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

- Other search criteria, e.g.
  - Range search
    
    ```sql
    SELECT * FROM R WHERE A > value;
    ```
  - Keyword search
    
    ```sql
    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
- Requirement on records
- Lookup
- Update

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

Example: look up 197

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

- Non-leaf node
  - to keys $k < 120$
  - to keys $120 \leq k < 150$
  - to keys $150 \leq k < 180$
  - to keys $180 \leq k$

- Leaf node
  - to keys $k$
  - 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></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>$\lceil f/2 \rceil$</td>
<td>$\lceil f/2 \rceil - 1$</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>$\lceil f/2 \rceil$</td>
<td>$\lceil f/2 \rceil$</td>
</tr>
</tbody>
</table>

Lookups

SELECT * FROM R WHERE $k = 179$;
SELECT * FROM R WHERE $k = 32$;
Range query

\[
\text{SELECT * FROM R WHERE } k > 32 \text{ AND } k < 179; \\
\text{Max fan-out: 4}
\]

Insertion

\(\star\) Insert a record with search key value 32

\[
\text{Max fan-out: 4}
\]

Another insertion example

\(\star\) Insert a record with search key value 152

\[
\text{Max fan-out: 4}
\]
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 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.

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

Max fan-out: 4

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 \( f \) is large)
  - Minus one if we cache the root in memory

- How big is \( b \)?
  - 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 \( \text{Payroll(salary)} \)
  - The update never stopped (why?)
- Solutions?
Building a B+-tree from scratch

- Naïve approach
  - Start with an empty B+-tree
  - Process each record as a B+-tree insertion

- Problem

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
  $$\Rightarrow$$ 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
  $$\Rightarrow$$ 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 (later)
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?

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