CompSci 516
Data Intensive Computing Systems

Lecture 8
Index

Instructor: Sudeepa Roy

Announcements

- HW1 due tomorrow:
  - Due on 09/21 (Thurs), 11:55 pm, no late days
  - Submit early – even if not complete
- Informal project proposal feedback soon

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.

Today

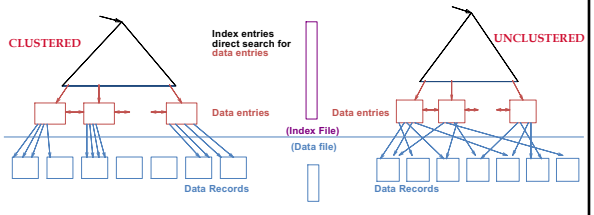
- How are pages stored in a file?
- How are records stored in a page?
  - Fixed length records
  - Variable length records
- How are fields stored in a record?
  - Fixed length fields/records
  - Variable length fields/records

Recap

- Storage :
  - Files -> Records -> Fields
  - Fixed and variable length
- Index
  - Search key k -> Data entry k* -> Record
  - Alternative 1/2/3 for k*
  - Primary/secondary, clustered/unclustered
- Today
  - B+ tree index
  - Hash based index

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

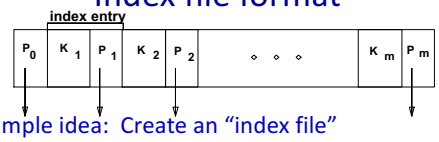


Tree-based Index and B+-Tree

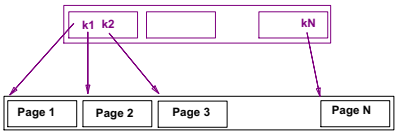
Range Searches

- ``*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



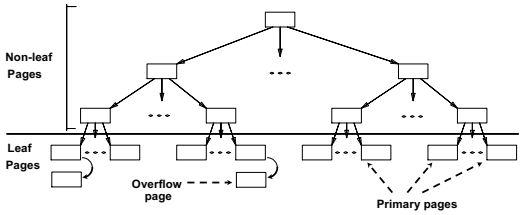
- Simple idea: Create an "index file"
  - <first-key-on-page, pointer-to-page>, sorted on keys



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

Indexed Sequential Access Method (ISAM)

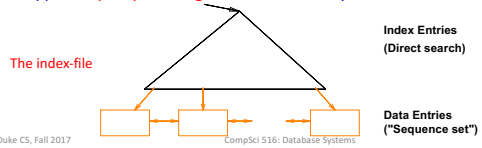
- Leaf-pages contain data entry – also some overflow pages
- DBMS organizes layout of the index – a static structure
- If a number of inserts to the same leaf, a long overflow chain can be created
  - affects the performance



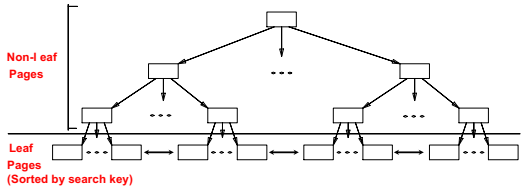
Leaf pages contain data entries.

B+ Tree

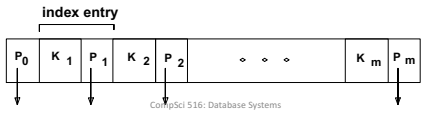
- Most Widely Used Index
  - a dynamic structure
- Insert/delete at $\log_F N$ cost = height of the tree (cost = I/O)
  - F = fanout, N = no. of leaf pages
  - tree is maintained height-balanced
- Minimum 50% occupancy
  - Each node contains **d** <= m <= 2**d** entries
  - Root contains 1 <= m <= 2d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



B+ Tree Indexes

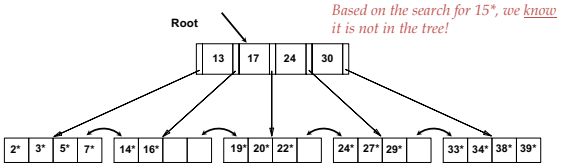


- Leaf pages contain data entries, and are chained (prev & next)
- Non-leaf pages have index entries; only used to direct searches:



