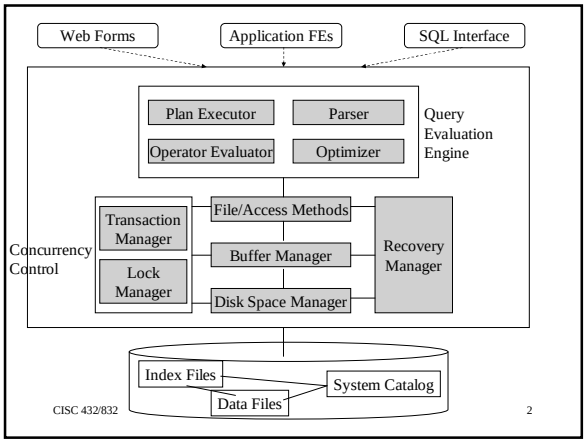
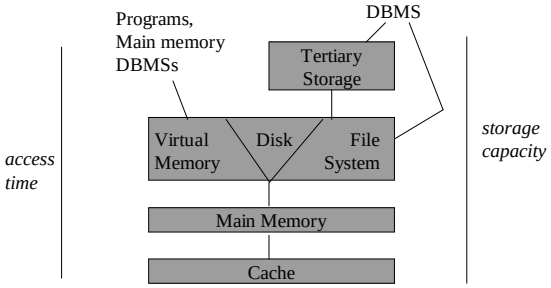
DBMS Storage and Indexing

Chs 8 - 11



The Memory Hierarchy



Simple Cost Model

- Assume
  - Simple system structure
  - Database too large to fit in memory
  - Each piece of data accessed by user must be initially retrieved from disk
  - Each disk page accessed individually – no blocked access
- **I/O cost is dominant** ⇒ number of block accesses is a good approximation to execution cost

Disk Costs are Key

- Data stored on disk and then brought into main memory when needed for processing
- Disk block is unit of transfer (typically 4KB or 8KB)
- Cost of disk I/O dominates cost of typical DB operations
  - Disk access slower than memory access by factor of 10^5!

Unit 1 Topics

- File organizations
- Buffer management
- Redundant Arrays of Independent Disks (RAID)
- Storage Area Networks (SANs)

File Organizations

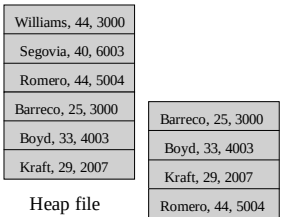
Putting data on the disk

Files

- File of records is the main data abstraction
  - Relation typically stored as a file
  - Stored as a collection of disk pages
  - Records identified by a unique **record id** or **rid**
  - Supports operations to create, destroy, open and close the file; Retrieve, add, delete and scan records in the file
- *File organization* is a method of arranging records in a file

Simple File Organizations

- Heap file
  - Records stored in random order
- Sorted file
  - Records physically ordered based on a key field value



Heap file

Sorted file

Indexes

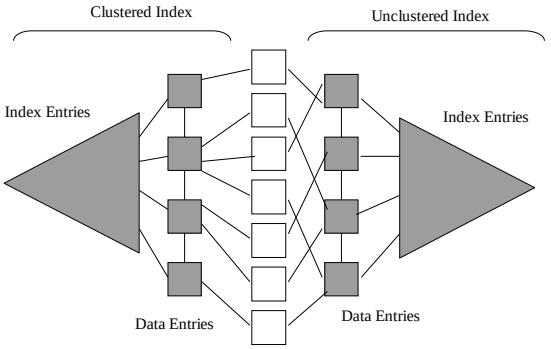
- **Index** is a data structure that organizes data records on disk to optimize certain types of retrieval operations
  - Can efficiently retrieve based on the **search key** of the index
- Consider employee file
  - Can store records organized as an index on age
  - Can create auxiliary index file based on salary

Indexes (Cont.)

- Index contains 2 types of entries – *index entries* and *data entries*
- 3 main alternatives for the data entries (records in the index file)
  - Data entry is an actual data record (with search key value *k*)
  - Data entry is a pair *<k, rid>*
  - Data entry is a pair *<k, rid-list>*

Indexes (Cont.)

