Indexing

CS348 Spring 2023
Instructor: Sujaya Maiyy
Sections: 002 & 004 only
Announcements

• Milestone 1
  • Due today!

• Midterm Exam
  • Monday, June 26th
  • Covers Lectures 1-10 [except lecture 6]
  • One 2-sided cheat sheet allowed
Outline

• Types of indexes

• Index structure

• How to use 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;`

• We call A in the above example a search key
  • The attribute whose values will be indexed
Indexes – conceptual understanding

- Commonly asked query: SELECT * FROM User WHERE name='…'
- Index on search key Name

<table>
<thead>
<tr>
<th>Table</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Bart</td>
</tr>
<tr>
<td>142</td>
<td>Jessica</td>
</tr>
<tr>
<td>279</td>
<td>Lisa</td>
</tr>
<tr>
<td>345</td>
<td>Martin</td>
</tr>
<tr>
<td>456</td>
<td>Milhouse</td>
</tr>
<tr>
<td>512</td>
<td>Nelson</td>
</tr>
<tr>
<td>679</td>
<td>Ralph</td>
</tr>
<tr>
<td>697</td>
<td>Sherri</td>
</tr>
<tr>
<td>857</td>
<td>Terri</td>
</tr>
<tr>
<td>912</td>
<td>Windel</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Dense v.s. 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 on disk

<table>
<thead>
<tr>
<th>Sparse index on <em>uid</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>123 Milhouse 10 0.2</td>
</tr>
<tr>
<td>142 Bart 10 0.9</td>
</tr>
<tr>
<td>279 Jessica 10 0.9</td>
</tr>
<tr>
<td>345 Martin 8 2.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dense index on <em>name</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>456 Ralph 8 0.3</td>
</tr>
<tr>
<td>512 Nelson 10 0.4</td>
</tr>
<tr>
<td>679 Sherri 10 0.6</td>
</tr>
<tr>
<td>697 Terri 10 0.6</td>
</tr>
<tr>
<td>857 Lisa 8 0.7</td>
</tr>
<tr>
<td>912 Windel 8 0.5</td>
</tr>
<tr>
<td>997 Jessica 8 0.5</td>
</tr>
</tbody>
</table>
Dense v.s. 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

**Dense index on name**

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Score</th>
<th>Cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bart</td>
<td>123</td>
<td>10.0</td>
<td>0.2</td>
</tr>
<tr>
<td>Jessica</td>
<td>142</td>
<td>10.0</td>
<td>0.9</td>
</tr>
<tr>
<td>Lisa</td>
<td>279</td>
<td>10.0</td>
<td>0.9</td>
</tr>
<tr>
<td>Martin</td>
<td>345</td>
<td>8.0</td>
<td>2.3</td>
</tr>
<tr>
<td>Milhouse</td>
<td>456</td>
<td>8.0</td>
<td>0.3</td>
</tr>
<tr>
<td>Nelson</td>
<td>512</td>
<td>10.0</td>
<td>0.4</td>
</tr>
<tr>
<td>Sherri</td>
<td>679</td>
<td>10.0</td>
<td>0.6</td>
</tr>
<tr>
<td>Terri</td>
<td>697</td>
<td>10.0</td>
<td>0.6</td>
</tr>
<tr>
<td>Lisa</td>
<td>857</td>
<td>8.0</td>
<td>0.7</td>
</tr>
<tr>
<td>Windel</td>
<td>912</td>
<td>8.0</td>
<td>0.5</td>
</tr>
<tr>
<td>Jessica</td>
<td>997</td>
<td>8.0</td>
<td>0.5</td>
</tr>
</tbody>
</table>

**Sparse index on uid**

<table>
<thead>
<tr>
<th>UID</th>
<th>Name</th>
<th>Value</th>
<th>Cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8.0</td>
<td>0.7</td>
</tr>
<tr>
<td>912</td>
<td>Windel</td>
<td>8.0</td>
<td>0.5</td>
</tr>
<tr>
<td>997</td>
<td>Jessica</td>
<td>8.0</td>
<td>0.5</td>
</tr>
</tbody>
</table>

- Smaller size
- Easier to update
- Must be clustered
- Can tell directly if a record exists
- May not fit into memory
Clustering v.s. non-clustering indexes

• An index on attribute A is a **clustering** index if tuples in the relation with similar values for A are stored together in the same block.

