Indexing

Juliana Freire

Some slides adapted from L. Delcambre, R. Ramakrishnan, G. Lindstrom, J. Ullman and Silberschatz, Korth and Sudarshan
Efficient Access to Data

- Data is transferred between disk and main memory in blocks
- Goal: Minimize the number of blocks read/written from disk
- Data are stored in a fixed structure
- A fixed structure is unlikely to be the best for all possible access patterns
  - Good for:
    List all accounts in the Downtown branch
  - What about:
    List all accounts with balance = 350

<table>
<thead>
<tr>
<th>Account</th>
<th>Branch</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-217</td>
<td>Brighton</td>
<td>750</td>
</tr>
<tr>
<td>A-101</td>
<td>Downtown</td>
<td>500</td>
</tr>
<tr>
<td>A-110</td>
<td>Downtown</td>
<td>600</td>
</tr>
<tr>
<td>A-215</td>
<td>Mianus</td>
<td>700</td>
</tr>
<tr>
<td>A-102</td>
<td>Perryridge</td>
<td>400</td>
</tr>
<tr>
<td>A-201</td>
<td>Perryridge</td>
<td>900</td>
</tr>
<tr>
<td>A-218</td>
<td>Perryridge</td>
<td>700</td>
</tr>
<tr>
<td>A-222</td>
<td>Redwood</td>
<td>700</td>
</tr>
<tr>
<td>A-305</td>
<td>Round Hill</td>
<td>350</td>
</tr>
</tbody>
</table>
Indexes: Motivation

Q1: List all accounts with balance = 350
Requires all tuples to be examined – very inefficient if table is large

• An index on balance makes it efficient to find tuples with a specific balance
  – Only accounts with balance = 350 are examined
  – Fewer blocks retrieved from disk!
Indexes and SQL

• Not part of the standard up to (and including) SQL99
• Most commercial systems allow the creation of indexes

CREATE INDEX balanceIndex on Account(balance);
DROP INDEX balanceIndex;
Using Indexes

• Given a value \( v \), the index takes us to only those tuples that have \( v \) in the attribute(s) of the index.

• Example:

CREATE INDEX BookInd ON Books(author);
CREATE INDEX SellInd ON Sells(store, bookId);

• Use BookInd and SellInd to find the prices of books authored by Ullman and sold by Joe. (next slide)
Using Indexes --- (2)

SELECT price FROM Books, Sells
WHERE author= 'Ullman' AND
    Books.id= Sells.bookId AND
    store= 'Joe''s Bookstore';

1. Use BookInd to get all the books written by Ullman.

2. Then use SellInd to get prices of those books, with bar = 'Joe’’s Bar’
Database Tuning: Index Selection

• Selecting the best indexes for a database is a hard problem

• Tradeoffs
  – ++Index on an attribute speeds up queries that mention that attribute (including joins)
  – -- Indexes make insertions, deletions and updates more complex and time consuming
  – -- Indexes use up space – an extra table

• Rule of thumb
  – If R is queried more often than updated, create indexes on attributes most frequently specified in queries
  – If R is updated often, be careful!
Example: Tuning

- Suppose the only things we did with our Books database was:
  1. Insert new facts into a relation (10%).
  2. Find the price of a given book at a given store (90%).
- Then SellInd on Sells(store,bookId) would be wonderful, but BookInd on Books(author) would be harmful.
Tuning Advisors

• A major research thrust.
  – Because hand tuning is so hard.

• An advisor gets a *query load*, e.g.:
  1. Choose random queries from the history of queries run on the database, or
  2. Designer provides a sample workload.
Tuning Advisors --- (2)

• The advisor generates candidate indexes and evaluates each on the workload.
  – Feed each sample query to the query optimizer, which assumes only this one index is available.
  – Measure the improvement/degradation in the average running time of the queries.
Index Selection: Example

- StarsIn is stored in 10 disk blocks – cost of examining entire relation = 10
- On avg, a star has appeared in 3 movies and a movie has 3 stars
- Tuples for a given star or movie are likely to be spread over the 10 disk blocks – it takes 3 disk accesses to find the (avg of) 3 tuples for a star or movie
- 1 block access required to read index. If index is modified, 2 block accesses are needed
- Insertion requires 2 block accesses

StarsIn(title, year, starName)
Q1: SELECT title, year
   FROM StarsIn
   WHERE starName = s;
Q2: SELECT starName
   FROM StarsIn
   WHERE title = t AND year = y;
I: INSERT INTO StarsIn
   VALUES(t,y,s)
Now, fill out the table below with the costs for each scenario.

