Indexing

CPS 116
Introduction to Database Systems

Announcements (Thu. Nov. 17)

- Project milestone #2 feedback will be emailed to by this weekend
- Homework #4 will be assigned 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

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 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

<table>
<thead>
<tr>
<th>Index blocks</th>
<th>Data blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>100, 108, 119, 121</td>
<td>192, 197, ...</td>
</tr>
<tr>
<td>123, 129, ...</td>
<td>200, ...</td>
</tr>
<tr>
<td>182, 197, ...</td>
<td>...</td>
</tr>
<tr>
<td>200, 202, ...</td>
<td>...</td>
</tr>
<tr>
<td>901, 907, ...</td>
<td>...</td>
</tr>
<tr>
<td>996, 997, ...</td>
<td>...</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Index blocks</th>
<th>Data blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>100, 108, 119, 121</td>
<td>192, 197, ...</td>
</tr>
<tr>
<td>123, 129, ...</td>
<td>200, ...</td>
</tr>
<tr>
<td>182, 197, ...</td>
<td>...</td>
</tr>
<tr>
<td>200, 202, ...</td>
<td>...</td>
</tr>
<tr>
<td>901, 907, ...</td>
<td>...</td>
</tr>
<tr>
<td>996, 997, ...</td>
<td>...</td>
</tr>
<tr>
<td>107</td>
<td>Overflow block</td>
</tr>
</tbody>
</table>

B+-tree

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

Max fan-out: 4
**Sample B+-tree nodes**

Max fan-out: 4

- Non-leaf keys:
  - 100 ≤ k < 120
  - 120 ≤ k < 150
  - 150 ≤ k < 180
  - 180 ≤ k

- Leaf keys:
  - 120 ≤ k < 150

- To records with these k values, or store records directly in leaves

---

**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>

---

**Lookups**

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

Max fan-out: 4
Range query

```
SELECT * FROM R WHERE k > 32 AND k < 179;
```

Max fan-out: 4

```
3 5 11 35 100 101 110 120 130 150 156 179 180 200
```

Look up 32…

And follow next-leaf pointers

---

Insertion

![Diagram of insertion process]

Insert a record with search key value 32

Max fan-out: 4

```
3 5 11 35 100 101 110 120 130 150 156 179 180 200
```

Look up where the inserted key should go…

And insert it right there

---

Another insertion example

![Diagram of another insertion process]

Insert a record with search key value 152

Max fan-out: 4

```
3 5 11 35 100 101 110 120 130 150 156 179 180 200
```

Oops, node is already full!
Node splitting

- Max fan-out: 4
- 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
- If a sibling has more than enough keys, steal one!
Stealing from a sibling

Remember to fix the key in the least common ancestor.

Max fan-out: 4

Another deletion example

- Delete a record with search key value 179

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

Coalescing

- 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

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 \( \log_{\text{fan-out}} N \), 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

\[ \text{B}^+\text{-tree in practice} \]

- 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

\[ \text{The Halloween Problem} \]

- 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?
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?

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.