Indexing

Cost Model for Data Access

- Data needs to be accessed fast
- How should we estimate the costs for accessing data?
  - number of I/Os: number of pages that need to be read from disk
- Why I/O and not other costs?
  - Real systems consider also CPU but I/O has definitely more weight
- Simplifications
  - only consider disk reads (ignore writes)
  - only consider number of I/Os and not the individual time for each read (ignores page pre-fetch)
  - Average-case analysis; based on several simplistic assumptions.

☛ Good enough to show the overall trends!

Typical Operations

- Scan over all records
  - `SELECT * FROM Skaters`
- Equality Search
  - `SELECT * FROM Skaters WHERE sid = 100`
- Range Search
  - `SELECT * FROM Skaters WHERE age > 5 and age <= 10`
- Insert
  - `INSERT INTO skaters VALUES (23, 'lilly', 10, 8)`
- Delete
  - `DELETE FROM Skaters WHERE sid = 100`
  - `DELETE FROM Skaters WHERE age > 30 AND age <= 50`
- Update
  - Delete+insert

File Organization

assume each relation is a file:

- <u>Heap files:</u>
  - Linked, unordered list of all pages of the file
  - Is it good for:
    - scan retrieving all records (SELECT *)?
      - yes, you have to retrieve all pages anyways
    - equality search on primary key
      - not great: have to read on avg. half the pages to return one record
    - range search or equality search on non-primary key
      - not great: have to read all pages to return subset of records.
    - insert
      - yes: Cost for insert low (insert anywhere)
    - delete/update
      - same as for equality/range search -- depends on WHERE clause

File Organizations II

- <u>Sorted Files:</u>
  - Records are ordered according to one or more attributes of the relation
  - Is it good for:
    - scan retrieving all records (SELECT *)?
      - yes, you have to retrieve all pages anyways
    - equality search on sort attribute
      - good: find first qualifying page with binary search in log2(number-of-pages)
    - range search on sort attribute
      - good: find first qualifying page with binary search in log2(number-of-pages); adjacent pages might have additional matching records
    - insert
      - not good: have to find proper page; overflow possible
    - delete/update
      - finding tuple same as equality/range search depending on WHERE clause
      - update itself might lead to restructuring of pages
    - Sorted output: (ORDER BY)
      - good if on sorted attribute

Indexes

- Even a sorted file only support queries on sorted attributes.
- Solution: Build an index for any attribute (collection of attributes) that is frequently used in queries
  - Additional information that helps finding specific tuples faster
  - We call the collection of attributes over which the index is built the **search key attributes** for the index.
  - Any subset of the attributes of a relation can be the search key for an index on the relation.
  - *Search key* is not the same as *primary key / key candidate*

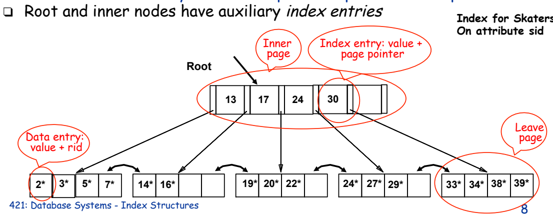
Creating an index in DB2

- **Simple**
  - ☆ **`CREATE INDEX ind1 ON Skaters(sid);`**
  - ☆ **`DROP INDEX ind1;`**

B+ Tree: The Most Widely Used Index

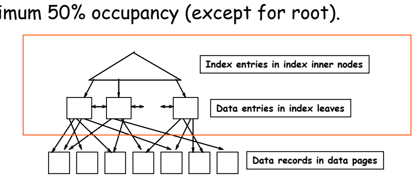
- Each node/leaf represents one page
  - ☆ Since the page is the transfer unit to disk
- Leafs contain *data entries* (denoted as k*)
  - ☆ For now, assume each data entry *represents* one tuple. The data entry consists of two parts
    - Value of the search key (k)
    - Record identifier (rid = (page-id, slot))
  - ☆ That is: data entry is NOT a tuple but a pointer to a tuple
- Root and inner nodes have auxiliary *index entries*

Index for Skaters On attribute sid



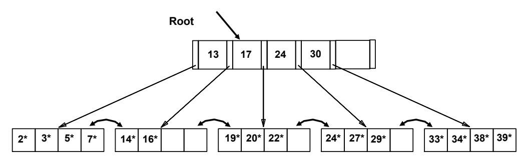
B+ Tree (contd.)

