Indexing

CPS 216
Advanced Database Systems

Outline

• The basics
• ISAM
• B+-tree
• Next time
  – R-tree
  – Inverted lists
  – Hash indexes

Indexing

• 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

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 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
  – Index can be created on non-key attribute(s)
    CREATE INDEX StudentGPAIndex ON Student(GPA);
ISAM

- What happens if you put a sparse index on top of another sparse index?
  -- Indexed Sequential Access Method (more or less)

![ISAM Index](image)

Updates with ISAM

- Overflow and empty data blocks degrade performance

![ISAM Updates](image)

B+-tree

- Balanced: good performance guarantee
- Disk-based: one node per block; large fan-out

![B+-tree Nodes](image)

B+-tree rules

- 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>ceil(f / 2)</td>
<td>ceil(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>floor(f / 2)</td>
<td>floor(f / 2)</td>
</tr>
</tbody>
</table>

Query examples

- Lookup: SELECT * FROM R WHERE k = 179;
  -- Find 179 and follow next-leaf pointers

![B+-tree Query](image)

- Range query: … WHERE k > 179 AND k < 200;

Sample B+-tree nodes

Non-leaf

Max fan-out: 4

Leaf

to keys

120 ≤ k < 150

150 ≤ k < 180

180 ≤ k

to next leaf node in sequence
to records with these k values;
or, store records directly in leaves

Query examples

- Range query: … WHERE k > 179 AND k < 200;
  -- Find 179 and follow next-leaf pointers

![B+-tree Query](image)
An insertion example (slide 1)
- Insert a record with search key value 152

Max fan-out: 4

Split when a node becomes too full

An insertion example (slide 2)
- Insert a record with search key value 152

Max fan-out: 4

Split again!

An insertion example (slide 3)
- Insert a record with search key value 152

Max fan-out: 4

Tree grows “up” when root is split (not shown in this example)

A deletion example (slide 1)
- Delete the record with search key value 130

Max fan-out: 4

Redistribute when a node becomes too empty

Right sibling has more than enough keys; steal one!

A deletion example (slide 2)
- Delete the record with search key value 130

Max fan-out: 4

Remember to fix the key in an ancestor node

Another deletion example (slide 1)
- Delete the record with search key value 179

Max fan-out: 4

Coalesce!

Cannot steal from siblings
Another deletion example (slide 2)

- Delete the record with search key value 179

Delete may propagate up; tree shrinks when root becomes empty (not shown in this example)

Max fan-out: 4

ParentDelete may propagate up; tree shrinks when root becomes empty

Performance analysis

- How many I/Os 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 (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
  - Fan-out is 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

- The index of choice in most commercial DBMS
- Complex reorganization for deletion often is not implemented (e.g., Oracle, Informix)

- Next
  - Bulk-loading
  - Concurrency control

Building a B$^+$-tree from scratch

- Naïve approach
  - Start with an empty $B^+$-tree
  - Process each record as a $B^+$-tree insertion

- Problem
  - Every record require $O(h)$ random I/Os

Bulk-loading a B$^+$-tree

- Sort all records (or record pointers) by search key
  - Just a few passes (assuming a big enough memory)
  - Can have more sequential I/Os
  - Now we already have all the 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

Concurrency control for B$^+$-trees

- Naïve approach
  - Treat nodes as data objects; use 2PL

- Problem: low concurrency
  - Every read/write starts from the root—root becomes bottleneck for locking
  - That’s the same as locking the entire table!
A simple B+-tree locking protocol

- A lookup transaction can release its lock on the parent once it gets a lock on the child
- An insert/delete transaction can do the same, provided that its modification cannot propagate up to the parent
- Never lock a node twice (even if its parent is locked all the time)

More reading in Red Book: “Efficient Locking for Concurrent Operations on B-Trees”

Remember the phantom?

T1:
```sql
INSERT INTO Student VALUES(512, "Nelson", 10, 2.1);
COMMIT;
```

T2:
```sql
SELECT * FROM Student WHERE age = 10;
SELECT * FROM Student WHERE age = 10;
COMMIT;
```

- T2 first locks all existing rows with age 10
- T1 inserts a new row with age 10
- T2 then sees the new row—phantom!

Predicate locking with B+-tree

- If there is a B+-tree on Student(age)
  - T2 will lock the B+-tree node containing age value 10
  - T1 has to wait for this lock to update the B+-tree
  - No more phantom!
- Predicate locking can be generalized to range predicates, e.g., age > 18 AND age < 20
  - Lock the B+-tree node (possibly non-leaf) containing this range

B+-tree versus ISAM

- ISAM is more static; B+-tree is more dynamic
- Performance
  - ISAM is more compact (at least initially)
    - Fewer levels and I/Os than B+-tree
  - Overtime, ISAM may not be balanced
    - Cannot provide guaranteed performance as B+-tree does
- Concurrency control
  - Much easier with ISAM
    - Because index blocks are never updated!

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/Os
- Problems
  - Storing more data in a node decreases fan-out and increases h
  - Records in leaves require more I/Os to access
  - Vast majority of the records live in leaves!