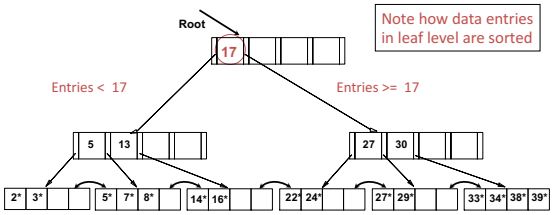
Example B+ Tree

- 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



Note how data entries in leaf level are sorted

- 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

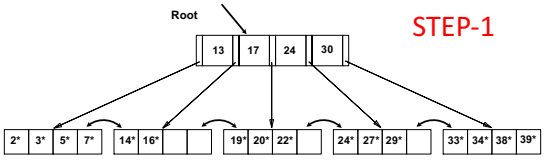
Inserting a Data Entry into a B+ Tree

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

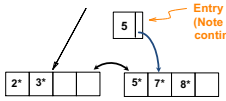
- Find correct leaf L
- Put data entry onto L
  - If L has enough space, done
  - Else, must split L
    - into L and a new node L2
    - Redistribute entries evenly, copy up middle key.
    - Insert index entry pointing to L2 into parent of L.
- This can happen recursively
  - To split index node, redistribute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

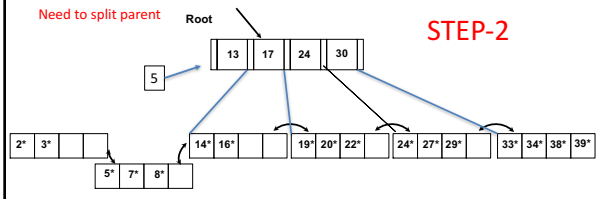


Entry to be inserted in parent node. (Note that 5 is copied up and continues to appear in the leaf.)

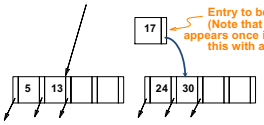
Inserting 8* into Example B+ Tree

Need to split parent

STEP-2

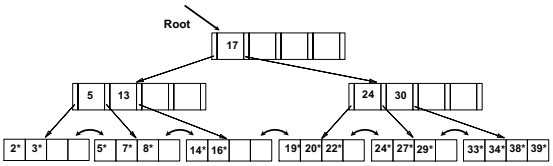


- Note difference between copy-up and push-up
- What is the reason for this difference?
- All data entries must appear as leaves
  - (for easy range search)
- no such requirement for indexes
  - (so avoid redundancy)



Entry to be inserted in parent node. (Note that 17 is pushed up and only appears once in the index. Contrast this with a leaf split.)

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

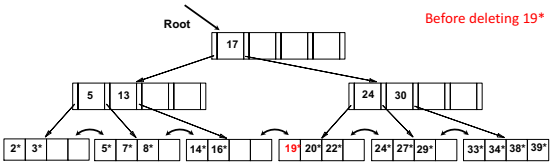
- Start at root, find leaf L where entry belongs
- Remove the entry
  - If L is at least half-full, done!
  - If L has only **d-1** entries,
    - Try to re-distribute, borrowing from sibling (adjacent node with same parent as L)
    - If re-distribution fails, merge L and sibling