- *height-balanced.*
  - ☆ Each path from root to tree has the same height
- F = fanout = number of children for each node (~ number of index entries stored in node)
- N = # leaf pages
- Insert/delete at $\log_F N$ cost;
- Minimum 50% occupancy (except for root).



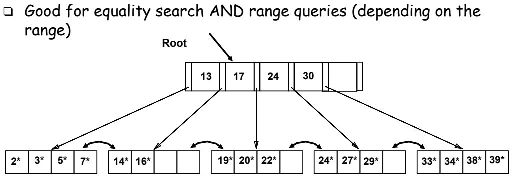
Example B+ Tree

- Example tree has height 2
- "`Select * from Skaters where sid = 5`"
  - ☆ Search begins at root, and key comparisons direct it to a leaf
  - ☆ Number of pages accessed:
    - three: root, leaf, data page with the corresponding record
  - ☆ Number of I/O:
    - depends of how much of tree in main memory
    - rough assumption: root always in main memory; index leaves and data pages not in main memory upon first access



Example B+ Tree

- "`Select * from Skaters where sid = 5`"
  - ☆ I/O costs:
    - one for leaf page with data entry, one for data page with data record
- "`Select * from Skaters where sid >= 33`"
  - ☆ I/O costs:
    - one for leaf page
    - four for data pages with records
- Good for equality search AND range queries (depending on the range)

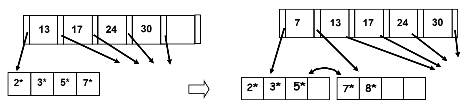


Inserting a Data Entry

- Find correct leaf *L*.
- Put data entry onto *L*.
  - ☆ If *L* has enough space, *done*!
  - ☆ Else, must *split L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - ☆ To split index node, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - ☆ Tree growth: gets *wider* or *one level taller at top.*

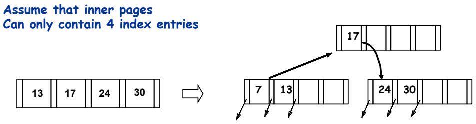
Inserting 8* into Example B+ Tree

Insert into Leaf with leaf split

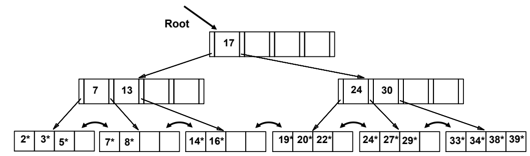


Insert into internal node with node split

Assume that inner pages
Can only contain 4 index entries



Example: After Inserting 8*



- Notice that root was split, leading to increase in height.
- In this example, we can avoid split by redistributing entries; however, this is usually not done in practice.

Data Entry Alternatives: indirect Indexing

- Indirect Indexing I
  - so far: <**k**, rid of data record with search key value **k**> (indirect indexing)
  - on non-primary key search key: (10, rid1), (10, rid2), (10, rid3), …
    - several entries with the same search key side by side
- Indirect indexing II
  - <**k**, list of rids of data records with search key **k**> (indirect indexing)
  - on non-primary key search key: (10, rid1, rid2, rid3)…
- Comparison:
  - first requires more space (search key repeated)
  - second has variable length data entries
  - second can have large data entries that span a page

Direct Indexing

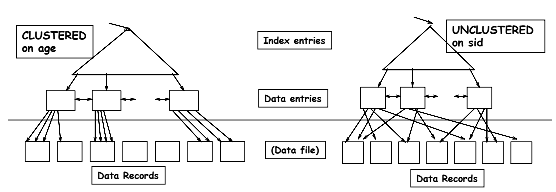
- Instead of data-entries in index leaves containing rids, they could contain the entire tuple
  - data-entry = tuple
  - no extra data pages
- This is kind of a sorted file with an index on top

Index Classification

- <u>Primary vs. secondary</u>: If search key contains primary key, then called primary index.
  - *Unique* index: Search key contains a candidate key.
- <u>Clustered vs. unclustered</u>:
  - clustered:
    - Relation in file sorted by the search key attributes of the index
  - unclustered:
    - Relation in heap file or sorted by an attribute different to the search key attribute of the index.
  - A file can be clustered on at most one search key.
  - Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!

