Indexing

Introduction to Database Management
CS348 Spring 2021
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;
    ```
  • Keyword search

  database indexing
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 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

<table>
<thead>
<tr>
<th>Sparse index on uid</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>
<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>

<table>
<thead>
<tr>
<th>Dense index on name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bart</td>
</tr>
<tr>
<td>Jessica</td>
</tr>
<tr>
<td>Lisa</td>
</tr>
<tr>
<td>Martin</td>
</tr>
<tr>
<td>Milhouse</td>
</tr>
<tr>
<td>Nelson</td>
</tr>
<tr>
<td>Ralph</td>
</tr>
<tr>
<td>Sherri</td>
</tr>
<tr>
<td>Terri</td>
</tr>
<tr>
<td>Windel</td>
</tr>
</tbody>
</table>

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

• An index on attribute A of a relation 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.
Primary and secondary indexes

• **Primary index**
  • Created for the primary key of a table
  • Records are usually clustered by the primary key
  • Clustering index → 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
• 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

```
Index blocks
  100, 108, 119, 121
  123, 129, ...
  ...

100, 123, ..., 192

200, ...

901, ..., 996

Data blocks
  192, 197, ...
  200, 202, ...
  901, 907, ...
  996, 997, ...
```
Updates with ISAM

• Overflow chains and empty data blocks degrade performance
  • Worst case: most records go into one long chain, so lookups require scanning all data!

Example: insert 107
Example: delete 129
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
Sample B+-tree nodes

Max fan-out: 4

Non-leaf

120
150
180

to keys
100 \leq k

to keys
100 \leq k \leq 120

to keys
120 \leq k < 150

to keys
150 \leq k < 180

to keys
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 \( k < 179 \);

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

Look up where the inserted key should go...

And insert it right there

Max fan-out: 4
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

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

```
100
  100 101 110
  120 150
  156 180 200
```
Performance analysis of B\(^+\)-tree

• 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
  • 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
  • $h(key) \mod f$: which pointer/block to which data entry with key belongs
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
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 $h$
  • Records in leaves require more I/O’s to access
  • Vast majority of the records live in leaves!
Beyond ISAM, B-, and B⁺-trees

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

- Hashing-based indexes: extensible hashing, linear hashing, etc.
- Text indexes: inverted-list index, suffix arrays, etc.
- Other tricks: bitmap index, bit-sliced index, etc.
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 surname 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 count(*) 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
How to choose indices for many queries?

- 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
    ```
    database indexing
    ```
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
   • a WHERE clause contains several conditions; or
   • it enables index-only plans
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
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