<table>
<thead>
<tr>
<th>Action</th>
<th>No idx</th>
<th>Star idx</th>
<th>Movie idx</th>
<th>Star+Movie idx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Index Selection: Example (cont.)

StarsIn(title, year, starName)
Q1: SELECT title, year
   FROM StarsIn
   WHERE starName = s;
Q2: SELECT starName
   FROM StarsIn
   WHERE title = t AND year = y;
I: INSERT INTO StarsIn VALUES(t,y,s)

Now, fill out the table below:

<table>
<thead>
<tr>
<th>Action</th>
<th>No idx</th>
<th>Star idx</th>
<th>Movie idx</th>
<th>Star+Movie idx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>10</td>
<td>4</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>Q2</td>
<td>10</td>
<td>10</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>I</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>
Index Selection: Estimating Avg Cost

StarsIn(title, year, starName)
Q1: SELECT title, year
   FROM StarsIn
   WHERE starName = s;
Q2: SELECT starName
   FROM StarsIn
   WHERE title = t AND year = y;
I: INSERT INTO StarsIn VALUES(t,y,s)

What is the best index configuration?
P1 = fraction of time we do Q1
P2 = fraction of time we do Q2
I = fraction of time we do I = 1 – P1 – P2

<table>
<thead>
<tr>
<th>Action</th>
<th>No idx</th>
<th>Star idx</th>
<th>Movie idx</th>
<th>Star+Movie idx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>10</td>
<td>4</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>Q2</td>
<td>10</td>
<td>10</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>I</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>cost</td>
<td>2+8p1+8p2</td>
<td>4+6p2</td>
<td>4+6p1</td>
<td>6-2p1-2p2</td>
</tr>
</tbody>
</table>
Selecting the Indexes

What is the best configuration if

1. $P1 = P2 = 0.1$
2. $P1 = P2 = 0.4$
3. $P1 = 0.5$ and $P2 = 0.1$

<table>
<thead>
<tr>
<th>Action</th>
<th>No idx</th>
<th>Star idx</th>
<th>Movie idx</th>
<th>Star+Movie idx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>10</td>
<td>4</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>Q2</td>
<td>10</td>
<td>10</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>I</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

| cost   | $2 + 8p1 + 8p2$ | $4 + 6p2$ | $4 + 6p1$ | $6 - 2p1 - 2p2$ |

- Mostly ins, few queries, no index
- Mostly Q1, best to index starName
- Many queries, few ins $\rightarrow$ both indexes

Q1: SELECT title, year
FROM StarsIn
WHERE starName = s;

Q2: SELECT starName
FROM StarsIn
WHERE title = t AND year = y;

I: INSERT INTO StarsIn VALUES(t, y, s)
Index: Examples and Concepts

• Some examples:
  – Internet directories, e.g., yahoo, google, dmoz
  – Search engines, e.g., google, altavista

• And many other applications, including databases!

• Basic concepts:
  – Search key: attribute to set of attributes used to look up records in a file.
  – An index file consists of records (called index entries) of the form (search key, pointer)

• Index speeds up selections on the search key field(s)
Index

• Any data structure that takes as input a property of records and quickly finds records with that property
  – Simple indexes on sorted files
  – Secondary indexes on unsorted files
  – B-trees
  – Hash tables
  – Bitmaps
Index Evaluation Metrics

- Access types supported efficiently -- which queries will benefit from the index
  - records with a specified value in the attribute
  - or records with an attribute value falling in a specified range of values.

