Indexing

CPS 116
Introduction to Database Systems

Announcements
- Homework #3 sample solution will be available next Tuesday (Nov. 9)
- Course project milestone #2 due next Thursday

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

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

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

**Updates with ISAM**

Example: insert 107
Example: delete 129

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

**B+-tree**

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

**Sample B+-tree nodes**

- Max fan-out: 4
- Non-leaf to keys
- to next leaf node in sequence
- Leaf 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></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

```sql
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)

Deletion

- Delete a record with search key value 130
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
- Max fan-out: 4

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?
  - \( h \), the height of the tree (more or less)
  - 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 \( \log_{f_{\text{fan-out}}} N \), 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)
  - 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
- Performance analysis
  - How many I/O’s are required for each operation?
    - \( h \), the height of the tree (more or less)
    - 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 \( \log_{f_{\text{fan-out}}} N \), 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...
- 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
**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!

**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.