Indexing

Introduction to Databases
CompSci 316 Spring 2020
Announcements (Thu., Feb 27)

• Private project threads created on piazza
  • Please check you are on your thread
  • Primary and secondary project mentors to be assigned soon
  • They will give you feedback on MS1, check updates, and help you as needed
  • Feel free to discuss projects in all TA office hours

• Project updates to be posted **every Monday**
  • Starts Monday 03/02: each member should say what you are supposed to do for the rest of the semester and also for MS2
  • Make sure that your primary TA says “sounds good” in the response to each update, otherwise do as they suggest
  • Other team members will also check the updates and respond to the threads as needed if there are confusions or clarifications
  • Try to resolve conflicts/concerns within group whenever possible, otherwise reach out to your TAs and me early
Announcements - contd (Thu., Feb 27)

• **Homework #4 published** due next Wednesday 03/04
  • One submission per project group
  • See updates on piazza about submitting the link to your website

• **Let me know if you want to meet me about midterm or anything else**
  • Final exam will be similar in nature (problem-solving based), but the length/difficulty/question types may vary
  • Comprehensive with more focus on topics after midterm
  • How to prepare? Think about what we discuss in class and ask me tough questions!

• **Heads up: (almost) weekly quiz or lab ** every Thursday**
  • For practicing problems for the final
  • In groups, but individual submissions
  • Quizzes are shorter and discussed in class, Labs are longer with extra time after class (extra credit for submitting within the class)
Today’s lecture

• Index

• Dense vs. Sparse
• Clustered vs. unclustered
• Primary vs. secondary
• Tree-based vs. Hash-index
What are indexes for?

• 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;

• Find data by 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
  - One entry may “point” to multiple records (e.g., two users named Jessica)
- **Sparse**: one index entry for each block
  - Records must be clustered according to the search key

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Value</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>10</td>
<td>0.2</td>
</tr>
<tr>
<td>142</td>
<td>Bart</td>
<td>10</td>
<td>0.9</td>
</tr>
<tr>
<td>279</td>
<td>Jessica</td>
<td>10</td>
<td>0.9</td>
</tr>
<tr>
<td>345</td>
<td>Martin</td>
<td>8</td>
<td>2.3</td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
<td>8</td>
<td>0.3</td>
</tr>
<tr>
<td>512</td>
<td>Nelson</td>
<td>10</td>
<td>0.4</td>
</tr>
<tr>
<td>679</td>
<td>Sherri</td>
<td>10</td>
<td>0.6</td>
</tr>
<tr>
<td>697</td>
<td>Terri</td>
<td>10</td>
<td>0.6</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
<td>0.7</td>
</tr>
<tr>
<td>912</td>
<td>Windel</td>
<td>8</td>
<td>0.5</td>
</tr>
<tr>
<td>997</td>
<td>Jessica</td>
<td>8</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Sparse index on `uid`

Dense index on `name`

When are these possible?

Comparison?
Dense versus sparse indexes

- Index size
  - ??

- Requirement on records
  - ??

- Lookup
  - ??

- Update
  - ??
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 by 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):
    
    ```sql
    CREATE INDEX UserPopIndex ON User(pop);
    ```
What if the index is too big as well?

<table>
<thead>
<tr>
<th>Sparse index on uid</th>
<th>Dense index on name</th>
</tr>
</thead>
<tbody>
<tr>
<td>uid</td>
<td>name</td>
</tr>
<tr>
<td>123</td>
<td>Milhouse</td>
</tr>
<tr>
<td>142</td>
<td>Bart</td>
</tr>
<tr>
<td>279</td>
<td>Jessica</td>
</tr>
<tr>
<td>345</td>
<td>Martin</td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
</tr>
<tr>
<td>512</td>
<td>Nelson</td>
</tr>
<tr>
<td>679</td>
<td>Sherri</td>
</tr>
<tr>
<td>697</td>
<td>Terri</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
</tr>
<tr>
<td>912</td>
<td>Windel</td>
</tr>
<tr>
<td>997</td>
<td>Jessica</td>
</tr>
</tbody>
</table>
What if the index is too big as well?

Put another (sparse) index on top of that!
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

Example: look up 197
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, so lookups require scanning all data!
Start: Tuesday 03/03
Announcements (Tue., Mar 03)

• Reminder: HW4 due tomorrow (Wed 03/04)
  • One group submission per group on gradescope
  • Mention all group members’ names

• Reminder: Lab-2 (index) on Thursday in class
Binary Search Tree

Each node can hold exactly one entry.

Height balanced: All leaves are at the same level (complete binary tree).

Leaves are sorted.
Each node can hold multiple entries, has fixed max size and is sorted.

Each node does not have To be full
#pointers = #entries + 1

Height balanced

Leaves are sorted
B⁺-tree: Data only at leaves

Index Nodes Containing Index entries

Data values can be repeated as index

Leaves are linked

Data entries: Pointers to actual tuples
B⁺-tree: Closer Look

- 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

3 5 11
30

30 35
100 101 110
120 130
150 156 179
180 200

to keys

$k < 100$
to keys

$100 \leq k$
Sample B+-tree nodes

Max fan-out: 4

Non-leaf

120

150

180

to keys
100 ≤ k

to keys
120 ≤ k < 150

to keys
150 ≤ k < 180

to keys
180 ≤ k

Leaf

120

130
to next leaf node in sequence
to records with these k values;
or, store records directly in leaves (pros/cons?)

100 ≤ k < 120
120 ≤ k < 150
150 ≤ k < 180
180 ≤ k
• Question (discuss with your neighbor):

• Why do we use $B^+$-tree as database index instead of binary trees?

• Why do we use $B^+$-tree as database index instead of $B$-trees?

• What are the differences/pros/cons of $B$-trees vs. $B^+$-tree as index?
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 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 $k = 32$;

Max fan-out: 4

Not found
Search key and Data entry

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

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

And follow next-leaf pointers until you hit upper bound
Insertion

• Insert a record with search key value 32

Look up where the inserted key should go...

And insert it right there
Another insertion example

• Insert a record with search key value 152

Max fan-out: 4

Oops, node is already full!

What are our options here?
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!

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

Max fan-out: 4
Stealing from a sibling

Remember to fix the key in the least common ancestor of the affected nodes

Max fan-out: 4
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
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
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
  • A key difference between hash and tree indexes!
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?
  • 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

Clustered vs. Unclustered Index

- If order of data records in a file is the same as, or `close to’, order of data entries in an index, then clustered, otherwise unclustered

- How does it affect # of page accesses? (in class)
Clustering vs. Unclustering Index

• How does it affect # of page accesses?
• (in class – discuss with your neighbors)

• SELECT * FROM USER WHERE age = 50
  • Assume 12 users with age = 50
  • Assume one data page can hold 4 User tuples
  • Suppose searching for a data entry requires 3 IOs in a
    B+-tree, which contain pointers to the data records (assume all
    matching pointers are in the same node of B+-tree)

• What happens if the index is **unclustered**?
• What happens if the index is **clustered**?