CompSci 516
Data Intensive Computing Systems

Lecture 7
Storage and
Index

Instructor: Sudeepa Roy

Announcements

- HW1 deadline this week:
  - Due on 09/21 (Thurs), 11:55 pm, no late days
- Project proposal deadline:
  - Preliminary idea and team members due by tonight 09/18 (Mon), 11:55 pm by email to the instructor
  - Proposal due on sakai by 09/25 (Mon), 11:55 pm
- Everyone should be in a group now
  - otherwise let the instructor know asap

Reading Material

- [RG]
  - Storage: Chapters 8.1, 8.2, 8.4, 9.4-9.7
  - Index: 8.3, 8.5
  - Tree-based index: Chapter 10.1-10.7
  - Hash-based index: Chapter 11

Additional reading
- [GUW]
  - Chapters 8.3, 14.1-14.4

Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.

Storage
(contd. from Lecture 6)

Recap

- Typical DBMS hierarchy
- Disk and main memory/buffer pool
- Unit = page or block
  - page replacement strategies
  - dirty bit
  - pin

Today

- How are pages stored in a file?
- How are records stored in a page?
  - Fixed length records
  - Variable length records
- How are fields stored in a record?
  - Fixed length fields/records
  - Variable length fields/records

Files of Records

- Page or block is OK when doing I/O, but higher levels of DBMS operate on records, and files of records
- FILE: A collection of pages, each containing a collection of records
- Must support:
  - insert/delete/modify record
  - read a particular record (specified using record id)
  - scan all records (possibly with some conditions on the records to be retrieved)

File Organization

- File organization: Method of arranging a file of records on external storage
  - One file can have multiple pages
  - Record id (rid) is sufficient to physically locate the page containing the record on disk
  - Indexes are data structures that allow us to find the record ids of records with given values in index search key fields
- NOTE: Several uses of "keys" in a database
  - Primary/foreign/candidate/super keys
  - Index search keys

Alternative File Organizations

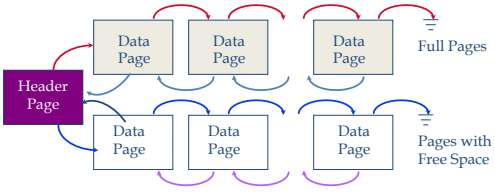
Many alternatives exist, each ideal for some situations, and not so good in others:

- Heap (random order) files: Suitable when typical access is a file scan retrieving all records
- Sorted Files: Best if records must be retrieved in some order, or only a "range" of records is needed.
- Indexes: Data structures to organize records via trees or hashing
  - Like sorted files, they speed up searches for a subset of records, based on values in certain ("search key") fields
  - Updates are much faster than in sorted files

Unordered (Heap) Files

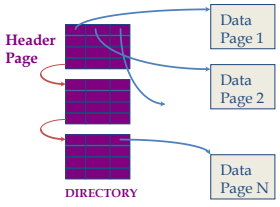
- Simplest file structure contains records in no particular order
- As file grows and shrinks, disk pages are allocated and de-allocated
- To support record level operations, we must:
  - keep track of the pages in a file
  - keep track of free space on pages
  - keep track of the *records* on a page
- There are many alternatives for keeping track of this

Heap File Implemented as a List



- The header page id and Heap file name must be stored someplace
- Each page contains 2 `pointers' plus data
- Problem?
  - to insert a new record, we may need to scan several pages on the free list to find one with sufficient space

Heap File Using a Page Directory

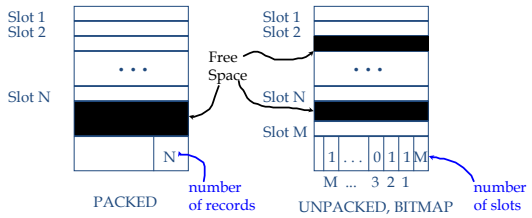


- The entry for a page can include the number of free bytes on the page.
- The directory is a collection of pages
  - linked list implementation of directory is just one alternative
  - Much smaller than linked list of all heap file pages!

How do we arrange a collection of records on a page?

