LAB 7 Notes

Overview

- How does the DBMS storage data?
- Why I/O is so important for database operations?
- Why are indices useful?

Introduction

- I/O is way more expensive than any in-memory operation
- Disks are the most important external storage device
- The basic DBMS storage abstraction is a file of records
- Each record in a file should have a unique identifier (record id or rid)
- The buffer manager is responsible for loading and writing a record from/to disk
- The file and access methods layer sits on top of the buffer manager, asking it to retrieve or write a given page rid

File Organization and Indexing

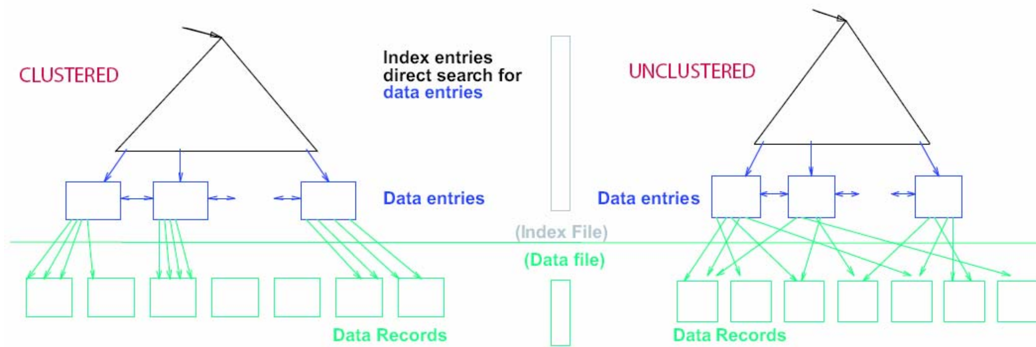
- A relation is typically stored as a file of records
- **Rows are packed into records** (pages)
- The simplest file structure is an unordered file or heap file
- Records are stored in random order
- The file manager is able to retrieve and store a particular record by its rid
- An **index** is a data structure that organizes data records to **optimize** certain kinds of retrieval operations
- They allow the efficient retrieval of records that satisfy a search condition on the search key
- Multiple indexes can be created on the same file (i.e table)

Data Entries

- A data entry is a record stored in an index file
- A data entry with search value k (denoted as k*) contains enough information to locate all records with search key value k
- There are 3 ways to store a data entry:
  - A data entry k* is an **actual data record** (with search key value k)
  - A data entry is a <k, rid> pair, where rid is the record id of the data record with search key k
  - A data entry is a <k, rid-list> pair, where rid − list is a list of record ids with search key value k

Clustered Indexes

- If the ordering of the data records is the same or close to the ordering of the data entries in some index, it is called clustered (indexing alternative 1) is clustered by definition
- Otherwise it is unclustered



Primary and Secondary Indices

- An index on the primary key is a primary index
- Other indexes are secondary indexes
- Two data entries are duplicates if the have same search key
- A primary index does not have duplicates
- A secondary index could have no duplicates if is specified as UNIQUE

Index Data Structures

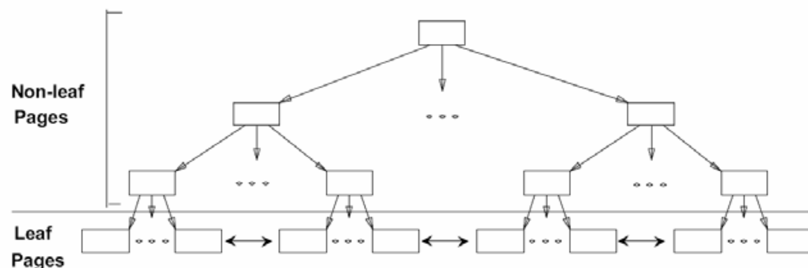
- Indexes are usually implemented in one of two ways:
  - Hash-based Indexing
  - Tree-based Indexing

