Announcements (February 3)

- Homework #1 due next Tuesday (February 8)
- No class next Thursday (February 10)
- Homework #2 will be assigned on the following Tuesday; meanwhile, use the time to think about course project!

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
    
    database indexing
    
    Search

Dense and sparse indexes

- Dense: one index entry for each search key value
- Sparse: one index entry for each block

<table>
<thead>
<tr>
<th>Name</th>
<th>ID</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bart</td>
<td>123</td>
<td>3.1</td>
</tr>
<tr>
<td>Jessica</td>
<td>142</td>
<td>2.3</td>
</tr>
<tr>
<td>Martin</td>
<td>456</td>
<td>2.1</td>
</tr>
<tr>
<td>Ralph</td>
<td>279</td>
<td>4</td>
</tr>
<tr>
<td>Lisa</td>
<td>345</td>
<td>2.3</td>
</tr>
<tr>
<td>Martin</td>
<td>456</td>
<td>2.1</td>
</tr>
<tr>
<td>Ralph</td>
<td>279</td>
<td>4</td>
</tr>
<tr>
<td>Terry</td>
<td>123</td>
<td>3.1</td>
</tr>
<tr>
<td>Sherri</td>
<td>679</td>
<td>3.3</td>
</tr>
<tr>
<td>Terri</td>
<td>697</td>
<td>3.3</td>
</tr>
<tr>
<td>Lisa</td>
<td>857</td>
<td>4.3</td>
</tr>
<tr>
<td>Windell</td>
<td>912</td>
<td>3.1</td>
</tr>
</tbody>
</table>

Dense index on name

Sparse index on $SID$

Dense versus sparse indexes

- Index size
  - Sparse index is smaller

- Requirement on records
  - Records must be clustered for sparse index

- Lookup
  - Sparse index is smaller and may fit in memory
  - Dense index can directly tell if a record exists

- Update
  - 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
  - $\text{PRIMARY KEY} \text{ declaration automatically creates a primary index, UNIQUE key automatically creates a secondary index}$
  - $\text{Secondary index can be created on non-key attribute(s)}$
  - $\text{CREATE INDEX StudentGPAIndex ON Student(GPA);}$
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

Data blocks

Updates with ISAM

Example: insert 107
Example: delete 129

B+-tree

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

Sample B+-tree nodes

Max fan-out: 4
(dicted by block size)

B+-tree balancing properties

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

<table>
<thead>
<tr>
<th></th>
<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>( f - 1 )</td>
<td>( \lfloor f/2 \rfloor )</td>
<td>( \lfloor f/2 \rfloor - 1 )</td>
</tr>
<tr>
<td>Root</td>
<td>( f )</td>
<td>( f - 1 )</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Leaf</td>
<td>( f )</td>
<td>( f - 1 )</td>
<td>( \lfloor f/2 \rfloor )</td>
<td>( \lfloor f/2 \rfloor )</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

```
Look up where the inserted key should go...
```

And insert it right there

Another insertion example

- Insert a record with search key value 152

```
Look up the key to be deleted...
```

And delete it

Oops, node is too empty!

Node splitting

```
Yikes, this node is also already full!
```

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

More node splitting

```
Oops, node is already full!
```

Deletion

- Delete a record with search key value 130

```
Look up the key to be deleted...
```

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

Oops, node is too empty!
Stealing from a sibling

Max fan-out: 4

Remember to fix the key in the lowest common ancestor

Another deletion example

Max fan-out: 4

Delete a record with search key value 179

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

Coalescing

Max fan-out: 4

Remember to delete the appropriate key from parent

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 \( O(f) \) 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/Os

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/Os
  - 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

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

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