Indexing

CPS 196.3
Introduction to Database Systems

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

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)

```sql
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
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

B⁺-tree

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

Sample B⁺-tree nodes

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

- Leaf nodes: 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>( \lfloor f / 2 \rfloor - 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>( \lfloor f / 2 \rfloor )</td>
</tr>
</tbody>
</table>

Lookups

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

Max fan-out: 4

Range query

```sql
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

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

- 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

Stealing from a sibling

- Remember to fix the key in the least common ancestor
Another deletion example

- Delete a record with search key value 179

![Diagram: B+-tree deletion example]

Max fan-out: 4

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

Coalescing

- Max fan-out: 4

![Diagram: B+-tree coalescing example]

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 $b$?
  - 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

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?