Indexing: Part I

CPS 216
Advanced Database Systems

Announcements (February 3)

- Reading assignment for this week
  - R-tree (due Wednesday night)
  - GiST (due next Monday night, but try to read it by Thursday’s lecture)
- Homework #1 due today (midnight)
- Homework #2 will be assigned next Thursday
  - Meanwhile, use the time to think about course project!
  - No student presentation before midterm (so we can catch up with lectures)

Basics

- Given a value, locate the record(s) with this value
  
  SELECT * FROM R WHERE A = value;

  SELECT * FROM R, S WHERE R.A = S.B;

- Other search criteria, e.g.
  - Range search
    
    SELECT * FROM R WHERE A > 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

  - Records must be clustered according to the search key

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
  - 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)
  - 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

```
<table>
<thead>
<tr>
<th>Index blocks</th>
<th>Data blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>100, 108, 119, 121</td>
<td>100, 108, 119, 121</td>
</tr>
<tr>
<td>123, 129, ...</td>
<td>200, 202, ...</td>
</tr>
<tr>
<td>901, 907, ...</td>
<td>901, 907, ...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

Updates with ISAM

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

B+-tree

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

```
<table>
<thead>
<tr>
<th>Non-leaf</th>
<th>Max fan-out: 4 (dictated by block size)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100, 108, 119, 121</td>
<td>100, 108, 119, 121</td>
</tr>
<tr>
<td>123, 129, ...</td>
<td>200, 202, ...</td>
</tr>
<tr>
<td>901, 907, ...</td>
<td>901, 907, ...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

Sample B+-tree nodes

```
<table>
<thead>
<tr>
<th>Max fan-out: 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>to keys</td>
</tr>
<tr>
<td>≤ 120</td>
</tr>
<tr>
<td>120 ≤ k ≤ 150</td>
</tr>
<tr>
<td>150 ≤ k ≤ 180</td>
</tr>
<tr>
<td>180 ≤ k</td>
</tr>
</tbody>
</table>

Leaf:
```
| to next leaf node in sequence |
| to records with these k values; or, store records directly in leaves |
```

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>[f/2]</td>
<td>[f/2] – 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>[f/2]</td>
</tr>
</tbody>
</table>
```

Lookups

```
SELECT * FROM R WHERE k = 179;
SELECT * FROM R WHERE k = 32;
```

```
<table>
<thead>
<tr>
<th>Max fan-out: 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>to keys</td>
</tr>
<tr>
<td>≤ 120</td>
</tr>
<tr>
<td>120 ≤ k ≤ 150</td>
</tr>
<tr>
<td>150 ≤ k ≤ 180</td>
</tr>
<tr>
<td>180 ≤ k</td>
</tr>
</tbody>
</table>

Leaf:
```
| to next leaf node in sequence |
| to records with these k values; or, store records directly in leaves |
```
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

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

Node splitting

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

More node splitting

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

Deletion

- Delete a record with search key value 130

```
If a sibling has more than enough keys, steal one!
And delete it
Oops, node is too empty!
```
Stealing from a sibling

- Max fan-out: 4
- Remember to fix the key in the least common ancestor
- Cannot steal from siblings
- Then coalesce (merge) with a sibling!

Another deletion example

- Delete a record with search key value 179

Coalescing

- Max fan-out: 4
- Remember to delete the appropriate key from parent
- 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

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

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 \( h \) 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

The Halloween Problem

- Story from the early days of System R...

```sql
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

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 $b$
  - 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