Improves page occupancy
  - Usually not used for non-leaf node splits. Why?
    - Increases I/O, especially if we check both siblings
    - Better if split propagates up the tree (rare)
    - Use only for leaf level entries as we have to set pointers
B+-Tree: Deleting a Data Entry

• Start at root, find leaf $L$ where entry belongs.
• Remove the entry.
  – If $L$ is at least half-full, done!
  – If $L$ has only $d-1$ 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 $L$.
• Merge could propagate to root, decreasing height.
DeleHng 22* and 20*

- Deleting 22* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is copied up.
... And Then Deleting 24*

- Must merge.
- In the non-leaf node, **toss** the index entry with key value = 27

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2*</td>
<td>3*</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>14*</td>
<td>16*</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>19*</td>
<td>27*</td>
<td>29*</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>33*</td>
<td>34*</td>
<td>38*</td>
<td>39*</td>
</tr>
</tbody>
</table>
```

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>13</td>
<td>17</td>
<td>30</td>
</tr>
</tbody>
</table>
```

Can this merge?

Pull down of index entry
Non-leaf Re-distribution

- Tree *during deletion* of 24*.
- Can re-distribute entry from left child of root to right child.
After Re-distribution

- Rotate through the parent node
- It suffices to re-distribute index entry with key 20; For illustration 17 also re-distributed
B+-Tree Deletion

- Try redistribution with all siblings first, then merge. Why?
  - Good chance that redistribution is possible (large fanout!)
  - Only need to propagate changes to parent node
  - Files typically grow not shrink!
Duplicates

• Duplicate Keys: many data entries with the same key value

• Solution 1:
  – All entries with a given key value reside on a single page
  – Use overflow pages!

• Solution 2:
  – Allow duplicate key values in data entries
  – Modify search
  – Use RID to get a unique (composite) key!

• Use list of rids instead of a single rid in the leaf level
  – Single data entry could still span multiple pages
A Note on Order

- **Order (d)** concept replaced by physical space criterion in practice *(at least half-full).*
  - Index (i.e. non-leaf) pages can typically hold many more entries than leaf pages.
    - Leaf pages could have actual data records
  - Variable sized records and search keys mean different nodes will contain different numbers of entries.
  - Even with fixed length fields, multiple records with the same search key value *(duplicates)* can lead to variable-sized data entries (e.g. list of rids).
ISAM - Indexed Sequential Access Method

• A static B+-tree
  – When the index is created, build a B+-tree on the relation
  – Updates and deletes don’t change the non-leaf pages.
  – Use overflow pages. Leaf pages could be empty!

• Search Cost: $\log_F N + \# \text{ overflow pages}$