Indexing

Introduction to Database Systems

Basics

- Given a value, locate the record(s) with this value
  \[
  \text{SELECT * FROM } R \text{ WHERE } A = \text{value};
  \]
  \[
  \text{SELECT * FROM } R, S \text{ WHERE } R.A = S.B;
  \]
- Other search criteria, e.g.
  - Range search
    \[
    \text{SELECT * FROM } R \text{ WHERE } A > \text{value};
    \]
  - Keyword 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
  - Secondary index can be created on non-key attribute(s)
    \[
    \text{CREATE INDEX StudentGPAIndex ON Student(GPA)};
    \]

ISAM

- What if an index is still too big?
  - Put a another (sparse) index on top of that!

Example: look up 197

<table>
<thead>
<tr>
<th>Index blocks</th>
<th>100, 200, 901</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data blocks</td>
<td>100, 200, 901</td>
</tr>
<tr>
<td></td>
<td>100, 200, 901</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

Overflow block

Index blocks

Data blocks

B+-tree

- Balanced (although not perfectly): good performance guarantee
- Disk-based: one node per block; large fan-out

Disk-based: one node per block; large fan-out

Max fan-out: 4

Sample B+-tree nodes

Max fan-out: 4

Non-leaf

Max fan-out: 4

Leaf

to keys

120 \leq k < 150

150 \leq k < 180

180 \leq k

Max #   Max #   Min #   Min #

pointers   keys   active pointers   keys

Non-leaf   /   / \ 1    [f/2]   [f/2] - 1

Root       /   / \ 1    2        1

Leaf       /   / \ 1    [f/2]   [f/2]

B+-tree balancing properties

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

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
Insertion

- Insert a record with search key value 32

```
100 101 110
120 130 140
150 156 179
180 200
```

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

```
100 101 110
120 130 150
152 156 179
180 200
```

Max fan-out: 4

Oops, node is already full!

Node splitting

```
100 101
120 130
150 156
179 180
200
```

Max fan-out: 4

Yikes, this node is also already full!

More node splitting

```
100 101
120 130
150 156
179 180
200
```

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

```
100 101 120
130 150 156
179 180
200
```

Max fan-out: 4

Look up the key to be deleted...
And delete it.
Oops, node is too empty!

Stealing from a sibling

```
100 101 120
130 150 156
179 180
```

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

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

Performance analysis

- How many I/O's are required for each operation?
  - $h$ (more or less), where $h$ is the height of the tree
  - Plus one or two to manipulate actual records
  - Plus $O(h)$ for reorganization (should be very rare if $h$ 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)
- Most commercial DBMS use $B^+$-tree instead of hashing-based indexes because $B^+$-tree handles range queries

$B^+$-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

- Hashing-based indexes: extensible hashing, linear hashing, etc.
- Tree-based indexes: R- and R⁺-trees, disk-based quad-trees, kdB-trees, etc.
- Other tricks: bitmap index, bit-sliced index, etc.