- Access time
- Insertion time
- Deletion time
- Space overhead
Index: Some notes

- Indexes usually help for queries where an attribute is compared against a constant, e.g., A=3; A<=3
- Indexes can greatly speed up queries, both for selections and joins
- Every index makes insertions, deletions and updates more costly
- Index selection is one of the hardest part of database design
  - Need to estimate query mix and db operations
  - Tradeoff between query speed-up and update cost
- If modifications are the predominant action, you should be very conservative about creating indexes
Index: Some more notes

• Indexes are often (automatically) created to enforce key constraints
• Indexes can speed up constraint checking!
• When inserting or updating new tuple, check in index if there is already a tuple with the unique value
  – This is much faster than scanning the whole relation!
• Some DB systems provide index advisors
Tips on Using Indexes

Bonus Material
Indexes on Sequential Files

- File is sorted on the search key of the index
  - search key is usually but not necessarily the primary key.
- AKA *Primary/Clustered* Index

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ashby</td>
<td>25</td>
<td>3000</td>
</tr>
<tr>
<td>Basu</td>
<td>33</td>
<td>4003</td>
</tr>
<tr>
<td>Bristow</td>
<td>30</td>
<td>2007</td>
</tr>
<tr>
<td>Basu</td>
<td>33</td>
<td>4003</td>
</tr>
<tr>
<td>Cass</td>
<td>50</td>
<td>5004</td>
</tr>
<tr>
<td>Daniels</td>
<td>22</td>
<td>6003</td>
</tr>
<tr>
<td>Jones</td>
<td>40</td>
<td>6003</td>
</tr>
<tr>
<td>Smith</td>
<td>44</td>
<td>3000</td>
</tr>
<tr>
<td>Tracy</td>
<td>44</td>
<td>5004</td>
</tr>
</tbody>
</table>

Search key is “Name”

Records are sorted by “Name” in the file
Example: Search key is not the primary key

<table>
<thead>
<tr>
<th>Search key is &quot;Branch name&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brighton</td>
</tr>
<tr>
<td>Mianus</td>
</tr>
<tr>
<td>Redwood</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A-217</th>
<th>Brighton</th>
<th>750</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-101</td>
<td>Downtown</td>
<td>500</td>
</tr>
<tr>
<td>A-110</td>
<td>Downtown</td>
<td>600</td>
</tr>
<tr>
<td>A-215</td>
<td>Mianus</td>
<td>700</td>
</tr>
<tr>
<td>A-102</td>
<td>Perryridge</td>
<td>400</td>
</tr>
<tr>
<td>A-201</td>
<td>Perryridge</td>
<td>900</td>
</tr>
<tr>
<td>A-218</td>
<td>Perryridge</td>
<td>700</td>
</tr>
<tr>
<td>A-222</td>
<td>Redwood</td>
<td>700</td>
</tr>
<tr>
<td>A-305</td>
<td>Round Hill</td>
<td>350</td>
</tr>
</tbody>
</table>

Records are sorted by "Branch name" in the file
Dense vs. Sparse Indexes

• Dense: Index record appears for every search-key value in the file
  – One per record in sequential file

• Index records for only some search-key values
  – *Applicable when records are sequentially ordered on search-key*
To find tuple with Name = Daniels
   – Search index blocks for Daniels
   – Follow associated pointer
Since the search keys are in the same sorted order as the file, what are the benefits of using a dense index?
Dense Indexes: Advantages

There are several factors that make dense indexes more efficient than it seems:

– Number of index blocks usually small compared with the number of data blocks – if index is too large, use sparse index instead

– Since keys are sorted, binary search can be used

– Index may fit in memory
  – Queries asking only for search key can be evaluated in memory
  – Queries asking for other attributes require only 1 disk I/O
Dense Indexes: Down to the Numbers

- R has 1,000,000 tuples
- 10 tuples per 4096 byte block →
  
  \[4096 \times 100,000\] blocks = **400MB**

- Key field: 30 bytes; Pointer: 8 bytes
- How big is a dense index for R?
  
  \[1,000,000 \times 38\] bytes = **40MB** = **10,000** blocks

- How many block accesses are required to find a search key?
  
  \[\log_2(10000) \approx 13\] – need 13-14 block accesses
### Dense Index: Another Example

<table>
<thead>
<tr>
<th>Location</th>
<th>Code</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brighton</td>
<td>A-217</td>
<td>750</td>
</tr>
<tr>
<td>Downtown</td>
<td>A-101</td>
<td>500</td>
</tr>
<tr>
<td>Mianus</td>
<td>A-110</td>
<td>600</td>
</tr>
<tr>
<td>Perryridge</td>
<td>A-215</td>
<td>700</td>
</tr>
<tr>
<td>Redwood</td>
<td>A-102</td>
<td>400</td>
</tr>
<tr>
<td>Round Hill</td>
<td>A-201</td>
<td>900</td>
</tr>
<tr>
<td></td>
<td>A-218</td>
<td>700</td>
</tr>
<tr>
<td></td>
<td>A-222</td>
<td>700</td>
</tr>
<tr>
<td></td>
<td>A-305</td>
<td>350</td>
</tr>
</tbody>
</table>
Example: Sparse Index