• Other indices are **non-clustering** (or secondary) indices.

• Note: A relation may have **at most one clustering index**, and any number of non-clustering indices.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></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**

A clustering index on **uid**

Dense index on **name**

A non-clustering index on name
Primary and secondary indexes

• **Primary index**
  - Created for the *primary key* of a table
  - Records are usually clustered by the primary key
  - Clustering index $\rightarrow$ sparse

• **Secondary index**
  - Non-clustering index, usually dense (to find each search key value, since records are not clustered by this search key)

• **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):
    
    ```
    CREATE INDEX UserPopIndex ON User(pop);
    ```
Outline

• Types of indexes
  • Sparse v.s. dense
  • Clustering v.s. non-clustering
  • Primary v.s. secondary

• Index structure

• How to use index
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!
B⁺-tree

- A hierarchy of nodes with intervals
- **Balanced**: good performance guarantee
- **Disk-based**: one node per block; large fan-out

Max fan-out: 4
Sample B\(^+\)-tree nodes

Max fan-out: 4

Non-leaf

120
150
180

100 \leq k

100 \leq k < 120
120 \leq k < 150
150 \leq k < 180
180 \leq k

Leaf

120
130

to next leaf node in sequence

to records with these k values;
or, store records directly 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>$[f/2]$</td>
<td>$[f/2] - 1$</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>$[f/2]$</td>
<td>$[f/2]$</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 $k < 179$;

Max fan-out: 4

Look up 32...

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

• Insert a record with search key value 32

[Diagram of a binary search tree with nodes labeled 3, 5, 11, 30, 35, 100, 101, 110, 120, 130, 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

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

Max fan-out: 4

Look up the key to be deleted...

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

And delete it

Oops, node is too empty!
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

Max fan-out: 4
Performance analysis of B+-tree

• How many I/O’s are required for each operation?
  • \( h \), the height of the tree
  • 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
  • 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
  • $h(value) \mod f$: bucket/block to which data entry with search key value belongs
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
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 $h$ requiring more I/O on average
  • Deletions are hard since search keys cannot be repeated
  • Range queries can become less efficient
Outline

• Types of indexes:
  • Dense v.s. sparse
  • Clustering v.s. non-clustering
  • Primary v.s. secondary

• Indexing structure
  • ISAM
  • B+-tree

• How to use index
Multi-attribute indices

• Index on several attributes of the same relation.
  • `CREATE INDEX NameIndex ON User(LastName,FirstName);`

  tuples (or tuple pointers) are organized first by Lastname. Tuples with a common last name are then organized by Firstname.

• This index would be *useful* for these queries:
  • `select * from User where Lastname = 'Smith'`
  • `select * from User where Lastname = 'Smith' and Firstname='John'`

• This index would be not *useful* at all for this query:
  • `select * from User where Firstname='John'`
Index-only plan

• For example:
  • SELECT firstname, pop FROM User WHERE pop > '0.8'
  AND firstname = 'Bob';
  • non-clustering index on (firstname, pop)

• A (non-clustered) index contains all the columns needed to answer the query without having to access the tuples in the base relation.
  • Avoid one disk I/O per tuple
  • The index is much smaller than the base relation
Physical design guidelines for indices

1. Don’t index unless the performance increase outweighs the update overhead

2. Attributes mentioned in WHERE clauses are candidates for index search keys

3. Multi-attribute search keys should be considered when

4. Choose indexes that benefit as many queries as possible

5. Each relation can have at most one clustering scheme; therefore choose it wisely
   - Target important queries that would benefit the most
     - Range queries benefit the most from clustering
   - A multi-attribute index that enables an index-only plan does not benefit from being clustered
Case study

• Common queries
  1. List the name, pop of users in a particular age range
  2. List the uid, age, pop of users with a particular name
  3. List the average pop of each age
  4. List all the group info, ordered by their starting date
  5. List the average pop of a particular group given the group name

• Pick a set of clustering/non-clustering indexes for these set of queries (without worrying too much about storage and update cost)
Case study

