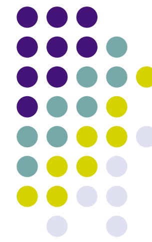
CMSC424: Database Design

Instructor: Amol Deshpande
amol@cs.umd.edu



Databases

- Data Models
  - Conceptual representation of the data
- Data Retrieval
  - How to ask questions of the database
  - How to answer those questions
- Data Storage
  - How/where to store data, how to access it
- Data Integrity
  - Manage crashes, concurrency
  - Manage semantic inconsistencies

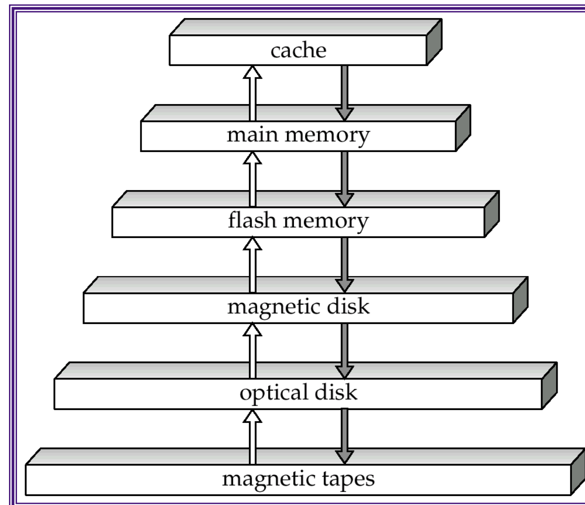
Outline

- Storage hierarchy
- Disks
- RAID
- File Organization
- Etc.…

Storage Hierarchy

- Tradeoffs between speed and cost of access
- Volatile vs nonvolatile
  - Volatile: Loses contents when power switched off
- Sequential vs random access
  - Sequential: read the data contiguously
  - Random: read the data from anywhere at any time

Storage Hierarchy



Storage Hierarchy

- Cache
  - Super fast; volatile
  - Typically on chip
  - L1 vs L2 vs L3 caches ???
    - Huge L3 caches available now-a-days
  - Becoming more and more important to care about this
    - Cache misses are expensive
  - Similar tradeoffs as were seen between main memory and disks
  - Cache-coherency ??

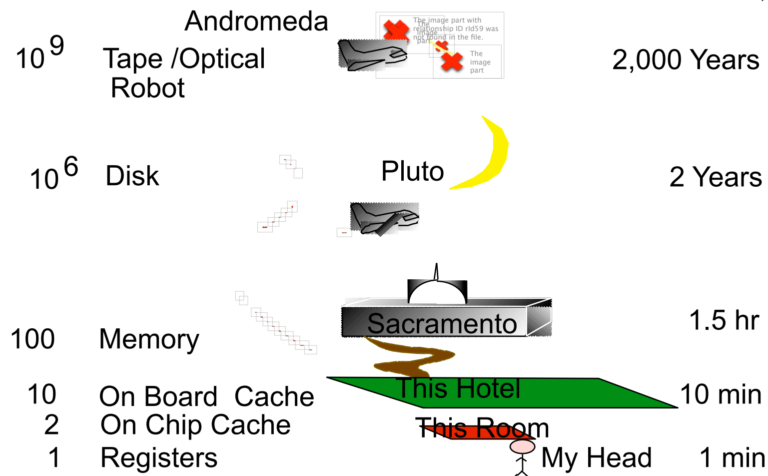
Storage Hierarchy

- Main memory
  - 10s or 100s of ns; volatile
  - Pretty cheap and dropping: 1GByte < 100$
  - Main memory databases feasible now-a-days
- Flash memory (EEPROM)
  - Limited number of write/erase cycles
  - Non-volatile, slower than main memory (especially writes)
  - Examples ?
- *Question*
  - *How does what we discuss next change if we use flash memory only ?*
  - *Key issue: Random access as cheap as sequential access*

Storage Hierarchy

- Magnetic Disk (Hard Drive)
  - Non-volatile
  - Sequential access much much faster than random access
  - Discuss in more detail later
- Optical Storage - CDs/DVDs; Jukeboxes
  - Used more as backups… Why ?
  - Very slow to write (if possible at all)
- Tape storage
  - Backups; super-cheap; painful to access
  - IBM just released a secure tape drive storage solution

Jim Gray's Storage Latency Analogy: How Far Away is the Data?