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*

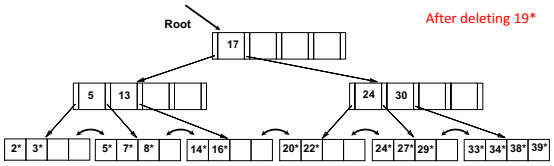
Before deleting 19*



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

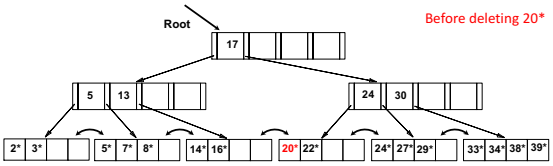
Example Tree: Delete 19*

After deleting 19*



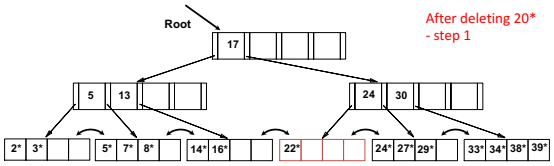
Example Tree: Delete 20*

Before deleting 20*



Example Tree: Delete 20*

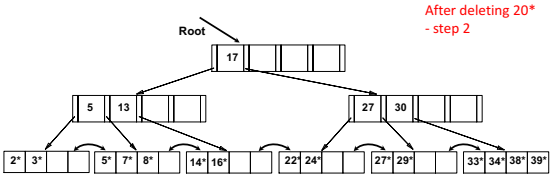
After deleting 20* - step 1



- < 2 entries in leaf-node
- Redistribute

Example Tree: Delete 20*

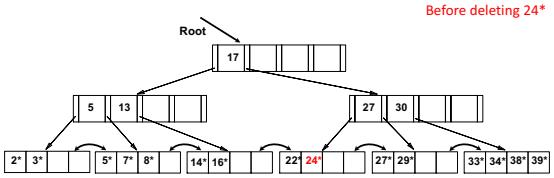
After deleting 20*
- step 2



- Notice how middle key is copied up

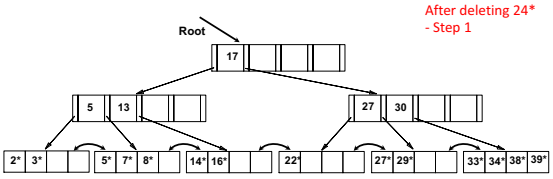
Example Tree: ... And Then Delete 24*

Before deleting 24*



Example Tree: ... And Then Delete 24*

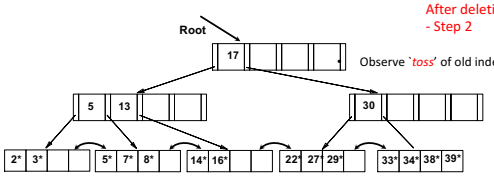
After deleting 24*
- Step 1



- 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*

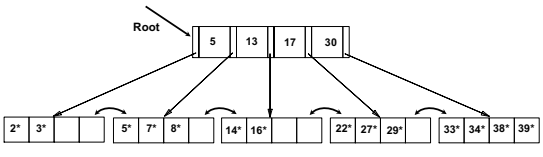
After deleting 24*
- Step 2



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

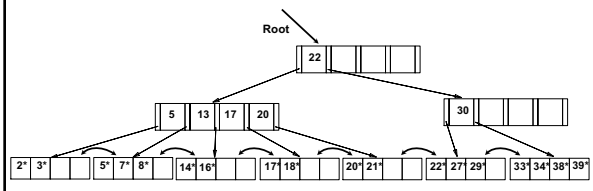
because, three index 5, 13, 30 but five pointers to leaves

Final Example Tree



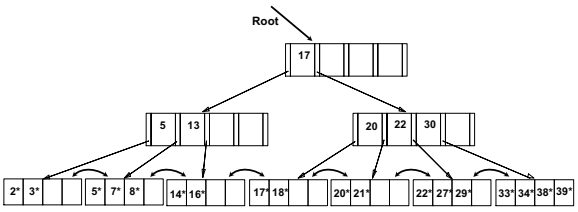
Example of Non-leaf Re-distribution

- 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

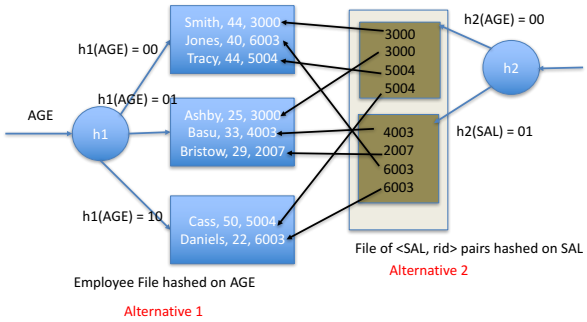
Hash-based Index

Hash-Based Indexes

- Records are grouped into buckets
  - Bucket = primary page plus zero or more overflow pages
- Hashing function **h**:
  - **h**(r) = bucket in which (data entry for) record r belongs
  - **h** looks at the search key fields of r
  - No need for "index entries" in this scheme

Example: Hash-based index

Index organized file hashed on AGE, with Auxiliary index on SAL

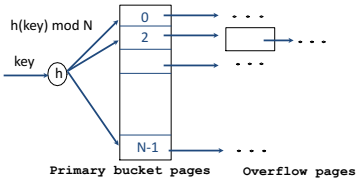


Introduction

- Hash-based indexes are best for equality selections
  - Find all records with name = "Joe"
  - Cannot support range searches
  - But useful in implementing relational operators like join (later)
- Static and dynamic hashing techniques exist
  - trade-offs similar to ISAM vs. B+ trees

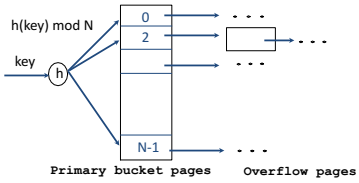
Static Hashing

- Pages containing data = a collection of buckets
  - each bucket has one primary page, also possibly overflow pages
  - buckets contain data entries k*



Static Hashing

- # primary pages fixed
  - allocated sequentially, never de-allocated, overflow pages if needed.
- h(k) mod N = bucket to which data entry with key k belongs
  - N = # of buckets



