Topics

- Conventional Indexes
- $B+$-Tree Indexes
- (not covered: Hashing Indexes)
Indexes

• Data structures used for quickly locating tuples that meet a specific type of condition
  – *Equality* condition:
    • *find Movie tuples where Director=*Bertolucci*
  – Other conditions possible, e.g., *range* conditions:
    • *find Employee tuples where Salary*>40 AND Salary<50*

• Many types of indexes. Evaluate them on
  – *Access* time
  – *Insertion* time
  – *Deletion* time
  – Disk *Space* needed
### Basic notions

- **Primary** index
  - the index on the attribute (a.k.a. search key) that determines the sequencing of the table on disk

- **Secondary** index
  - index on any other attribute

- **Dense** index
  - every value of the indexed attribute appears in the index

- **Sparse** index
  - many values do not appear

#### A Dense Primary Index

<table>
<thead>
<tr>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>70</th>
<th>80</th>
<th>90</th>
<th>100</th>
<th>120</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td>70</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td>80</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td>90</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td></td>
<td></td>
<td></td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
<td></td>
<td>120</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sequential File
Dense and Sparse Primary Indexes

**Dense Primary Index**

<table>
<thead>
<tr>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>70</th>
<th>80</th>
<th>90</th>
<th>100</th>
<th>120</th>
</tr>
</thead>
</table>

**Sparse Primary Index**

(e.g., one pointer into each data block)

<table>
<thead>
<tr>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>80</th>
<th>100</th>
<th>140</th>
<th>160</th>
<th>200</th>
</tr>
</thead>
</table>

Find the index record with largest value that is less or equal to the value we are looking.

+ better access to overflow records
+ less index space

Q: What's the cost of checking for existence of a tuple with key value K (dense vs. sparse)?
Q: Does a sparse secondary index make sense?
Multi-Level Indexes

- Treat the index as data and build an index on it
- "Two levels are usually sufficient. More than three levels are rare." (use B-Trees instead)
- Q: Can we build a dense second level index for a dense index?
Representation of Duplicate Values in Primary Indexes

- If primary index attribute is not a primary key, duplicates may occur
- Keep one index entry for each key value, point to first instance of each value only
Insertion/deletion from Dense Index

- Insertion and deletion from dense primary index file with no duplicate values are handled in the same way as for sequential files.

Lists of available entries
Insertion in Sparse Index

- if no new block is created then do nothing
- else create an index entry with the new value
- how to find space:
  - find nearby free space and slide blocks backward, or
  - use an overflow block
Deletion from Sparse Index

- if the deleted entry does not appear in the index do nothing (to the index)
Deletion from Sparse Index (cont’d)

- if the deleted entry appears in the index replace it with the next search-key value
  - comment: we could leave the deleted value in the index assuming that no part of the system may assume it still exists without checking the block
Deletion from Sparse Index (cont’d)

- ...unless the next search key value has its own index entry. In this case delete the entry.

<table>
<thead>
<tr>
<th>Header</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>50</td>
<td>20</td>
</tr>
<tr>
<td>80</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>140</td>
<td>70</td>
</tr>
<tr>
<td>160</td>
<td>80</td>
</tr>
<tr>
<td>200</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>120</td>
</tr>
</tbody>
</table>
Secondary Indexes

- The file is not sorted according to the secondary search key
- *secondary index* has to be dense
- a *second level* index on that one would be *sparse*
**Duplicate Values and Secondary Indexes**

- store together all pointers with the same search key value

<table>
<thead>
<tr>
<th>10</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
• store together all pointers with the same search key value
• introduce a separate level of buckets
  – if many pointers for each search key value it is better to separate the pointers from the values
Find employees of the Toys dept with 4 years in the company
SELECT Name FROM Employee
WHERE Dept="Toys" AND Year=4

Advantage of Buckets: Process Queries Using Pointers Only
Buckets and Pointers Operation Used in Information Retrieval

- known as *inverted index*
- an entry in an inverted list represents occurrence of a word in an article
- lists range from 1 to 1,000,000 words
- compression also used

**Inverted Index**

<table>
<thead>
<tr>
<th>cat</th>
</tr>
</thead>
<tbody>
<tr>
<td>dog</td>
</tr>
</tbody>
</table>

**Articles**

- my **cat** is fat and hairy...
- my **cat** and dog fight all the time...
- Mary hates John’s **dog**
Summary of Indexing So Far

- Basic topics in conventional indexes
  - primary/secondary
  - sparse/dense
  - multiple levels
  - duplicate keys and buckets
  - deletion/insertion similar to sequential files
- Advantages
  - simple algorithms
  - index is sequential file
- Disadvantages
  - eventually sequentiality is lost because of overflows
B+-Tree Indexes

- Balanced (equal length paths) trees
- for minimizing disk I/O
- number of levels (logarithmic) automatically maintained w.r.t. size of the data file
- no overflow blocks (but insert, delete more complex)
- guaranteed upper limits on access, insert, delete times
Properties of B+Trees