Storage…

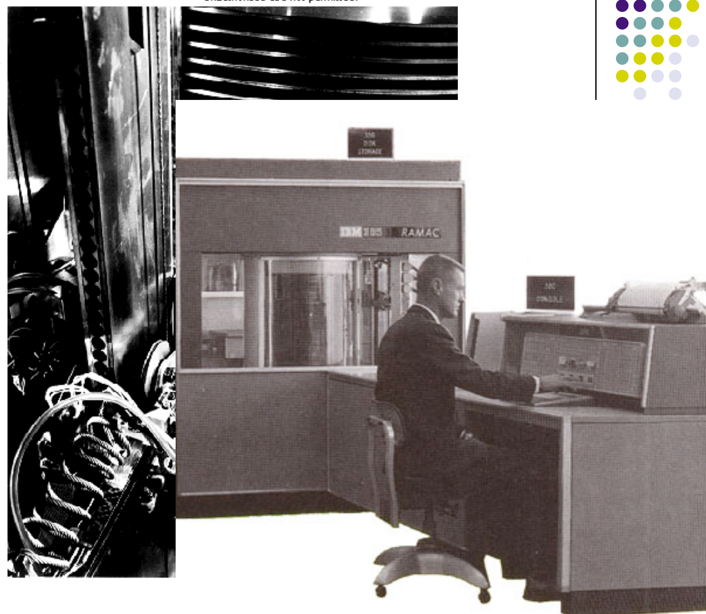
- Primary
  - e.g. Main memory, cache; typically volatile, fast
- Secondary
  - e.g. Disks; non-volatile
- Tertiary
  - e.g. Tapes; Non-volatile, super cheap, slow

Outline

- Storage hierarchy
- Disks
- RAID
- File Organization
- Etc.…

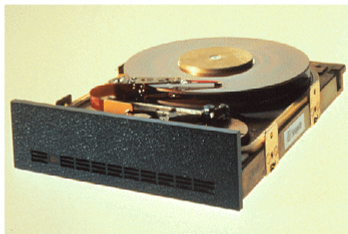
1956
IBM RAMAC
24” platters
100,000 characters each
5 million characters

From Computer Desktop Encyclopedia
Reproduced with permission.
© 1996 International Business Machines Corporation
Unauthorized use not permitted.



1979
SEAGATE
5MB

From Computer Desktop Encyclopedia
Reproduced with permission.
© 1998 Seagate Technologies

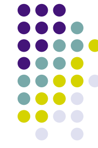


1998
SEAGATE
47GB

From Computer Desktop Encyclopedia
Reproduced with permission.
© 1998 Seagate Technologies

2004
Hitachi
400GB
Height (mm): 25.4. Width (mm): 101.6.
Depth (mm): 146. Weight (max. g): 700

2006
Western Digital
500GB
Weight (max. g): 600g



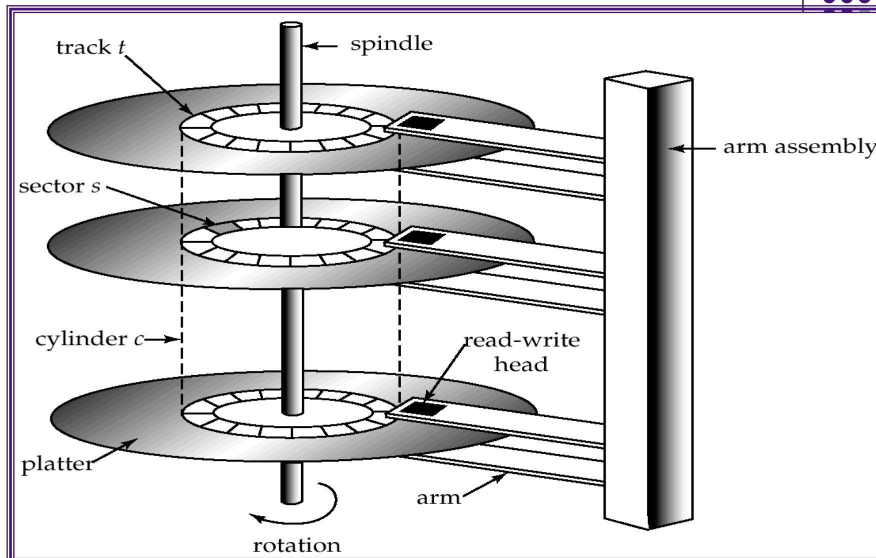
Latest:

Single hard drive:
- Seagate Barracuda 7200.10 SATA
- 750 GB
- 7200 rpm
- weight: 720g
- Uses “perpendicular recording”

Microdrives

IBM 1 GB

Toshiba 80GB



"Typical" Values

Diameter:	1 inch → 15 inches
Cylinders:	100 → 2000
Surfaces:	1 or 2
(Tracks/cyl)	2 (floppies) → 30
Sector Size:	512B → 50K
Capacity:	360 KB (old floppy) → 300 GB

Accessing Data

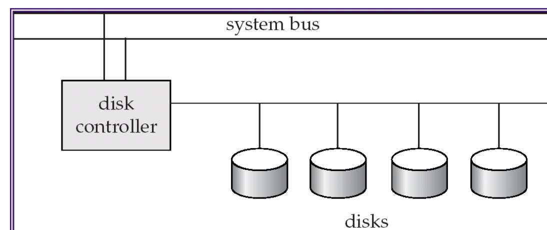
- Accessing a sector
  - Time to *seek* to the track (seek time)
    - average 4 to 10ms
  - \+ Waiting for the sector to get under the head (rotational latency)
    - average 4 to 11ms
  - \+ Time to transfer the data (transfer time)
    - very low
  - About 10ms per access
    - So if randomly accessed blocks, can only do 100 block transfers
    - 100 x 512bytes = 50 KB/s
- Data transfer rates
  - Rate at which data can be transferred (w/o any seeks)
  - 30-50MB/s (Compare to above)
    - Seeks are bad !

Reliability

- Mean time to/between failure (MTTF/MTBF):
  - 57 to 136 years
- Consider:
  - 1000 new disks
  - 1,200,000 hours of MTTF each
  - On average, one will fail 1200 hours = 50 days !

Disk Controller

- Interface between the disk and the CPU
- Accepts the commands
- *checksums* to verify correctness
- Remaps bad sectors

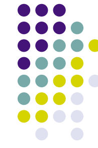


Optimizing block accesses

- Typically sectors too small
- Block: A contiguous sequence of sectors
  - 512 bytes to several Kbytes
  - All data transfers done in units of blocks
- Scheduling of block access requests ?
  - Considerations: *performance* and *fairness*
  - *Elevator algorithm*

Outline

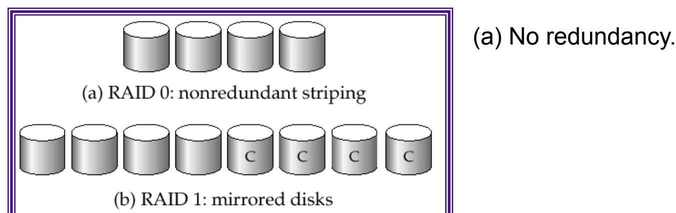
- Storage hierarchy
- Disks
- RAID
- File Organization
- Etc….



RAID

- Redundant array of independent disks
- Goal:
  - Disks are very cheap
  - Failures are very costly
  - Use “extra” disks to ensure reliability
    - If one disk goes down, the data still survives
  - Also allows faster access to data
- Many raid “levels”
  - Different reliability and performance properties

RAID Levels



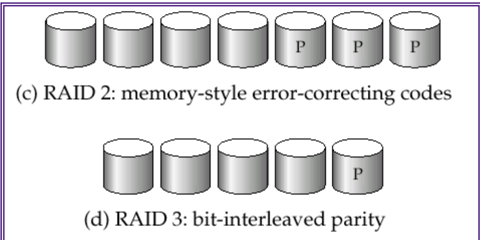
(a) RAID 0: nonredundant striping

(b) RAID 1: mirrored disks

(a) No redundancy.

(b) Make a copy of the disks.
If one disk goes down, we have a copy.
Reads: Can go to either disk, so higher data rate possible.
Writes: Need to write to both disks.

RAID Levels



(c) RAID 2: memory-style error-correcting codes

(d) RAID 3: bit-interleaved parity

(c) Memory-style Error Correcting

Keep extra bits around so we can reconstruct.

Superceeded by below.

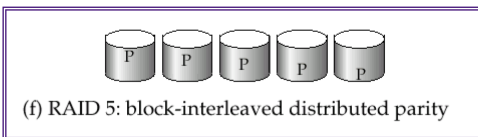
(d) One disk contains "parity" for the main data disks.

Can handle a single disk failure.