Hash-based Indexing

- Use a hashing function to cluster records (and find them)
- Records in a file are clustered in buckets
- A bucket is composed of primary page and additional pages linked in a chain.
- The DBMS uses a hashing function to determine, for a given record, its bucket number
- Given a bucket number the DBMS should be able to retrieve the record in 1 or 2 I/Os (for the sake of this chapter, we will assume 1 I/O)
- If alternative (1) is used, the buckets contain the records, otherwise they contain the <k, rid> or <k, rid − list>

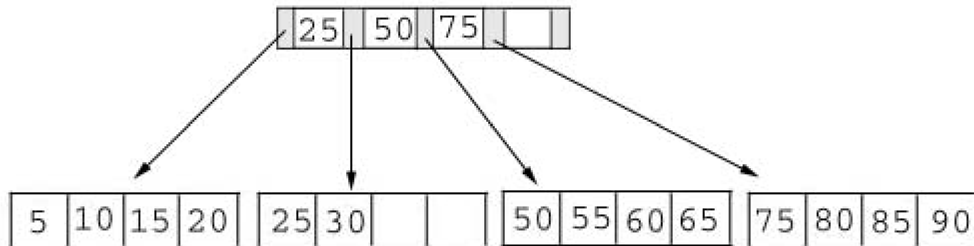
Tree-based Indexing

- The tree organizes the keys in order
- The inside nodes contain only keys, while the leafs contains the data entries

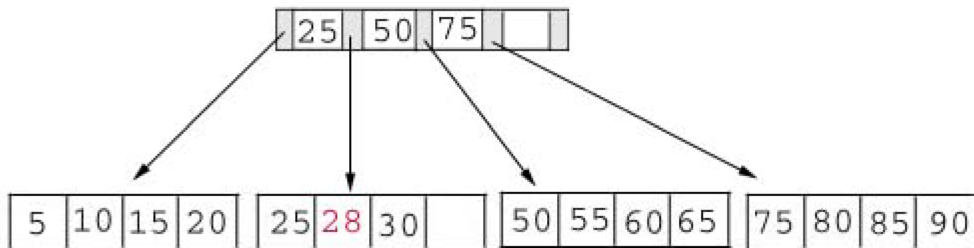


B+Tree

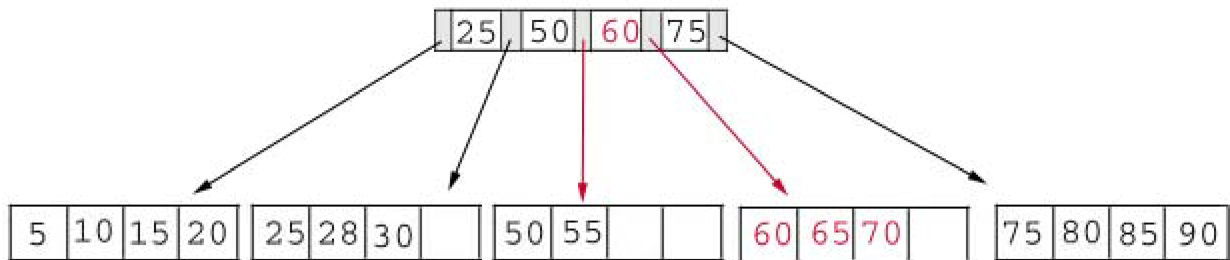
- B+ trees are the most commonly used
- They are guaranteed to be balanced
- The height of the tree is the length of a path from the root to the leaf (guaranteed to the always the same)



Add Record with Key 28



Add Record with Key 70



The middle key of 60 is placed in the index page between 50 and 75.