Clustered vs. Unclustered Index

- Example for Skaters:
  - clustered on age
  - unclustered on sid



B+-tree cost example

- ☆ Relation R(A,B,C,D,E,F)
- ☆ A and B are int (each 6 Bytes), C-F is char[40] (160 Bytes)
  - Size of tuple: 172 Bytes
- ☆ 200,000 tuples
- ☆ Each data page has 4 K and is around 80% full
  - 200,000 / ((0.8.*4000)/172)
  - 200,000*172/(0.8*4000) = 10750 pages
- ☆ Values of B are within [0;19999] uniform distribution
- ☆ Non-clustered B-tree for attribute B, indirect indexing (2)
- ☆ An index page has 4K and intermediate pages are filled between 50% - 100%
- ☆ The size of an rid = 10 Bytes
- ☆ The size of a pointer in intermediate pages: 8 Bytes
- ☆ Index entry size in root and intermediate pages:
  - size(key)+size(pointer) = 6 Bytes + 8 Bytes = 14 Bytes

Size of B+tree

- ☆ The average number of rids per data entry
  - Number of tuples / different values (if uniform) (Example 200,000/20,000 = 10)
- ☆ The average length per data entry:
  - Key value + #rids * size of rid (Example: 6 + 10*10 = 106)
- ☆ The average number of data entries per leaf page:
  - Fill-rate * page-size / length of data entry
  - Example: 0.75*4000 / 106 = 28 entries per page
- ☆ The estimated number of leaf pages:
  - Number of entries = number of different values / #entries per page
  - Example 20000 / 28 = 715
- ☆ Number of entries intermediate page:
  - Fill-rate * page-size /length of index entry
  - Min fill-rate: 0.5, max fill rate: 1
  - Example: 0.5 * 4000 / 14 = 143 entries ; 1* 4000/14 = 285 entries
- ☆ Height is 3: the root has between three and four children
  - Three children: each child has around 715/3 = 238 entries
  - Four children: each child has around 715/4 = 179 entries

B+ Trees in Practice

- Typical order d of inner nodes: 100 (I.e., an inner node has between 100 and 200 index entries)
  - ☆ Typical fill-factor: 67%.
  - ☆ average fanout = 133
- Leaf nodes have often less entries since data entries larger (rids)
- Typical capacities
  - ☆ Height 4: 133^4 = 312,900,721 records
  - ☆ Height 3: 133^3 = 2,352,637 records
  - ☆ Height 2: 133^2 = 17,689 records
- Can often hold top levels in buffer pool:
  - ☆ Level 1 (root) = 1 page = 4 Kbytes
  - ☆ Level 2 = 133 pages = 0.5 Mbyte
  - ☆ Level 3 = 17,689 pages = 70 MBytes

Index in DB2

- **Simple**
  - ☆ `CREATE INDEX ind1 ON Skaters(sid);`
  - ☆ `DROP INDEX ind1;`
- **Index also good for referential integrity (uniqueness)**
  - ☆ `CREATE UNIQUE INDEX indname ON Skaters(name)`
- **Additional attributes**
  - ☆ `CREATE UNIQUE INDEX ind1 ON Skaters(sid) INCLUDE(name)`
  - ☆ **Index only on sid**
  - ☆ **Data entry contains key value (sid) + name + rid**
  - ☆ `SELECT name FROM Skaters WHERE sid = 100`
    - **Can be answered without accessing the real data pages of Skaters relation!**
- **Clustered index**
  - ☆ `CREATE INDEX ind1 on Skaters(sid) CLUSTER`

Index in DB2

- Index on multiple attributes:
  - ☆ `CREATE INDEX ind1 ON Skaters(age,rating);`
  - ☆ **Order is important:**
    - **Here data entries are first ordered by age**
    - **Skaters with the same age are then ordered by rating**
  - ☆ **Supports:**
    - `SELECT * FROM Skaters WHERE age = 20;`
    - `SELECT * FROM Skaters WHERE age = 20 AND rating < 5;`
  - ☆ **Does not support**
    - `SELECT * FROM Skaters WHERE rating < 5;`