Static Hashing

- Hash function works on search key field of record r
  - Must distribute values over range 0 ... N-1
  - h(key) = (a * key + b) usually works well
    - bucket = h(key) mod N
  - a and b are constants – chosen to tune h
- Advantage:
  - #buckets known – pages can be allocated sequentially
  - search needs 1 I/O (if no overflow page)
  - insert/delete needs 2 I/O (if no overflow page) (why 2?)
- Disadvantage:
  - Long overflow chains can develop if file grows and degrade performance
  - Or waste of space if file shrinks
- Solutions:
  - keep some pages say 80% full initially
  - Periodically rehash if overflow pages (can be expensive)
  - or use Dynamic Hashing

Dynamic Hashing Techniques

- Extendible Hashing
- Linear Hashing

Extendible Hashing

- Consider static hashing
- Bucket (primary page) becomes full

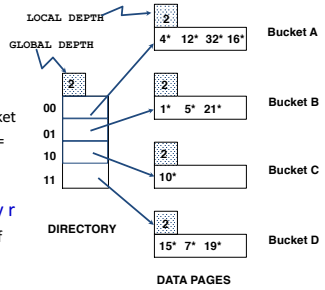
- Why not re-organize file by doubling # of buckets?
  - Reading and writing (double #pages) all pages is expensive

- Idea: Use directory of pointers to buckets
  - double # of buckets by doubling the directory, splitting just the bucket that overflowed
  - Directory much smaller than file, so doubling it is much cheaper
  - Only one page of data entries is split
  - No overflow page (new bucket, no new overflow page)
  - Trick lies in how hash function is adjusted

Example

- Directory is array of size 4
  - each element points to a bucket
  - #bits to represent = log 4 = 2 = global depth

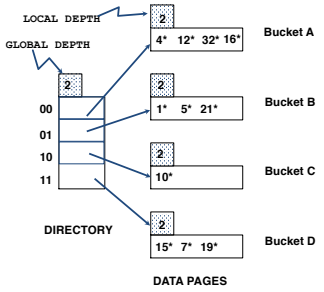
- To find bucket for search key r
  - take last global depth # bits of **h**(r)
  - assume h(r) = r
  - If **h**(r) = 5 = binary 101
  - it is in bucket pointed to by 01



Example

Insert:
- If bucket is full, split it
- allocate new page
- re-distribute

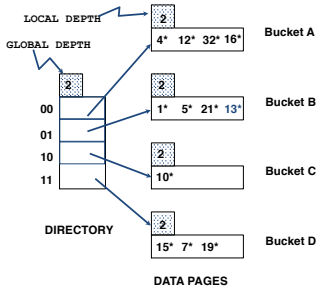
Suppose inserting 13*
- binary = 1101
- bucket 01
- Has space, insert



Example

Insert:
- If bucket is full, split it
- allocate new page
- re-distribute

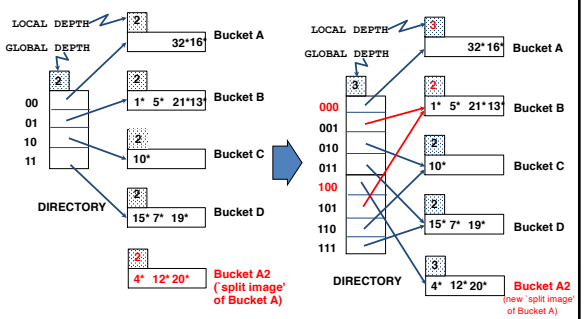
Suppose inserting 20*
- binary = 10100
- bucket 00
- Already full
- To split, consider last three bits of 10100
- Last two bits the same 00 – the data entry will belong to one of these buckets
- Third bit to distinguish them



Example

Global depth: Max # of bits needed to tell which bucket an entry belongs to

Local depth: # of bits used to determine if an entry belongs to this bucket
- also denotes whether a directory doubling is needed while splitting
- no directory doubling needed when 9* = 1001 is inserted (LD< GD)



When does bucket split cause directory doubling?

- Before insert, local depth of bucket = global depth
- Insert causes local depth to become > global depth
- directory is doubled by copying it over and `fixing' pointer to split image page

Comments on Extendible Hashing

- If directory fits in memory, equality search answered with one disk access (to access the bucket); else two.
  - 100MB file, 100 bytes/rec, 4KB page size, contains 10^6 records (as data entries) and 25,000 directory elements; chances are high that directory will fit in memory.
  - Directory grows in spurts, and, if the distribution *of hash values* is skewed, directory can grow large
  - Multiple entries with same hash value cause problems
- Delete:
  - If removal of data entry makes bucket empty, can be merged with `split image'
  - If each directory element points to same bucket as its split image, can halve directory.