- **Common queries**
  1. List the name, pop of users in a particular age range
  2. List the uid, age, pop of users with a particular name
  3. List the average pop of each age
  4. List all the group info, ordered by their starting date
  5. List the average pop of a particular group given the group name

- **Entities**
  - User(uid, name, age, pop)
  - Group(gid, name, date)
  - Member(uid, gid)
Case study

• Common queries
  1. List the name, pop of users in a particular age range
  2. List the uid, age, pop of users with a particular name
  3. List the average pop of each age
  4. List all the group info, ordered by their starting date
  5. List the average pop of a particular group given the group name

• User(uid, name, age, pop)
• Group(gid, name, date)
• Member(uid, gid)

A clustering index on User(age)
A non-clustering index on User(name)
A non-clustering index on User(age, pop) → index-only plan
Case study

• Common queries
  1. List the name, pop of users in a particular age range
  2. List the uid, age, pop of users with a particular name
  3. List the average pop of each age
  4. List all the group info, ordered by their starting date
  5. List the average pop of a particular group given the group name

• User(uid, name, age, pop)
• Group(gid, name, date)
• Member(uid, gid)

A clustering index on User(age)

A clustering index on Group(date)

A non-clustering index on User(name)

A non-clustering index on User(age, pop) → index-only plan

A non-clustering index on User(age)
Case study

- Common queries

  1. List the name, pop of users in a particular age range
  2. List the uid, age, pop of users with a particular name
  3. List the average pop of each age
  4. List all the group info, ordered by their starting date
  5. List the average pop of a particular group given the group name

- A non-clustering index on User(age, pop) → index-only plan

- A clustering index on User(age)

- A non-clustering index on User(name)

- A non-clustering index on User(age, pop)

- A clustering index on Group(date)

(i) Search gid by a particular name
→ Clustering/non-clustering index on Group(name)?

(ii) Search uid by a particular gid
→ Clustering/non-clustering index on Member(gid)?

(iii) Search pop by a particular uid
→ Clustering/non-clustering index on User(uid)?

- Non-clustering, as we already have a clustered index on Group(date)

If many other queries require a clustering index on Group(name), we may reconsider!
Case study

• Common queries
  1. List the name, pop of users in a particular age range.
  2. List the uid, age, pop of users with a particular name.
  3. List the average pop of each age.
  4. List all the group info, ordered by their starting date.
  5. List the average pop of a particular group given the group name.

A clustering index on User(age)
A non-clustering index on User(name)

• User(uid, name, age, pop)
• Group(gid, name, date)
• Member(uid, gid)

A non-clustering index on User(age, pop) → index-only plan

A clustering index on User(age)
A non-clustering index on User(name)

A join between User(uid, ..., pop), Member(uid,gid), Group(gid, name)

(i) Search gid by a particular name → Non-clustering index on Group(name)

(ii) Search uid by a particular gid → Clustering/non-clustering index on Member(gid)?

(iii) Search pop by a particular uid → Clustering/non-clustering index on User(uid)?

A clustering index on Group(date)

Clustering -> all records of the same gid are clustered

Or clustering index on Member(gid,uid)
Case study

• Common queries
  1. List the name, pop of users in a particular age range
  2. List the uid, age, pop of users with a particular name
  3. List the average pop of each age
  4. List all the group info, ordered by their starting date
  5. List the average pop of a particular group given the group name

  A non-clustering index on User(age, pop) \rightarrow \text{index-only plan}

  A clustering index on User(age)

  A non-clustering index on User(name)

  A join between User(uid, ..., pop), Member(uid, gid), Group(gid, name)

(i) Search gid by a particular name
   \rightarrow \text{Non-clustering index on Group(name)}

(ii) Search uid by a particular gid
    \rightarrow \text{Clustering index on Member(gid)}

(iii) Search pop by a particular uid
     \rightarrow \text{Clustering/non-clustering index on User(uid)}?

Or non-clustering index on User(uid, pop) \rightarrow \text{index-only plan, if without worrying about storage/update cost}

Non-clustering, as we already have a clustering index on User(age)
Summary

• Types of indexes:
  • Dense v.s. sparse
  • Clustering v.s. non-clustering
  • Primary v.s. secondary

• Indexing structure
  • ISAM
  • B+-tree

• How to use index
  • Use multi-attribute indices
  • Index-only plan
  • General guideline