Indexing: Part I

CPS 216
Advanced Database Systems

Announcements

- Reading assignment
  - B*-tree tricks by Lomet
  - R-tree by Guttman
  - GiST by Hellerstein et al.
- Homework #1 due today (February 9)
- Homework #2 will be assigned Wednesday (February 11) and due in two weeks (February 26)
- No recitation session this Friday (February 14)
- Guest lecture next Monday (February 17)
  - Jennifer Widom on stream data processing
  - 4-5PM 130A North
  - No regular lecture on that day

Basics

- Given a value, locate the record(s) with this value
  
  \[
  \text{SELECT} \ast \text{FROM } R \text{ WHERE } A = \text{value};
  \]
  
  \[
  \text{SELECT} \ast \text{FROM } R, S \text{ WHERE } R.A = S.B;
  \]
- Other search criteria, e.g.
  - Range search
    
    \[
    \text{SELECT} \ast \text{FROM } R \text{ WHERE } A > \text{value};
    \]
  - 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 index on name

Sparse index on S10

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

Example: look up 197

![ISAM Index Diagram]

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

![Updates with ISAM Diagram]

B⁺-tree

- Balanced (more or less): 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

\[ k < 120 \] to keys

\[ 120 \leq k < 150 \] to keys

\[ 150 \leq k < 180 \] to keys

\[ 180 \leq k \] to keys

Leaf

\[ 20 \] 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>([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

Look up 32…

And follow next-leaf pointers

Insertion

◊ Insert a record with search key value 32

Max fan-out: 4

Look up where the inserted key should go…

And insert it right there

Another insertion example

◊ Insert a record with search key value 152

Max fan-out: 4

Oops, node is already full!
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

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.

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

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
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 \( h \)?
  - Roughly \( \log f \), where \( f \) is the number of records
  - \( B^+ \)-tree properties guarantee that fan-out is at most \( f / 2 \) 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 \( Payroll(salary) \)
  - 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
  - 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 (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