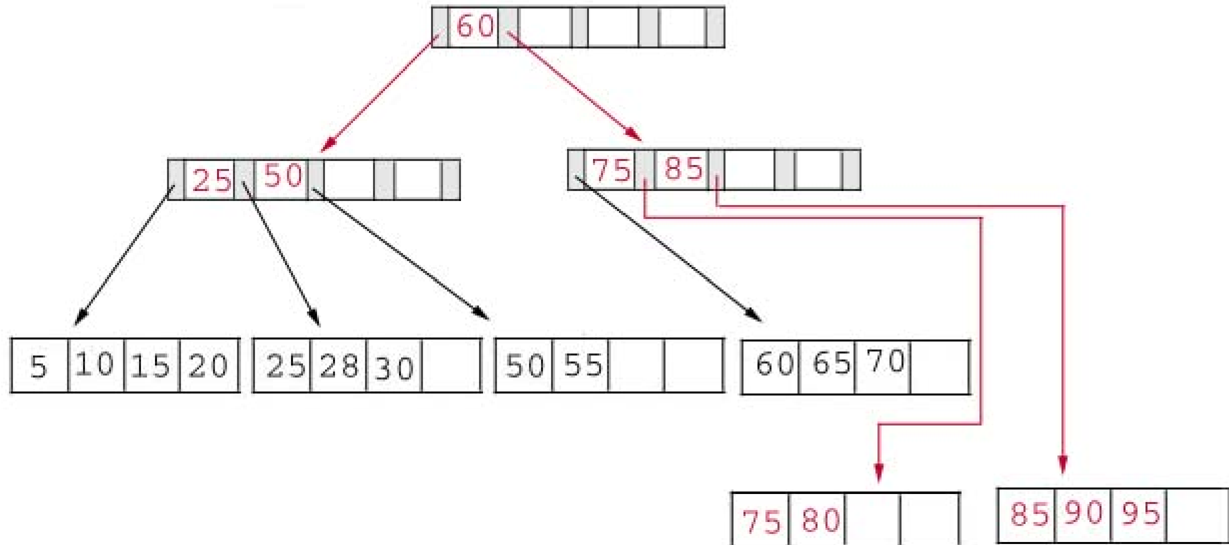
Add Record with Key 95

This record belongs in the page containing 75, 80, 85, and 90. Since this page is full we split it into two pages:

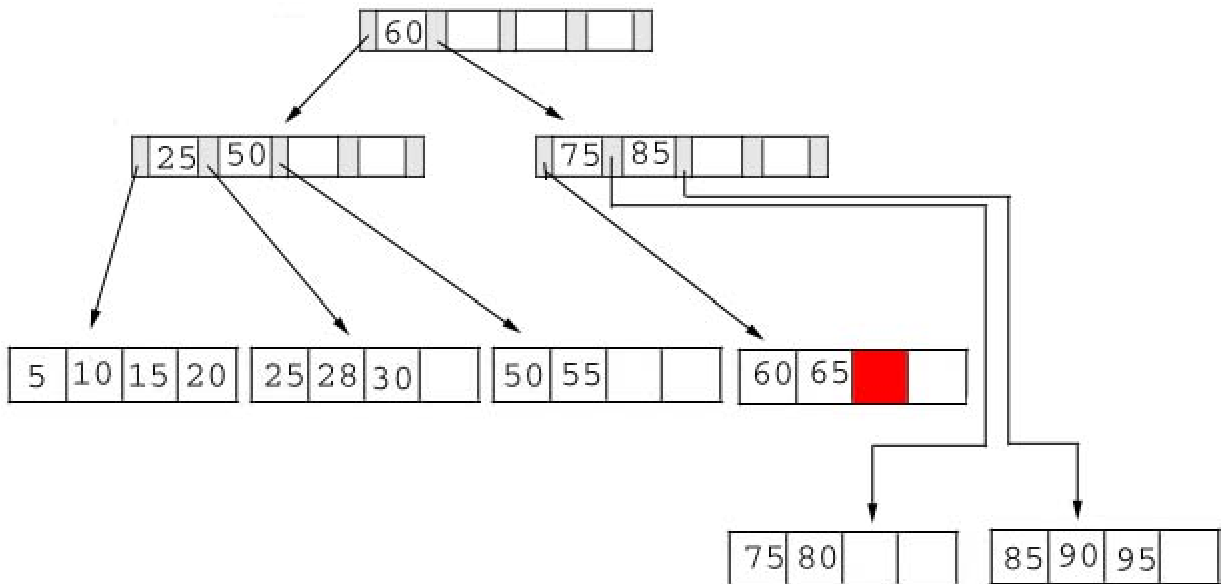
Left Leaf Page	Right Leaf Page
75 80	85 90 95