- Each page contains several slots
  - one for each record
- Record is identified by <page-id, slot-number>
- Fixed-Length Records
- Variable-Length Records
- For both, there are options for
  - Record formats (how to organize the fields within a record)
  - Page formats (how to organize the records within a page)

Page Formats: Fixed Length Records

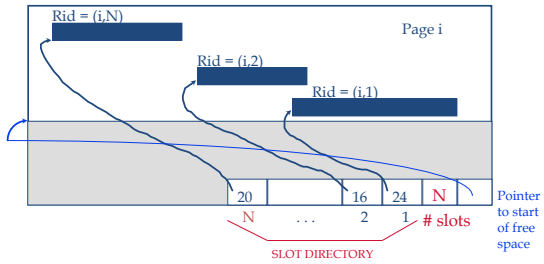


- Record id = <page id, slot #>
- Packed: moving records for free space management changes rid; may not be acceptable
- Unpacked: use a bitmap – scan the bit array to find an empty slot
- Each page also may contain additional info like the id of the next page (not shown)

Page Formats: Variable Length Records

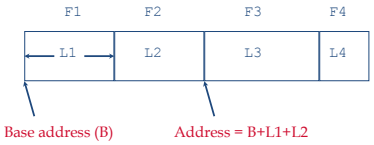
- Need to find a page with the right amount of space
  - Too small – cannot insert
  - Too large – waste of space
- if a record is deleted, need to move the records so that all free space is contiguous
  - need ability to move records within a page
- Can maintain a directory of slots (next slide)
  - Slot contains <record-offset, record-length>
  - deletion = set record-offset to -1
- Record-id rid = <page, slot-in-directory> remains unchanged

Page Formats: Variable Length Records



- Can move records on page without changing rid
  - so, attractive for fixed-length records too
- Store (record-offset, record-length) in each slot
- rid-s unaffected by rearranging records in a page

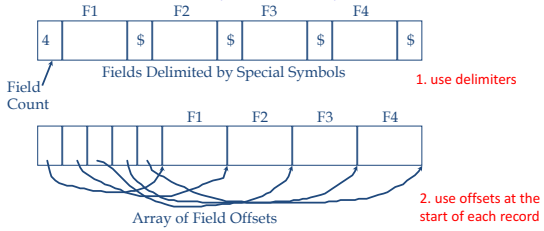
Record Formats: Fixed Length



- Each field has a fixed length
  - for all records
  - the number of fields is also fixed
  - fields can be stored consecutively
- Information about field types same for all records in a file
  - stored in system catalogs
- Finding i-th field does not require scan of record
  - given the address of the record, address of a field can be obtained easily

Record Formats: Variable Length

- Cannot use fixed-length slots for records
- Two alternative formats (# fields is fixed):



- Second offers direct access to i-th field, efficient storage of nulls (special don't know value); small directory overhead
- Modification may be costly (may grow the field and not fit in the page)

Indexes

Indexes

- An index on a file speeds up selections on the search key fields for the index
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - "Search key" is not the same as "key"
    key = minimal set of fields that uniquely identify a tuple
- An index contains a collection of data entries, and supports efficient retrieval of all data entries k* with a given key value k

Remember Terminology

- Index search key (key): k
  - Used to search a record
- Data entry : k*
  - Pointed to by k
  - Contains record id(s) or record itself
- Records or data
  - Actual tuples
  - Pointed to by record ids

INDEX does this

Alternatives for Data Entry k* in Index k

- In a data entry k* we can store:
  1. (Alternative 1) The actual data record with key value **k,** or
  2. (Alternative 2) <**k**, rid>
     - rid = record of data record with search key value **k**, or
  3. (Alternative 3) <**k**, rid-list>
     - list of record ids of data records with search key **k**>
- Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value **k**

Alternatives for Data Entries: Alternative 1

- In a data entry k* we can store:
  1. The actual data record with key value **k**
  2. <**k**, rid>
     - rid = record of data record with search key value **k**
  3. <**k**, rid-list>
     - list of record ids of data records with search key **k**>

Advantages/ Disadvantages?