Search key is “Name”

Records are sorted by “Name” in the file

If dense index is too large use sparse index instead

– One search key per data block

How to locate a record with search-key value \( K \)?
Sparse Index: Locating a Record

- Search key is “Name”
- Records are sorted by “Name” in the file

\[ K = \text{Jones} \]

Find largest search key \( \leq K \)

Use binary search within that block
Sparse Indexes: Down to the Numbers

• R has 1,000,000 tuples
• 10 tuples per 4096 byte block – 100,000 data blocks
• 100 key-pointer pairs in one index block
• How big is a sparse index for R?
  100,000/100 = 1,000 blocks = 4MB

More likely to fit in memory
Example of Sparse Index Files

<table>
<thead>
<tr>
<th>Building</th>
<th>Code</th>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brighton</td>
<td>A-217</td>
<td>Brighton</td>
<td>750</td>
</tr>
<tr>
<td></td>
<td>A-101</td>
<td>Downtown</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td>A-110</td>
<td>Downtown</td>
<td>600</td>
</tr>
<tr>
<td></td>
<td>A-215</td>
<td>Mianus</td>
<td>700</td>
</tr>
<tr>
<td></td>
<td>A-102</td>
<td>Perryridge</td>
<td>400</td>
</tr>
<tr>
<td></td>
<td>A-201</td>
<td>Perryridge</td>
<td>900</td>
</tr>
<tr>
<td></td>
<td>A-218</td>
<td>Perryridge</td>
<td>700</td>
</tr>
<tr>
<td></td>
<td>A-222</td>
<td>Redwood</td>
<td>700</td>
</tr>
<tr>
<td></td>
<td>A-305</td>
<td>Round Hill</td>
<td>350</td>
</tr>
</tbody>
</table>
Challenge Exercise

• Suppose blocks hold either 3 data records, or 10 key-pointer pairs. As a function of the number of records \( n \), how many blocks do we need to hold

1) The data file \( \frac{n}{3} \)

and

2) A dense index? \( \frac{n}{10} \) \( \text{Total} = \frac{n}{10} + \frac{n}{3} = \frac{13n}{30} \)

3) A sparse index? \( \frac{n}{30} \) \( \text{Total} = \frac{n}{30} + \frac{n}{3} = \frac{11n}{30} \)
Challenge Exercise

• Suppose blocks hold either 3 data records, or 10 key-pointer pairs. As a function of the number of records $n$, how many blocks do we need to hold

1) The data file $n/3$

and

2) A dense index? $n/10$ Total = $n/10 + n/3 = 13n/30$

Should contain one index record per data record, a total of $n$ index records. Since we can fit 10 of those in a block, we need a total of $n/10$ blocks

3) A sparse index? $n/30$ Total = $n/30 + n/3 = 11n/30$

Should contain one index record per data block, a total of $n/3$ index records. Since we can fit 10 of those in a block, we need a total of $(n/3)/10$ blocks
Multilevel Index

- Index can cover many blocks – expensive to locate search key even using binary search
- If primary index does not fit in memory, access becomes expensive.
- To reduce number of disk accesses, put an index on the index
  - treat primary index on disk as a sequential file (inner index)
  - construct a sparse index on primary index (outer index)
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.
Multilevel Index (Cont.)
A Multi-Level Index

Looking on the Web for pages about Digital Art

Web = billions of pages

10s of thousands of pages

The web organized by topic into categories.
You can go directly to “Digital” -- no need to scan all the 10s of thousands pages
Multi-Level Indexes: Down to the Numbers

• R has 1,000,000 tuples
• 10 tuples per 4096 byte block – 100,000 data blocks
• 100 key-pointer pairs in one index block
• First-level index: 100,000/100 = 1,000 blocks
• How big is a second-level index for R?
  – 1,000/100 = 10 blocks  Surely fits in memory!
• 2 disk I/O per lookup
  – 1\textsuperscript{st} access to memory
  – 1 I/O to access first-level index
  – 1 I/O to access data block
Managing Indexes During Data Modifications

