Indexing and Query Processing

Introduction to Databases
CompSci 316 Fall 2017
Announcements (Wed., Mar. 8)

• Homework #3
  • follow piazza posts for updates after each lecture

• Project
  • Comments on milestone-1 tomorrow
  • Keep working on it

• No class next week
  • spring break
Today:

• Finish B+ tree and index
• Start query processing
Index
Recall: What are indexes for?

- Given a value (*search key*), locate the record(s) with this value, or range search
  - `SELECT * FROM R WHERE A = value;`
  - `SELECT * FROM R, S WHERE R.A = S.B;`
  - `SELECT * FROM R WHERE A > value;`

- Search key ≠ key in a relation (unique attributes)
  - “Key” is highly overloaded in databases

- Recap: index structure on whiteboard
Recall: Index classification

- Dense vs. Sparse
- Clustered vs. unclustered
- Primary vs. Secondary
- Tree-based vs. Hash-based
  - we will only do tree indexes in 316
Recall: B⁺-tree

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

Max fan-out: 4
Recall: Sample B$^+$-tree nodes

Max fan-out: 4

Non-leaf

Non-leaf nodes

Max fan-out: 4

Non-leaf

Non-leaf

Non-leaf

Non-leaf

Non-leaf

Non-leaf

Non-leaf

Non-leaf

Non-leaf

Non-leaf

Non-leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf
Recall: B\textsuperscript{+}-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>$\lceil f/2 \rceil$</td>
<td>$\lceil f/2 \rceil - 1$</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>$\lfloor f/2 \rfloor$</td>
<td>$\lfloor f/2 \rfloor$</td>
</tr>
</tbody>
</table>
Lookups

- SELECT * FROM R WHERE $k = 179$;
- SELECT * FROM R WHERE $k = 32$;
Range query

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

Look up 32...

And follow next-leaf pointers until you hit upper bound

Max fan-out: 4
Insertion

• Insert a record with search key value 32

And insert it right there
Another insertion example

- Insert a record with search key value 152

Oops, node is already full!
Node splitting

Max fan-out: 4

Oops, that node becomes full!

Need to add to parent node a pointer to the newly created node
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

Max fan-out: 4

Need to add to parent node a pointer to the newly created node
Deletion

• Delete a record with search key value 130

Look up the key to be deleted...

And delete it

Oops, node is too empty!

Max fan-out: 4

If a sibling has more than enough keys, steal one!
Stealing from a sibling

Remember to fix the key in the least common ancestor of the affected nodes.
Another deletion example

• Delete a record with search key value 179

Max fan-out: 4

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.

Max fan-out: 4
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 (rare if \( f \) is large)
  • Minus one if we cache the root in memory

• How big is \( h \)?
  • Roughly \( \log_{\text{fanout}} N \), 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 (next slide)
Typical B+ Trees in Practice

• Typical max entries: 200.
  • Typical fill-factor: 67%
  • average fanout F = 133

• Typical capacities:
  • Height 4: $133^4 = 312,900,700$ records
  • Height 3: $133^3 = 2,352,637$ records

• Can often hold top levels in buffer pool:
  • Level 1 = 1 page = 8 Kbytes
  • Level 2 = 133 pages = 1 Mbyte
  • Level 3 = 17,689 pages = 133 MBytes
B\(^+\)-tree in practice

- Complex reorganization for deletion often is not implemented (e.g., Oracle)
  - 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
The Halloween Problem

• Story from the early days of System R...
  
  UPDATE Payroll
  SET salary = salary * 1.1
  WHERE salary <= 25000;
  
  • There is a B⁺-tree index on Payroll(salary)
  • The update never stopped until all employees earned 25k (why?)

• Solutions?
  
  • Scan index in reverse, or
  • Before update, scan index to create a “to-do” list, or
  • During update, maintain a “done” list, or
  • Tag every row with transaction/statement id

**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
  - Due to “skew”
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 $h$
  • Records in leaves require more I/O’s to access
  • Vast majority of the records live in leaves!
B+ tree vs. Hash-based indexes

- Extensible hashing, linear hashing, etc.
- Can only handle “=” in join or selection
  - Cannot handle range predicates >, ≥, <, ≤

Index organized file hashed on AGE, with Auxiliary index on SAL

Employee File hashed on AGE

File of <SAL, rid> pairs hashed on SAL

<table>
<thead>
<tr>
<th>AGE</th>
<th>SAL</th>
<th>Name</th>
<th>Age</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000</td>
<td></td>
<td>Smith</td>
<td>44</td>
<td>3000</td>
</tr>
<tr>
<td>3000</td>
<td></td>
<td>Jones</td>
<td>40</td>
<td>6003</td>
</tr>
<tr>
<td>5004</td>
<td></td>
<td>Tracy</td>
<td>44</td>
<td>5004</td>
</tr>
<tr>
<td>2500</td>
<td></td>
<td>Ashby</td>
<td>25</td>
<td>3000</td>
</tr>
<tr>
<td>3300</td>
<td></td>
<td>Basu</td>
<td>33</td>
<td>4003</td>
</tr>
<tr>
<td>2900</td>
<td></td>
<td>Bristow</td>
<td>29</td>
<td>2007</td>
</tr>
<tr>
<td>5004</td>
<td></td>
<td>Daniels</td>
<td>22</td>
<td>6003</td>
</tr>
</tbody>
</table>
Beyond ISAM, B-, and B⁺-trees, and hash

- Other tree-based indexes: R-trees and variants, GiST, etc.
  - How about binary tree?

- Text indexes: inverted-list index, suffix arrays, etc.
- Other tricks: bitmap index, bit-sliced index, etc.
Query Processing
Overview

• Many different ways of processing the same query
  • Scan? Sort? Hash? Use an index?
  • All have different performance characteristics and/or make different assumptions about data

• Best choice depends on the situation
  • Implement all alternatives
  • Let the query optimizer choose at run-time
Notation

- Relations: $R, S$
- Tuples: $r, s$
- Number of tuples: $|R|, |S|$
- Number of disk blocks: $B(R), B(S)$
- Number of memory blocks available: $M$
- Cost metric
  - Number of I/O’s
  - Memory requirement

- We do not count the cost of final write to disk

- Do not try to memorize the formulas for cost estimation!
  - understand the logic
  - recall the diagram of disk and memory on whiteboard
Scanning-based algorithms
Table scan

- Scan table $R$ and process the query
  - Selection over $R$
  - Projection of $R$ without duplicate elimination

- I/O’s: $B(R)$
  - Trick for selection: stop early if it is a lookup by key

- Memory requirement: 2

- Not counting the cost of writing the result out
  - Same for any algorithm!
  - Maybe not needed—results may be pipelined into another operator
Nested-loop join

\[ R \Join_p S \]

- For each block of \( R \), and for each \( r \) in the block:
  For each block of \( S \), and for each \( s \) in the block:
    Output \( rs \) if \( p \) evaluates to true over \( r \) and \( s \)
- \( R \) is called the outer table; \( S \) is called the inner table
- I/O’s: \( B(R) + |R| \cdot B(S) \)
- Memory requirement: 3

Improvement: block-based nested-loop join

- For each block of \( R \), for each block of \( S \):
  For each \( r \) in the \( R \) block, for each \( s \) in the \( S \) block: ...
- I/O’s: \( B(R) + B(R) \cdot B(S) \)
- Memory requirement: same as before

End of lecture 15