The middle key, 85, rises to the index page. Unfortunately, the index page is also full, so we split the index page:

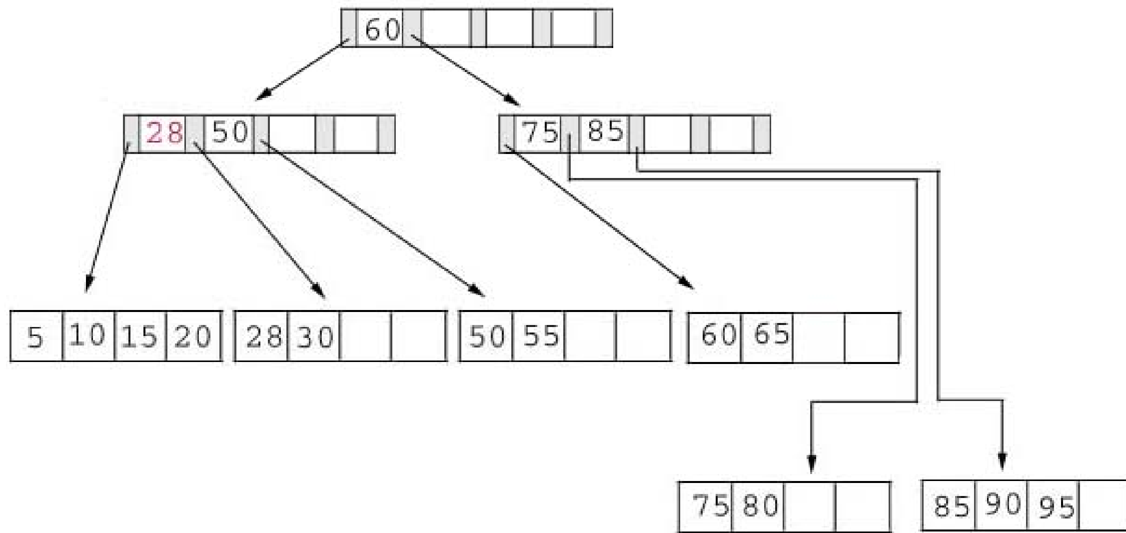
Left Index Page	Right Index Page	New Index Page
25 50	75 85	60



Delete Record with Key 70



Delete Record with Key 25

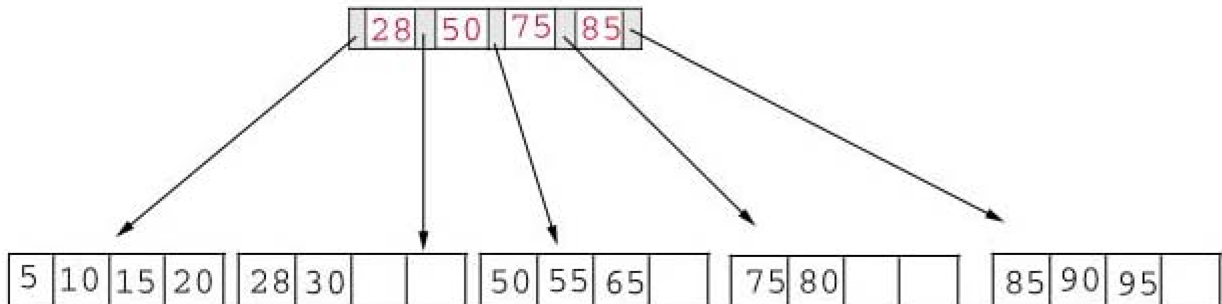


As our last example, we're going to delete 60 from the B+ tree. This deletion is interesting for several resasons:

1. The leaf page containing 60 (60 65) will be below the fill factor after the deletion. Thus, we must combine leaf pages.
2. With recombined pages, the index page will be reduced by one key. Hence, it will also fall below the fill factor. Thus, we must combine index pages.
3. Sixty appears as the only key in the root index page. Obviously, it will be removed with the deletion.

