Announcements (Thu. Oct 13)

• Project, project, & project!
  • MS-2 due today (10/13)
  • HW4 due 10/20 - group submission per project team
  • DS7 – team work for project & HW4
Recall the Disk-Main Memory diagram!
Topics

• Index

• Dense vs. Sparse
• Clustered vs. unclustered
• Primary vs. secondary
• Tree-based vs. Hash-index

Related
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

### Dense index on name

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

### Sparse index on uid

<table>
<thead>
<tr>
<th>UID</th>
<th>Name</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>10</td>
</tr>
<tr>
<td>142</td>
<td>Bart</td>
<td>10</td>
</tr>
<tr>
<td>279</td>
<td>Jessica</td>
<td>10</td>
</tr>
<tr>
<td>345</td>
<td>Martin</td>
<td>8</td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
<td>8</td>
</tr>
<tr>
<td>512</td>
<td>Nelson</td>
<td>10</td>
</tr>
<tr>
<td>679</td>
<td>Sherri</td>
<td>10</td>
</tr>
<tr>
<td>697</td>
<td>Terri</td>
<td>10</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
</tr>
<tr>
<td>912</td>
<td>Windel</td>
<td>8</td>
</tr>
<tr>
<td>997</td>
<td>Jessica</td>
<td>8</td>
</tr>
</tbody>
</table>

**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**
  • May be easier for sparse index (less movement for updates)
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):
    CREATE INDEX UserPopIndex ON User(pop);
What if the index is too big as well?

Sparse index on uid

<table>
<thead>
<tr>
<th>UID</th>
<th>Name</th>
<th>Value1</th>
<th>Value2</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>

Dense index on name

- Bart
- Jessica
- Lisa
- Martin
- Milhouse
- Nelson
- Ralph
- Sherri
- Terri
- Windel
What if the index is too big as well?

Put a another (sparse) index on top of that!
Each node can hold
Exactly one entry

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

Leaves are sorted

End of Lecture 10/6
(some earlier slides were skipped
That will be covered next weeks)
Remember Terminology

- **Index search key** (key): k
  - Used to search a record

- **Data entry**: k*
  - Pointed to by k
  - Contains record id(s)
    - another level of indirection (+) small and fixed length entries
    - or record itself
      - can be large, cannot be stored in memory, (+) saves some disk access

- **Records or data**
  - Actual tuples
  - Pointed to by record ids
B-tree: Generalizing Binary Search Trees

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

Height balanced

Leaves are sorted

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

Fill in the class

25 27 35 39 44 46

25 27 35 39 44 46
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 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>
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

to keys

\[ k < 100 \]

\[ 100 \leq k \]
Sample B⁺-tree nodes

Max fan-out: 4

Non-leaf

Leaf

to keys
100 ≤ k

to keys
100 ≤ k < 120

to keys
120 ≤ k < 150

to keys
150 ≤ k < 180

to keys
180 ≤ k

to next leaf node in sequence

to records with these k values;
or, store records directly in leaves (pros/cons?)
• Questions

• 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 (next slide)?
  • 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!
Lookups

- SELECT * FROM R WHERE $k = 179$;
- SELECT * FROM R WHERE $k = 32$;
Practice Problems

• SELECT * FROM R WHERE \( k = 179 \);
• SELECT * FROM R WHERE \( k = 32 \);

Assumptions: \( \text{Cost} = 3 \)
1. Height = 3
2. Each node = 1 block
   = 1 disk I/O = 1 cost
3. All nodes of B+tree on disk
4. At most one matching tuple for every search key
   Just find if such a tuple exist

Assumptions: \( \text{Cost} = 1 \)
1. Height = 3
2. Each node = 1 block
   = 1 disk I/O = 1 cost
3. First two levels are in memory
4. At most one matching tuple for every search key

Give me the tuple as well
Recap: 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
Practice Problem 2

- SELECT * FROM R WHERE \( k > 32 \) AND \( k < 179 \)

Cost = \( h + L - 1 \)

Assume height = \( h \)
Assume there are \( L \) matching leaves
= matching data entries reside on \( L \) leaves
Assume one node = one block
= 1 cost = 1 I/O
Everything on disk
Assume you only need to check existence

End of lecture
Thurs 10/13
Insertion

• Insert a record with search key value 32

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!

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
Deletion

• Delete a record with search key value 130

Max fan-out: 4

Look up the key to be deleted...

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

```
Max fan-out: 4
```

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
  • \( \text{B}^+ \)-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 \( \text{B}^+ \)-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!
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)
Clustered vs. Unclustered Index

• How does it affect # of page accesses?
  • Recall disk-memory diagram!

• 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 = data entries are in the same node of B+-tree)

• What happens if the index is unclustered? (cost 3+12)
• What happens if the index is clustered? (cost <= 3 + (3 +1))
  • +1 for page boundary
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 $Payroll(salary)$
  • All employees end up earning $\geq 25000$ (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

ISAM

- ISAM (Index Sequential Access Method), static version of B+-tree
- Updates are handled by (long) overflow chains
  - Overflow chains and empty data blocks degrade performance
    - Worst case: most records go into one long chain, so lookups require scanning all data!
Beyond ISAM, B-trees, and B⁺-trees

- Other tree-based indexes: R-trees and variants, GiST, etc.

- Hashing-based indexes: extensible hashing, linear hashing, etc.

- Text indexes: inverted-list index, suffix arrays, etc.

- Other tricks: bitmap index, bit-sliced index, etc.
Hash vs. Tree Index

- Hash indexes can only handle equality queries
  - \texttt{SELECT * FROM R WHERE age = 5} (requires hash index on \texttt{(age)})
  - \texttt{SELECT * FROM R, S WHERE R.A = S.A} (requires hash index on \texttt{R.A} or \texttt{S.A})
  - \texttt{SELECT * FROM R WHERE age = 5 and name = ‘Bart’} (requires hash index on \texttt{(age, name)})

- Index on prefixes: There can be “composite” hash or tree index on a set of attributes.
  - E.g., a tree-index on composite attributes \texttt{(A, B)} may have values as data entries (2, 1), (2, 2) (3, 1), (3, 5), (3, 7), (4, 1), (4, 5), ...
  - Like “lexicographic order” – when same value of \texttt{A}, sort by \texttt{B}

- (\textbf{-}) Hash index Cannot handle range queries or prefixes
  - \texttt{SELECT * FROM R WHERE age \geq 5}
  - need to use tree indexes (more common)
  - Tree index on \texttt{(age)}, or \texttt{(age, name)} works, but not \texttt{(name, age)} – why?
  - Hash index on only \texttt{(age)} works, hash index on \texttt{(age, name)} does not work

- (\textbf{+}) Hash-indexes are more amenable to parallel processing
  - Will learn more in hash-based join
  - Performance depends on how good the hash function is (whether the hash function distributes data uniformly and whether data has skew)
Trade-offs for Indexes

• Should we use as many indexes as possible?
Trade-offs for Indexes

• Should we use as many indexes as possible?

• Indexes can make
  • queries go faster
  • updates slower

• Require disk space, too
Index-Only Plans

- A number of queries can be answered without retrieving any tuples from one or more of the relations involved if a suitable index is available

```
SELECT E.dno, COUNT(*)
FROM Emp E
GROUP BY E.dno
```

```
<E.dno>
Tree index!
```

```
SELECT MIN(E.sal)
FROM Emp E
WHERE E.age=25 AND E.sal BETWEEN 3000 AND 5000
```

```
Tree index!
```

- If you have an index on E.dno in the above query, no need to access data
- For index-only strategies, clustering is not important