• So far we considered indexes as a \textit{packed} sequence of blocks
• As data is modified, index records are inserted, deleted and updated
• Alternatives similar to \textit{file organization} for sequential files:
  – Create overflow blocks – extensions of the primary block without entry in sparse index
  – Insert new block in the sequential order – need entry in sparse index
  – If there is no space, slide records to adjacent blocks. If adjacent blocks are too empty, they can be combined
## Index Updates: Summary

<table>
<thead>
<tr>
<th>Action</th>
<th>Dense Index</th>
<th>Sparse Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create empty overflow block</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Delete empty overflow block</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Create empty sequential block</td>
<td>None</td>
<td>Insert</td>
</tr>
<tr>
<td>Delete empty sequential block</td>
<td>None</td>
<td>delete</td>
</tr>
<tr>
<td>Insert record</td>
<td>Insert</td>
<td>Update(?)</td>
</tr>
<tr>
<td>Delete record</td>
<td>Delete</td>
<td>Update(?)</td>
</tr>
<tr>
<td>Slide record</td>
<td>Update</td>
<td>Update(?)</td>
</tr>
</tbody>
</table>

- **Dense Index**:
  - No affect on dense indexes – they point to records.

- **Sparse Index**:
  - No affect on sparse indexes – they point to primary blocks.
  - Need to create/delete entry for new block.
  - Typically no effect, except: adding 1st record in block; delete last record in block or record with search key.
Index Update: Deletion

• If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.

• Single-level index deletion:
  – Dense indices – deletion of search-key is similar to file record deletion.
  – Sparse indices – if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.
Index Update: Insertion

• Single-level index insertion:
  – Perform a lookup using the search-key value appearing in the record to be inserted.
  – Dense indices – if the search-key value does not appear in the index, insert it.
  – Sparse indices – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created. In this case, the first search-key value appearing in the new block is inserted into the index.

• Multilevel insertion (as well as deletion) algorithms are simple extensions of the single-level algorithms
Sparse vs. Dense Index Files

• Sparse uses less space and incurs less maintenance overhead for insertions and deletions.
• Generally slower than dense index for locating records.
• Good tradeoff: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.
Secondary/Unclustered Indexes

• Useful to have multiple indexes for a table -- can only have one order for data blocks
• Secondary indexes serve the same purpose as primary indexes, but search key specifies an order different from the sequential order of the file

Search key is “Age”

<table>
<thead>
<tr>
<th>Age</th>
<th>Records</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Ashby, 25, 3000</td>
</tr>
<tr>
<td>25</td>
<td>Basu, 44, 4003</td>
</tr>
<tr>
<td>30</td>
<td>Bristow, 30, 2007</td>
</tr>
<tr>
<td>33</td>
<td>Cass, 50, 5004</td>
</tr>
<tr>
<td>40</td>
<td>Daniels, 22, 6003</td>
</tr>
<tr>
<td>44</td>
<td>Jones, 40, 6003</td>
</tr>
<tr>
<td>44</td>
<td>Smith, 44, 3000</td>
</tr>
<tr>
<td>50</td>
<td>Tracy, 33, 5004</td>
</tr>
</tbody>
</table>

Records are sorted by “Name” in the file
More on Secondary Indexes

• Secondary indexes are always dense
• It makes no sense to talk of a sparse secondary index
  – Can’t use it to predict the location of any record not mentioned in the index – can’t find the record without scanning the whole file!

<table>
<thead>
<tr>
<th></th>
<th>sparse</th>
<th>dense</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>secondary</td>
<td>NO!</td>
<td>YES</td>
</tr>
</tbody>
</table>
Secondary Index on **balance** field of **account**

- Secondary indices may have duplicate search keys – several accounts with the same balance

- To avoid space wastage, create another level of indirection – do not repeat the search key value
Secondary Index on *balance* field of *account*

Does this extra level really help?

It helps only if:

- search key values are larger than pointers
- the average key appears at least twice

- Secondary indices may have duplicate search keys – several accounts with the same balance

- To avoid space wastage, create another level of indirection – do not repeat the search key value
Primary and Secondary Indexes

• *Secondary indices have to be dense.*
• When a file is modified, every index on the file must be updated: updating indices imposes overhead on database modification.
• Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  – each record access may fetch a new block from disk
Answering Queries using the Index

- When queries have multiple conditions, and each condition has a secondary index, we can find the bucket pointers by *intersecting the sets of pointers* (in memory) and *retrieve only records pointed to by the intersection*
Example

Select title
From movie
Where studioName='Disney' and year=1995
Composite Search Keys: Four dense, secondary indices on a table

Index based on <age, sal>

Index based on <sal, age>

Index based on <sal>

Index based on <age>

Data records sorted by name
Using Composite Search Keys

Which index can you use for each of these queries?

