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
    ```

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

<table>
<thead>
<tr>
<th>Name</th>
<th>SID</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Milhouse</td>
<td>123</td>
<td>1.1</td>
</tr>
<tr>
<td>Bart</td>
<td>142</td>
<td>2.3</td>
</tr>
<tr>
<td>Jessica</td>
<td>279</td>
<td>4</td>
</tr>
<tr>
<td>Martin</td>
<td>456</td>
<td>2.1</td>
</tr>
<tr>
<td>Nelson</td>
<td>512</td>
<td>4</td>
</tr>
<tr>
<td>Ralph</td>
<td>679</td>
<td>3.3</td>
</tr>
<tr>
<td>Sherri</td>
<td>697</td>
<td>3.3</td>
</tr>
<tr>
<td>Terri</td>
<td>857</td>
<td>4.3</td>
</tr>
<tr>
<td>Windel</td>
<td>912</td>
<td>3.1</td>
</tr>
</tbody>
</table>

Dense index on name

Sparse index on $x_1$

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 an another (sparse) index on top of that!

ISAM (Index Sequential Access Method), more or less

Example: look up 197

<table>
<thead>
<tr>
<th>Index blocks</th>
<th>Data blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>100, 101, 119, 123, 129, ...</td>
<td>100, 101, ...</td>
</tr>
<tr>
<td>192, ...</td>
<td>901, 907, ...</td>
</tr>
<tr>
<td>200, ...</td>
<td>996, 997, ...</td>
</tr>
</tbody>
</table>

Example: insert 107

Overflow block

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

Non-leaf

Max fan-out: 4

Max # pointers Max # keys Min # active pointers Min # keys
Non-leaf $f$ $f - 1$ $\lceil f/2 \rceil$ $\lceil f/2 \rceil - 1$
Root $f$ $f - 1$ 2 1
Leaf $f$ $f - 1$ $\lceil f/2 \rceil$ $\lfloor f/2 \rfloor$

B+-tree balancing properties

- All leaves at the same lowest level
- All nodes at least half full (except root)

Sample B+-tree nodes

Non-leaf

Max fan-out: 4

Max # pointers Max # keys Min # active pointers Min # keys
Non-leaf $f$ $f - 1$ $\lceil f/2 \rceil$ $\lceil f/2 \rceil - 1$
Root $f$ $f - 1$ 2 1
Leaf $f$ $f - 1$ $\lceil f/2 \rceil$ $\lfloor f/2 \rfloor$

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

- Yikes, this node is also already full!

Max fan-out: 4

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!

* 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
Stealing from a sibling

Max fan-out: 4

Remember to fix the key in the least common ancestor

Another deletion example

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

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 \( O(h) \) 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
  - \( \text{B}^+\)-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 \( \text{B}^+\)-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 \( \text{B}^+\)-tree instead of hashing-based indexes because \( \text{B}^+\)-tree handles range queries

The Halloween Problem

◊ Story from the early days of System R…

\[
\text{UPDATE Payroll} \\
\text{SET salary} = \text{salary} * 1.1 \\
\text{WHERE salary} \geq 100000; \\
\]
  - There is a \( \text{B}^+\)-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.