The following table shows the B+ tree after the deletion of 60. Notice that the tree contains a single index page.

Delete Record with Key 60



The contenders

- Heap file of employee records
- File of employee records, sorted on <age, sal>
- Cluster B+ tree with search key <age, sal>
- Heap file with an unclustered B+ tree index on <age, sal>
- Heap file with an unclustered hash index on <age, sal>

What we compare

We will use a very simple cost model

- B is the number of data pages
- R is the number of records per page
- D is the avg. time to read or write a page
- C is the cost of comparing a key
- F is the fan out of the B+ trees
- H is the time to apply the hash function

D ≈ 15ms, while C and H ≈ 100ns, so we will concentrate on I/O cost.

Heap Files

- Scan: BD
- Search with Equality: BD/2
- Range Search: BD
- Insert: 2D
- Delete: Search + D

Sorted Files

- Scan: BD
- Search with Equality: Dlog2B
- Range Search:Dlog2B+cost of retrieving sequential matching records
- Insert: Search + BD
- Delete: Search + BD

Clustered B+ Tree

- Fan out: F (F keys per inner node)
- In leave nodes, on average only 67% of the space is used, hence we need 1.5 times the number of leaves we would need if each node was full (1.5 B, as each node uses one page)
  - Scan: 1.5BD (scan all the leave nodes)
  - Search with Equality: DlogF 1.5B Use the tree, with depth
- proportional to logF (number of leave nodes)
  - Range Search:DlogF 1.5B+ cost of retrieving sequential
- matching rows (find first, scan sequential)
  - Insert: Search + D (search and insert)
  - Delete: Search + D (search and destroy)

Heap File with Unclustered Tree Index

- Assume Index data is 1/10th of the size of the data record
- So 10 times more index records < key, rid > than data records per leave page
- Total number of leave pages 0.15B (assuming 67% occupancy rate)
  - Scan:
    - Cost of scanning all index leave pages 0.15BD +
    - Per each key in the index, retrieve one data page (heap file) BRD.
      Total: BD(0.15 + R)
  - Search with Equality: Search in tree + 1 data page ,D(logF 0.15B + 1)
- Range Search: Search + cost of matching records D(logF 0.15B +matching rows)
- Insert: Insert into heap + insert into tree: 2D + D(logF 0.15B + 1)
- Delete: Search + 2D

Heap File with Unclustered Hash Index

- Assume: 80% occupancy rate and simple hash structures (no overflow).
- Hence and 8 index entries per page in index: total size of index 0.125B pages
  - Scan: Read of rids: 0.125BD + cost of retrieval of each data page: BRD.
    Total: BD(0.125 + R)
  - Search with Equality: 2D
  - Range Search: Search: BD
  - Insert: 4D
  - Delete: Search + 2D

File Type	Scan	Eq. Search	Range S.	Insert	Delete
Heap	BD	$0.5BD$	BD	$2D$	$Search + D$
Sorted Files	BD	$Dlog_2 B$	$Dlog_2 B + \#MP$	$Search + BD$	$Search + BD$
Clustered B+ tree	$1.5BD$	$Dlog_F B$	$log_F 1.5B + \#MP$	$Search + D$	$Search + D$
Unclusted B+ tree	$BD(0.15 + R)$	$D(log_F 0.15B + 1)$	$D(log_F 0.15B + \#MP)$	$3D + Dlog_F 0.15B$	$Search + 2D$
Unclusted hash	$BD(0.125 + R)$	$2D$	BD	$4D$	$Search + 2D$

#MP is the number of pages that contain matching records in a range search.

So, which one is better?