- Index structure is a file organization for data records
  - instead of a Heap file or sorted file
- How many different indexes can use Alternative 1?
- At most one index can use Alternative 1
  - Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency
- If data records are very large, #pages with data entries is high
  - Implies size of auxiliary information in the index is also large

Alternatives for Data Entries: Alternative 2, 3

- In a data entry k* we can store:
  1. The actual data record with key value **k**
  2. <**k**, rid>
     - rid = record of data record with search key value **k**
  3. <**k**, rid-list>
     - list of record ids of data records with search key **k**>

Advantages/ Disadvantages?

- Data entries typically much smaller than data records
  - So, better than Alternative 1 with large data records
  - Especially if search keys are small.
- Alternative 3 more compact than Alternative 2
  - but leads to variable-size data entries even if search keys have fixed length.

Index Classification

- Primary vs. secondary
- Clustered vs. unclustered
- Tree-based vs. Hash-based

Primary vs. Secondary Index

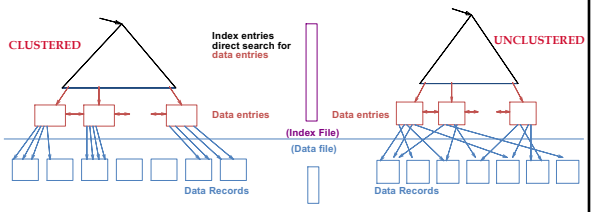
- If search key contains primary key, then called primary index, otherwise secondary
  - Unique index: Search key contains a candidate key
- Duplicate data entries:
  - if they have the same value of search key field k
  - Primary/unique index never has a duplicate
  - Other secondary index can have duplicates

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
  - Alternative 1 implies clustered
  - Alternative 2, 3 are typically unclustered
    - unless sorted according to the search key
  - Sometimes, clustered also implies Alternative 1
    - since sorted files are rare
  - A file can be clustered on at most one search key
  - Cost of retrieving data records (range queries) through index varies greatly based on whether index is clustered or not

Clustered vs. Unclustered Index

- Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file
- To build clustered index, first sort the Heap file
  - with some free space on each page for future inserts
  - Overflow pages may be needed for inserts
  - Thus, data records are `close to', but not identical to, sorted



Methods for indexing

- Tree-based
- Hash-based
- (in detail later)

System Catalogs

- For each index:
  - structure (e.g., B+ tree) and search key fields
- For each relation:
  - name, file name, file structure (e.g., Heap file)
  - attribute name and type, for each attribute
  - index name, for each index
  - integrity constraints
- For each view:
  - view name and definition
- Plus statistics, authorization, buffer pool size, etc.
- (described in [RG] 12.1)

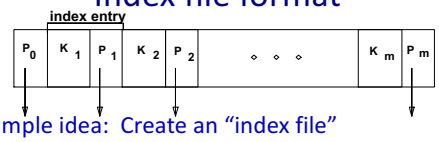
Catalogs are themselves stored as relations!

Tree-based Index and B+-Tree

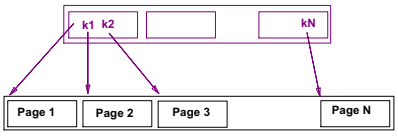
Range Searches

- ``*Find all students with gpa > 3.0*''
  - If data is in sorted file, do binary search to find first such student, then scan to find others.
  - Cost of binary search can be quite high.

Index file format



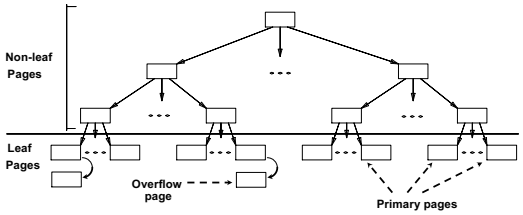
- Simple idea: Create an "index file"
  - <first-key-on-page, pointer-to-page>, sorted on keys



Can do binary search on (smaller) index file but may still be expensive: apply this idea repeatedly

Indexed Sequential Access Method (ISAM)