- **Clustered versus unclustered**
  - Ordering of data records is same as, or close to, ordering of entries in index then the index is **clustered**, otherwise it is **unclustered**
- **Primary versus secondary**
  - Index on a set of fields that includes the primary key is a **primary** index
  - Other indexes called **secondary** indexes



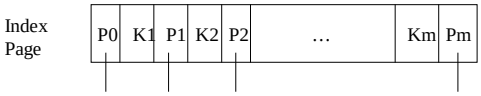
Tree-based Indexing

Tree Indexes

- Properties:
  - Efficient for range queries, including sorted file scans
  - Efficient insertion and deletion compared to sorted files
  - Efficient for equality queries (but not as good as hash index)

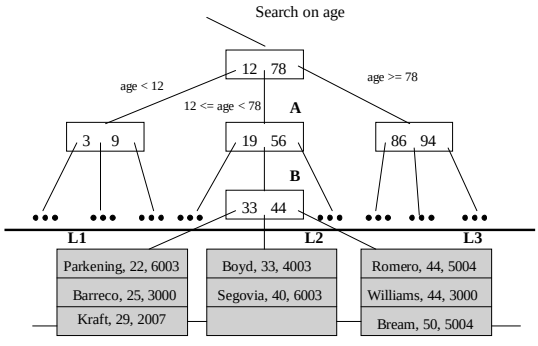
Tree Indexes (Cont.)

- Leaf pages contain data entries
- Non-leaf pages contain index entries of form *<search key value, page id>*



B+ Tree

- Most widely used type of index
- Dynamically adjusts to inserts and deletes to maintain a *balanced* tree
  - Insert and delete have cost $\log_F N$
- Minimum 50% occupancy (except root)
  - Each node contains *m* entries where $d \leq m \leq 2d$ where *d* is the **order** of the tree



B+ Tree – In Practice

- Typical order – 100; typical fill factor – 67%; average fanout – 133
- Typical capacities:
  - H = 4: 133^4 = 312,900,700 records
  - H = 3: 133^3 = 2,352,637
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8KB
  - Level 2 = 133 pages = 1 MB
  - Level 3 = 17,689 pages = 133 MB

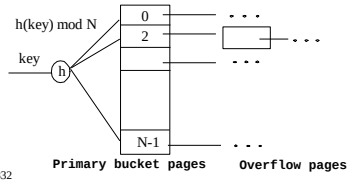
Hash-based Indexing

Introduction

- Hash-based indexes are best for equality selections. Cannot support range searches.
- Static and dynamic hashing techniques exist; trade-offs similar to ISAM vs. B+ trees.

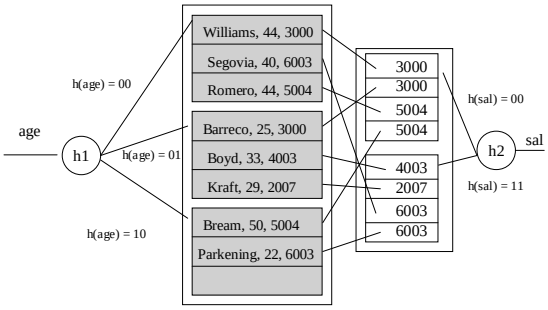
Static Hashing

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



Static Hashing (Cont.)

- Buckets contain *data entries*.
- Hash fn works on *search key* field of record *r*. Must distribute values over range 0 ... M-1.
  - **h**(*key*) = (a * *key* + b) usually works well.
  - a and b are constants; lots known about how to tune **h**.
- Long overflow chains can develop and degrade performance.
  - *Extendible Hashing*: Dynamic techniques to fix this problem.



Comparison of File Organizations

Organizations for Employee File

- Heap file
- File sorted on <age, sal>
- Clustered B+ tree file with search key <age, sal>
- Heap file with unclustered B+ tree index on <age, sal>
- Heap file with unclustered hash index on <age,sal>

Operations

- Scan: fetch all the records in the file
- Search with equality condition: eg find the employee with age = 23 and sal = 50
- Search with a range condition: eg find all employees with age > 35
- Insert a record

Assumptions

- *B* = number of data pages in file
- *R* = blocking factor of file
- *D* = data access time
- *F* = fan-out for tree indexes
- Clustered file usually 67% full => number of data pages 1.5*B*

Assumptions (Cont.)

- For unclustered index, data entry is $1/10^{th}$ size of data record => number of pages at leaf level is $0.1(1.5B) = 0.15B$
- For static hashing page occupancy around 80% => number pages to store data entries is $1.25(0.1B) = 0.125B$

Comparison

	Heap File	**Sorted File**	**Clustered B+ tree**	**Unclust B+ tree**	**Unclust Hash**
Scan	BD	BD	1.5*BD	D(0.15B + BR)	BD
Equality Search	0.5*BD	$D(\log_2 B)$	$D(\log_F(1.5*B))$	$D(\log_F(0.15B) + 1)$	2D
Range Search	BD	$D(\log_2 B$ + match)	$D(\log_F(1.5*B)$ +match)	$D(\log_F(0.15B)$ + match)	BD
Insert	2D	$D(\log_2 B$ + B)	$D(\log_F(1.5*B)$ + 1)	$D(\log_F(0.15B) + 3)$	4D