Announcements

- Reading assignment
  - B+-tree tricks by Lomet
  - R-tree by Guttman
  - GiST by Hellerstein et al.
- Homework #1 due today (February 9)
- Homework #2 will be assigned Wednesday (February 11) and due in two weeks (February 26)
- No recitation session this Friday (February 14)
- Guest lecture next Monday (February 17)
  - Jennifer Widom on stream data processing
  - 4-5PM 130 North Building
  - No regular lecture on that day

Basics

- Given a value, locate the record(s) with this value
  
  \[
  \text{SELECT * FROM } R \text{ WHERE } A = \text{value};
  \]
  
  \[
  \text{SELECT * FROM } R, S \text{ WHERE } R.A = S.B;
  \]
- Other search criteria, e.g.
  - Range search
    
    \[
    \text{SELECT * FROM } R \text{ WHERE } A > \text{value};
    \]
  - Keyword search

Dense and sparse indexes

- Dense: one index entry for each search key value
  
  \[
  \text{CREATE INDEX StudentGPAIndex ON Student(GPA)};
  \]
- Sparse: one index entry for each block
  
  - Records must be clustered according to the search key
  - Dense index can directly tell if a record exists
  - Easier for sparse index

Primary and secondary indexes

- Primary index
  - Created for the primary key of a table
  - Records are usually clustered according to the primary key
  - Can be sparse
- Secondary index
  - Usually dense
- SQL
  - PRIMARY KEY declaration automatically creates a primary index,
    UNIQUE key automatically creates a secondary index
  - Secondary index can be created on non-key attribute(s)
    
    \[
    \text{CREATE INDEX StudentGPAIndex ON Student(GPA)};
    \]
**ISAM**

- What if an index is still too big?
  - Put another (sparse) index on top of that!
- ISAM (Index Sequential Access Method), more or less

Example: look up 197

```
Index blocks: 100, 108, 119, 121, 123, 129, ..., 901, 907, ..., 996, 997, ...
Data blocks: 192, 197, ..., 200, 202, ...
```

Updates with ISAM

- Overflow chains and empty data blocks degrade performance
  - Worst case: most records go into one long chain

Example: insert 107

```
Index blocks: 100, 108, 119, 121, 123, 129, ..., 901, 907, ..., 996, 997, ...
Data blocks: 100, 108, 119, 121, 123, 129, ..., ...
```

**B+-tree**

- Balanced (more or less): good performance guarantee
- Disk-based: one node per block; large fan-out

```
Max fan-out: 4
```

Sample B+-tree nodes

- Max fan-out: 4

```
Non-leaf to keys
120 ≤ k < 120
150 ≤ k ≤ 180
Leaf to keys
120 ≤ k < 150
k = 120
```

B+-tree balancing properties

- All leaves at the same lowest level
- All nodes at least half full (except root)

<table>
<thead>
<tr>
<th>Max # pointers</th>
<th>Max # keys</th>
<th>Min # active pointers</th>
<th>Min # keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-leaf</td>
<td>f</td>
<td>$\lceil f/2 \rceil$</td>
<td>$\lceil f/2 \rceil - 1$</td>
</tr>
<tr>
<td>Root</td>
<td>f</td>
<td>f − 1</td>
<td>2</td>
</tr>
<tr>
<td>Leaf</td>
<td>f</td>
<td>f − 1</td>
<td>$\lceil f/2 \rceil$</td>
</tr>
</tbody>
</table>

Lookups

- SELECT * FROM R WHERE k = 179;
- SELECT * FROM R WHERE k = 32;
Range query

```
SELECT * FROM R WHERE k > 32 AND k < 179;
```

Insertion

- Insert a record with search key value 32

Another insertion example

- Insert a record with search key value 152

Node splitting

- Yikes, this node is also already full!

More node splitting

- In the worst case, node splitting can "propagate" all the way up to the root of the tree (not illustrated here)
  - Splitting the root causes the tree to grow "up" by one level

Deletion

- Delete a record with search key value 130

- If a sibling has more than enough keys, steal one!
Stealing from a sibling

```
Remember to fix the key in the least common ancestor
```

```
Max fan-out: 4
```

Another deletion example

```
Delete a record with search key value 179
```

```
Max fan-out: 4
```

```
Cannot steal from siblings
Then coalesce (merge) with a sibling!
```

Coalescing

```
Remember to delete the appropriate key from parent
```

```
Max fan-out: 4
```

```
Deletion can "propagate" all the way up to the root of the tree (not illustrated here)
```

```
• When the root becomes empty, the tree "shrinks" by one level
```

Performance analysis

```
How many I/O's are required for each operation?
```

```
• h (more or less), where h is the height of the tree
• Plus one or two to manipulate actual records
• Plus $O(h)$ for reorganization (should be very rare if $f$ is large)
• Minus one if we cache the root in memory
```

```
How big is $h$?
```

```
• Roughly $\log_{\text{fan-out}} N$, where N is the number of records
• $B^+$-tree properties guarantee that fan-out is least $f/2$ for all non-root nodes
• Fan-out is typically large (in hundreds)—many keys and pointers can fit into one block
• A 4-level $B^+$-tree is enough for typical tables
```

B$^+$-tree in practice

```
Complex reorganization for deletion often is not implemented (e.g., Oracle, Informix)
```

```
Most commercial DBMS use $B^+$-tree instead of hashing-based indexes because $B^+$-tree handles range queries
```

The Halloween Problem

```
Story from the early days of System R…
```

```
UPDATE Payroll
SET salary = salary * 1.1
WHERE salary >= 100000;
```

```
• There is a $B^+$-tree index on Payroll(salary)
```

```
• The update never stopped (why?)
```

```
Solutions
```

```
• Scan index in reverse
```

```
• Before update, scan index to create a complete "to-do" list
```

```
• During update, maintain a "done" list
```

```
• Tag every row with transaction/statement id
```

```
``
Building a B+-tree from scratch

- Naïve approach
  - Start with an empty B+-tree
  - Process each record as a B+-tree insertion

- Problem
  - Every record requires $O(h)$ random I/O's

Bulk-loading a B+-tree

- Sort all records (or record pointers) by search key
  - Just a few passes (assuming a big enough memory)
  - More sequential I/O's
  - Now we already have all leaf nodes!

- Insert each leaf node in order
  - No need to look for the proper place to insert
  - Only the rightmost path is affected; keep it in memory

Bulk-loading diagram (not shown)

Other B+-tree tricks

- Compressing keys
  - Head compression: factor out common key prefix and store it only once within an index node
  - Tail compression: choose the shortest possible key value during a split
  - In general, any order-preserving key compression
  - Why does key compression help?

- Improving binary search within an index node
  - Cache-aware organization
  - Micro-indexing

- Using B+-tree to solve the phantom problem (later)

B+-tree versus ISAM

- ISAM is more static; B+-tree is more dynamic
- ISAM is more compact (at least initially)
  - Fewer levels and I/O's than B+-tree
- Overtime, ISAM may not be balanced
  - Cannot provide guaranteed performance as B+-tree does

B+-tree versus B-tree

- B-tree: why not store records (or record pointers) in non-leaf nodes?
  - These records can be accessed with fewer I/O's
  - Problems?
    - Storing more data in a node decreases fan-out and increases $h$
    - Records in leaves require more I/O's to access
    - Vast majority of the records live in leaves!

Coming up next

- Other tree-based indexes: R-trees and variants, GiST
- Hashing-based indexes: extensible hashing, linear hashing, etc.
- Text indexes: inverted-list index, suffix arrays