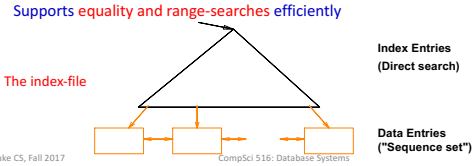
- Leaf-pages contain data entry – also some overflow pages
- DBMS organizes layout of the index – a static structure
- If a number of inserts to the same leaf, a long overflow chain can be created
  - affects the performance



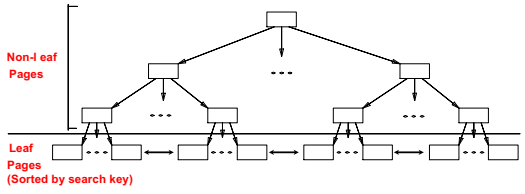
Leaf pages contain data entries.

B+ Tree

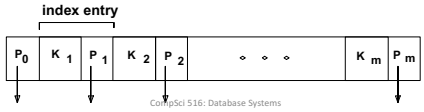
- Most Widely Used Index
  - a dynamic structure
- Insert/delete at $\log_F N$ cost = height of the tree
  - F = fanout, N = no. of leaf pages
  - tree is maintained height-balanced
- Minimum 50% occupancy
  - Each node contains **d** <= m <= 2**d** entries
  - Root contains 1 <= m <= 2d entries
  - The parameter **d** is called the order of the tree
- Supports equality and range-searches efficiently



B+ Tree Indexes



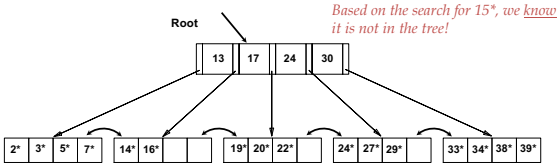
- Leaf pages contain data entries, and are chained (prev & next)
- Non-leaf pages have index entries; only used to direct searches:



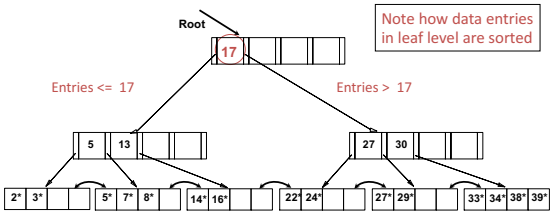
Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf
- Search for 5*, 15*, all data entries >= 24* ...

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



Example B+ Tree



- Find
  - 28*?
  - 29*?
  - All > 15* and < 30*

B+ Trees in Practice

- Typical order: d = 100. Typical fill-factor: 67%
  - average fanout F = 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

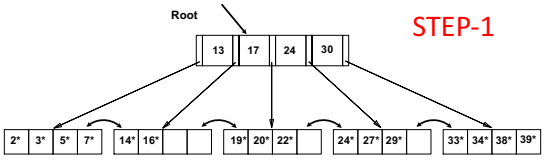
Inserting a Data Entry into a B+ Tree

See this slide later, First, see examples on the next few slides

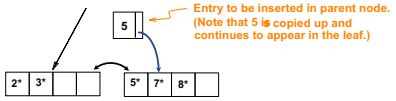
- Find correct leaf L
- Put data entry onto L
  - If L has enough space, done
  - Else, must split L
    - into L and a new node L2
    - Redistribute entries evenly, copy up middle key.
    - Insert index entry pointing to L2 into parent of L.
- This can happen recursively
  - To split index node, redistribute 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

STEP-1



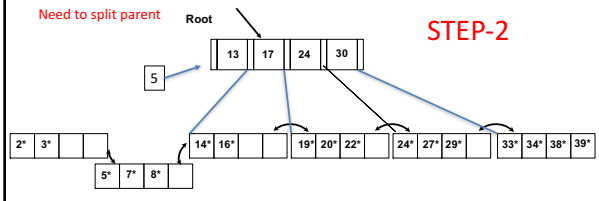
- Copy-up: 5 appears in leaf and the level above
- Observe how minimum occupancy is guaranteed



