Indexing

CPS 196.3
Introduction to Database Systems

Announcements

- Homework #3 due this Friday (November 7)
- Course project milestone 2 due next Wednesday (November 12)

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

Sparse index on SID

Dense index on name

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 a another (sparse) index on top of that!
- ISAM (Index Sequential Access Method), more or less

![ISAM Index Blocks and Data Blocks Diagram]

Updates with ISAM

- Overflow chains and empty data blocks degrade performance
  - Worst case:

![Updates with ISAM Diagram]

B+-tree

- Balanced (although not perfectly): good performance guarantee
- Disk-based: one node per block; large fan-out

![B+-tree Diagram]
Sample B+-tree nodes

Max fan-out: 4

Non-leaf

<table>
<thead>
<tr>
<th>k &lt; 120</th>
<th>120 ≤ k &lt; 150</th>
<th>150 ≤ k &lt; 180</th>
<th>180 ≤ k</th>
</tr>
</thead>
<tbody>
<tr>
<td>to keys</td>
<td>to keys</td>
<td>to keys</td>
<td>to keys</td>
</tr>
</tbody>
</table>

Leaf

<table>
<thead>
<tr>
<th>k = 20</th>
<th>k = 80</th>
<th>k = 150</th>
</tr>
</thead>
<tbody>
<tr>
<td>to records with these k values; or, store records directly in leaves</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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>[f/2]</td>
<td>[f/2] − 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>[f/2]</td>
<td>[f/2]</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;
```

Max fan-out: 4

---

Insertion

- Insert a record with search key value 32

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

Max fan-out: 4

And insert it right there

---

Another insertion example

- Insert a record with search key value 152

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

Max fan-out: 4
Node splitting

Max fan-out: 4

Yikes, this node is also already full!

More node splitting

Max fan-out: 4

- In the worst case, node splitting can "propagate" all the way up to the root of the tree (not illustrated here)
  - Splitting the root introduces a new root of fan-out 2 and causes the tree to grow "up" by one level

Deletion

- Delete a record with search key value 130

Max fan-out: 4

- Look up the key to be deleted
- If a sibling has more than enough keys, steal one!
- And delete it
- Oops, node is too empty!
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

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

- How big is $b$?
  - Roughly $\log_{fan-out} N$, where $N$ is the number of records
  - B+-tree properties guarantee that fan-out is least $b/2$ 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)
  - Leave nodes less than half full and periodically reorganize
- 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?
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?

Beyond ISAM, B-, and B^+-trees

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