Indexing

CPS 116
Introduction to Database Systems

Announcements (November 6)
- Homework #3 due today!
  - Sample solution will be available next Tuesday
- Project milestone #2 due next Tuesday

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

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
  - Additional 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

Index blocks
100, 123, ..., 192
200, ...
901, ..., 996

Data blocks
100, 123, ..., 192
...
901, ..., 996

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

B+-tree

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

Sample B+-tree nodes

Non-leaf

Max fan-out: 4

to keys
100 ≤ k < 120
120 ≤ k < 150
150 ≤ k < 180
180 ≤ k

Leaf

to next leaf node in sequence

to records with these k values;
or, store records directly in leaves

Max fan-out: 4

B+-tree balancing properties

- Height constraint: all leaves at the same lowest level
- Fan-out constraint: 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>

Updates with ISAM

Example: insert 107
Example: delete 129

Sample B+-tree nodes

Max fan-out: 4

Lookups

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

Max fan-out: 4

Not found
**Range query**

```sql
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!

**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 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

- Look up the key to be deleted...

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

Max fan-out: 4

Remember to fix the key in the least common ancestor

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

Another deletion example

Max fan-out: 4

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

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 $O(f)$ 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)
  - 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?
  - 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 can be 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.
  - How about binary tree?
- Hashing-based indexes: extensible hashing, linear hashing, etc.
- Text indexes: inverted-list index, suffix arrays, etc.
- Other tricks: bitmap index, bit-sliced index, etc.