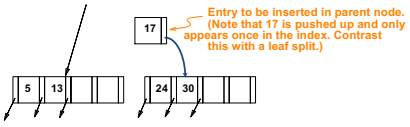
Inserting 8* into Example B+ Tree

Need to split parent

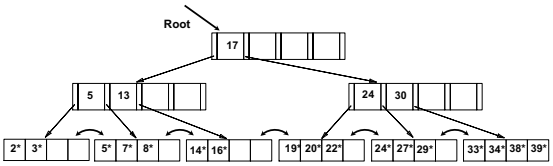
STEP-2



- Note difference between copy-up and push-up
- What is the reason for this difference?
- All data entries must appear as leaves
  - (for easy range search)
- no such requirement for indexes
  - (so avoid redundancy)



Example B+ Tree After Inserting 8*



- Notice that root was split, leading to increase in height.
- In this example, we can avoid split by re-distributing entries (insert 8 to the 2nd leaf node from left and copy it up instead of 13)
  - however, this is usually not done in practice – since need to access 1-2 extra pages always (for two siblings), and average occupancy may remain unaffected as the file grows

Deleting a Data Entry from a B+ Tree

Each non-root node contains **d** <= m <= 2**d** entries

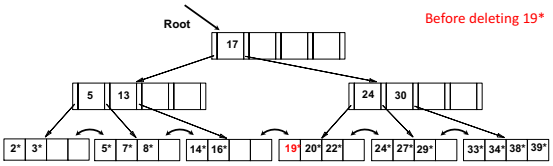
- Start at root, find leaf L where entry belongs
- Remove the entry
  - If L is at least half-full, done!
  - If L has only **d-1** entries,
    - Try to re-distribute, borrowing from sibling (adjacent node with same parent as L)
    - If re-distribution fails, merge L and sibling

See this slide later, First, see examples on the next few slides

- If merge occurred, must delete entry (pointing to L or sibling) from parent of L
- Merge could propagate to root, decreasing height