• age = 20
• age = 20 and sal = 20
• age=20 and sal > 10
• age > 20 and sal > 30
B⁺-Tree Index Files

B⁺-tree indices are an alternative to indexed-sequential files.

- **Disadvantage of indexed-sequential files:** performance degrades as file grows
  - Many overflow blocks are created—periodic reorganization of entire file is required.

- **Advantage of B⁺-tree index files:**
  - Automatically reorganizes itself with small, local, changes, in the face of insertions and deletions—no need for overflow blocks!
  - Reorganization of entire file is not required to maintain performance.
  - Supports equality and range-searches efficiently

- **Disadvantage of B⁺-trees:**
  - Extra insertion and deletion overhead, space overhead.

- **Advantages of B⁺-trees outweigh disadvantages, and they are used extensively.**
A B\(^+\)-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length – a balanced tree (remember from Algorithms!)
- Minimum 50% occupancy (except for root)
  - Leaf: \(\lceil (n-1)/2 \rceil \leq \text{occupancy} \leq n-1\)
  - Non-leaf: \(\lceil n/2 \rceil \leq \text{occupancy} \leq n\)
- \(n\) is fixed for a given tree
Example B+ Tree

- \( n=5 \)
- Search-keys in a node are ordered
- Pointers to nodes or records

<table>
<thead>
<tr>
<th>Id</th>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>987</td>
<td>John Doe</td>
<td>1</td>
</tr>
<tr>
<td>123</td>
<td>Jane Doe</td>
<td>2</td>
</tr>
<tr>
<td>444</td>
<td>Mary Mary</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Search over B+ Tree

- Search begins at root, and key comparisons direct it to a leaf
- Search for 5*, 15*, all data entries $\geq 24^*$ ...

Based on the search for 15*, we know it is not in the tree!
B+ Trees in Practice

- Typical n: 198. Typical fill-factor: 67%.
  - average fanout = 198*.67 = 133

- Typical capacities:
  - Height 4: $133^4 = 312,900,700$ records
  - Height 3: $133^3 = 2,352,637$ records

- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes
Observations about B⁺-trees

• Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close.

• The non-leaf levels of the B⁺-tree form a hierarchy of sparse indices.

• The B⁺-tree contains a relatively small number of levels (logarithmic in the size of the main file), thus searches can be conducted efficiently.

• *Insertions and deletions to the main file can be handled efficiently*, as the index can be restructured in logarithmic time (see textbook for details).
B⁺-Tree File Organization

- Index file degradation problem is solved by using B⁺-Tree indices.
- Data file degradation problem is solved by using B⁺-Tree File Organization.
- The leaf nodes in a B⁺-tree file organization store records, instead of pointers.
- Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Leaf nodes are still required to be half full.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B⁺-tree index.
B⁺-Tree File Organization: Example
Document Retrieval and Inverted Indexes

• How does Google work?
• Crawler goes around the Web and retrieves all documents it can find
• Retrieved docs are parsed and its words extracted
  – d1: w1,w2,w3
  – d2: w1,w2,w4,w5
• Index is inverted:

<table>
<thead>
<tr>
<th>Word/Doc</th>
<th>d1</th>
<th>d2</th>
<th>d3</th>
<th>...</th>
<th>dn</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>W2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>W3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>W4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>W5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

How are queries evaluated?
Document Retrieval and Inverted Indexes

- **Index:** word is the search key
- **Bucket contains pointers to all documents where word can be found**
- **Documents**
Index Definition in SQL

• Create an index

  \textbf{create index} \texttt{<index-name>} \textbf{on} \texttt{<relation-name>} \hfill (\texttt{<attribute-list>})

  E.g.: \textbf{create index} \texttt{b-index on branch(branch-name)}

• Use \textbf{create unique index} to indirectly specify and enforce the condition that the search key is a candidate key is a candidate key.
  – Not really required if SQL \texttt{unique} integrity constraint is supported

• To drop an index

  \textbf{drop index} \texttt{<index-name>}