- **Parameter $n$:** A node holds
  - $n$ search key values (sorted) and
  - $n+1$ pointers (to interior nodes or records)
- **Left key $\leq$ pointed-to value $<$ right key**
- **Choose $n$ so large that a node fits in a block**
- **Interior node:**
  - Between half and all of the $n+1$ pointers are used
- **Leaf node:**
  - Rightmost pointer to the next leaf

**Interior node for $n=2$**

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>
```

**K < 32  32 ≤ K < 40  K ≥ 40**

**Leaf node for $n=2$**

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>
```

**Record(K=32)  Record(K=40)**
Example B+Tree

- data file not sorted, then leafs have to constitute a dense index
- data file sorted, then leafs may constitute a sparse index
Lookup Algorithm

Find 24
Insertion Algorithm

- first locate the leaf page where the item should appear
- if the leaf page is not full simply include item in the page
Insertion Algorithm: Splitting Nodes

If the leaf page has $2m+1$ items after the insertion then:
- Create a new page with $m$ items.
- Insert the pointer of the new page and the first item in the parent directory.

```
<table>
<thead>
<tr>
<th>26</th>
<th>Insert 30</th>
</tr>
</thead>
<tbody>
<tr>
<td>13 22</td>
<td></td>
</tr>
<tr>
<td>25 7 8</td>
<td>13 14 15 18</td>
</tr>
</tbody>
</table>
```

Insert 29 and pointer
• splitting at one level can cause an insertion at the higher level: *recursively* apply procedure at the higher level
• when reaching the root and there is no more space, then *create a new level*
Deleting from B+ Trees

- locate the record
- delete the pointed-to record from the data file
- delete the key-pointer pair from the B+-tree ...

Delete 32

2 5 7 8
13 14 15 18
22 24
26 27 28
32 35 38
40 42 45 46
Deletion: The No-Combining Pages Case

- recall that \( n=4 \), i.e., each internal node has at least \( m=2 \) keys and \( m+1=3 \) pointers (at most 4 keys, 5 pointers)
- if the node from which was deleted is still half full (has \( m=2 \) keys):
  - DONE (lookup still works), or
  - update parent if deleted leftmost key

#### Diagram

![Diagram of a B-tree deletion process]

- Delete \( 32 \)
Deletion: The No-Combining Pages Case

- if the node from which was deleted is still half full:
  - DONE, or
  - update parent if deleted leftmost key
- otherwise (Delete 22 ??)
Deletion: Transferring Items From Siblings

- if the node N from which is deleted has minimum \((m=2)\) items:
- if there is a neighbor N' (left or right)* with \(>m\) items then
  - transfer the first (or last) item of N' to N, and
  - update the appropriate ancestors of N
- else ... (Delete 28: next page)

* transfer the last element of the left neighbor or the first of the right neighbor
Deletion: Reducing Levels

When the root is left with two children a deletion may cause removal of a level.
B+-Tree Summary

- B+-tree automatically maintain as many index levels as appropriate (no overflow blocks necessary!)
- a node (block/page) holds up to $n$ keys and $n+1$ pointers
- nodes are maintained to be between half-full and full
- range queries are supported (as for indexed sequential files)
B+Tree Indexes in Practice

- The SQL standard does not talk about indexes!
- But every real DBMS allows statements like
  
  ```sql
  CREATE INDEX IndAgeRating ON Students
  WITH STRUCTURE = BTREE,
  KEY = (age, gpa)
  ```
Multi-Key Indexing

- **Motivation**: queries of the form
  - `SELECT … FROM R WHERE cond1 and cond2`
  - `cond1` and `cond2` are equality or range conditions

- **Solution 1**: use index for only one of the conditions
  - suggested if there is a very selective condition

- **Solution 2**: pointer intersection
  - fairly selective conditions

```sql
SELECT Name FROM Employee
WHERE Dept="Toys" AND Year > 3

πName
σDept="Toys" AND Year>3
Employee
```

**Rewriting & Optimization**

```
πName
σSCAN Year>3
IND Dept="Toys"
Employee
```
Find employees of the Toys dept with >3 years in the company
SELECT Name FROM Employee
WHERE Dept="Toys" AND Year > 3
Solution 3: Multi-Key Indexing

- Appropriate when
  - each condition is not very selective
  - but their conjunction is very selective
- Brute force
- Grid structure
Common Applications of Multi-Key Indexing

- Geographic Data
  - find the city located at latitude 35, longitude 50
  - find cities in within … coordinates

- Many types of geographic index
  - R-trees: indexing of spatial objects
  - LSD trees: indexing of multidimensional points
  - k-d trees

- Similar indexing methods for multimedia queries
  - find \( k \) nearest neighbors
Grid Structure

- Space overhead (very sparse structure)
- Expensive insertion and deletion if new key values