Example Tree: Delete 19*

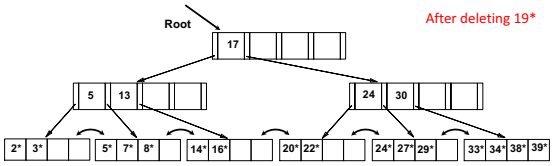
Before deleting 19*



- We had inserted 8*
- Now delete 19*
- Easy

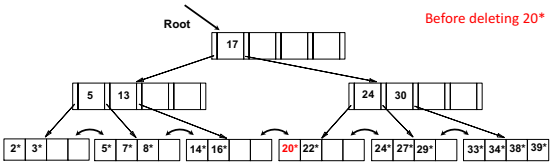
Example Tree: Delete 19*

After deleting 19*



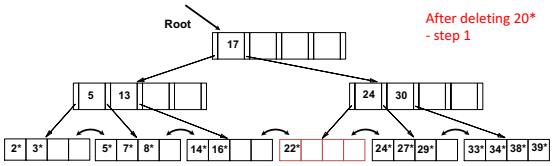
Example Tree: Delete 20*

Before deleting 20*



Example Tree: Delete 20*

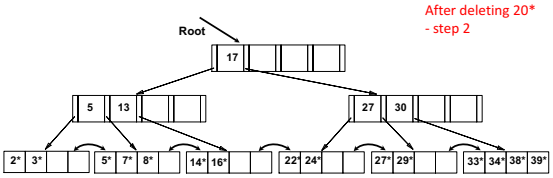
After deleting 20* - step 1



- < 2 entries in leaf-node
- Redistribute

Example Tree: Delete 20*

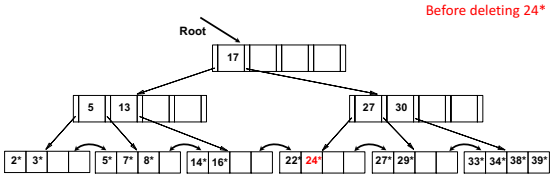
After deleting 20*
- step 2



- Notice how middle key is copied up

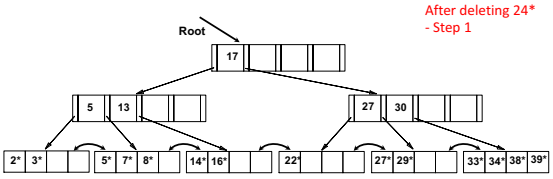
Example Tree: ... And Then Delete 24*

Before deleting 24*



Example Tree: ... And Then Delete 24*

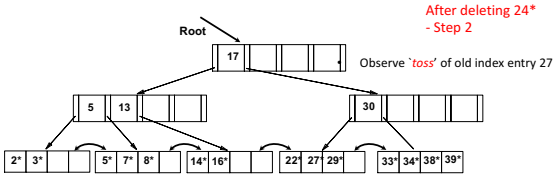
After deleting 24*
- Step 1



- Once again, imbalance at leaf
- Can we borrow from sibling(s)?
- No – d-1 and d entries (d = 2)
- Need to merge

Example Tree: ... And Then Delete 24*

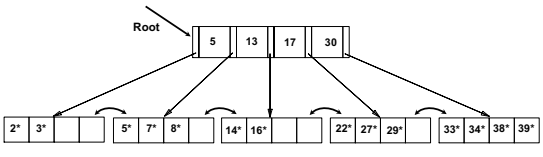
After deleting 24*
- Step 2



- Imbalance at parent
- Merge again
- But need to "pull down" root index entry

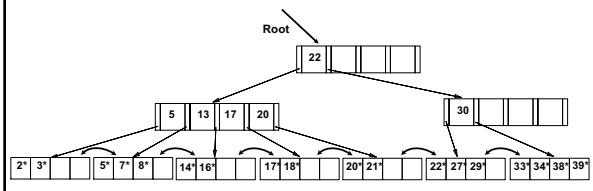
because, three index 5, 13, 30 but five pointers to leaves

Final Example Tree



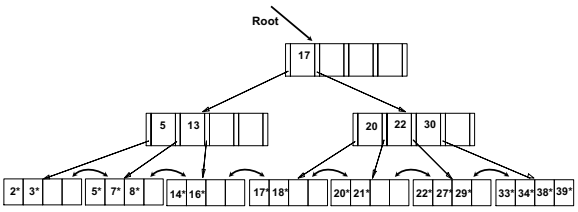
Example of Non-leaf Re-distribution

- An intermediate tree is shown
- In contrast to previous example, can re-distribute entry from left child of root to right child



After Re-distribution

- Intuitively, entries are re-distributed by `pushing through' the splitting entry in the parent node.
  - It suffices to re-distribute index entry with key 20; we've re-distributed 17 as well for illustration.



Duplicates

- First Option:
  - The basic search algorithm assumes that all entries with the same key value resides on the same leaf page
  - If they do not fit, use overflow pages (like ISAM)
- Second Option:
  - Several leaf pages can contain entries with a given key value
  - Search for the left most entry with a key value, and follow the leaf-sequence pointers
  - Need modification in the search algorithm
- if k* = <k, rid>, several entries have to be searched
  - Or include rid in k – becomes unique index, no duplicate
  - If k* = <k, rid-list>, some solution, but if the list is long, again a single entry can span multiple pages

A Note on `Order'

- Order (d)
  - denotes minimum occupancy
- replaced by physical space criterion in practice (`at least half-full')
  - Index pages can typically hold many more entries than leaf pages
  - Variable sized records and search keys mean different nodes will contain different numbers of entries.
  - Even with fixed length fields, multiple records with the same search key value (duplicates) can lead to variable-sized data entries (if we use Alternative (3))

Summary

- Tree-structured indexes are ideal for range-searches, also good for equality searches
- ISAM is a static structure
  - Only leaf pages modified; overflow pages needed
  - Overflow chains can degrade performance unless size of data set and data distribution stay constant
- B+ tree is a dynamic structure
  - Inserts/deletes leave tree height-balanced; $\log_F N$ cost
  - High fanout (**F**) means depth rarely more than 3 or 4
  - Almost always better than maintaining a sorted file
  - Most widely used index in database management systems because of its versatility.
  - One of the most optimized components of a DBMS
- Next: Hash-based index