Little overhead (only 25% in the above case).

RAID Level 5

- Distributed parity "blocks" instead of bits
- Subsumes Level 4
- Normal operation:
  - "Read" directly from the disk. Uses all 5 disks
  - "Write": Need to read and update the parity block
    - To update 9 to 9'
      - read 9 and P2
      - compute P2' = P2 *xor* 9 *xor* 9'
      - write 9' and P2'

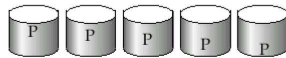


(f) RAID 5: block-interleaved distributed parity

P0	0	1	2	3
4	P1	5	6	7
8	9	P2	10	11
12	13	14	P3	15
16	17	18	19	P4

RAID Level 5

- Failure operation (disk 3 has failed)
  - "Read block 0": Read it directly from disk 2
  - "Read block 1" (which is on disk 3)
    - Read P0, 0, 2, 3 and compute 1 = P0 *xor* 0 *xor* 2 *xor* 3
  - "Write":
    - To update 9 to 9'
      - read 9 and P2
        - Oh… P2 is on disk 3
        - So no need to update it
      - Write 9'



(f) RAID 5: block-interleaved distributed parity

P0	0		2	3
4	P1		6	7
8	9		10	11
12	13		P3	15
16	17		19	P4

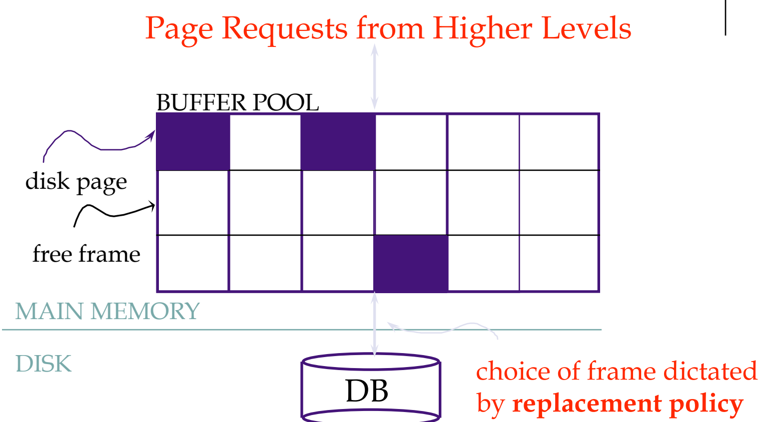
Choosing a RAID level

- Main choice between RAID 1 and RAID 5
- Level 1 better write performance than level 5
  - Level 5: 2 block reads and 2 block writes to write a single block
  - Level 1: only requires 2 block writes
  - Level 1 preferred for high update environments such as log disks
- Level 5 lower storage cost
  - Level 1 60% more disks
  - Level 5 is preferred for applications with low update rate, and large amounts of data

Outline

- Storage hierarchy
- Disks
- RAID
- Buffer Manager
- File Organization
- Etc.…

Buffer Manager



- *Data must be in RAM for DBMS to operate on it!*
- *Buffer Mgr hides the fact that not all data is in RAM*

Buffer Manager

- Similar to *virtual memory manager*
- Buffer replacement policies
  - What page to evict ?
  - LRU: Least Recently Used
    - Throw out the page that was not used in a long time
  - MRU: Most Recently Used
    - The opposite
    - Why ?
  - Clock ?
    - An efficient implementation of LRU

Buffer Manager

- *Pinning* a block
  - Not allowed to write back to the disk
- *Force-output (force-write)*
  - Force the contents of a block to be written to disk
- *Order the writes*
  - This block must be written to disk before this block
- Critical for fault tolerant guarantees
  - Otherwise the database has no control over whats on disk and whats not on disk

Outline

- Storage hierarchy
- Disks
- RAID
- Buffer Manager
- File Organization
- Etc.…

File Organization

- How are the relations mapped to the disk blocks ?
  - Use a standard file system ?
    - High-end systems have their own OS/file systems
    - OS interferes more than helps in many cases
  - Mapping of relations to file ?
    - One-to-one ?
    - Advantages in storing multiple relations clustered together
  - A *file* is essentially a *collection of disk blocks*
    - How are the tuples mapped to the disk blocks ?
    - How are they stored within each block

File Organization

- Goals:
  - Allow insertion/deletions of tuples/records
  - Fetch a particular record (specified by record id)
  - Find all tuples that match a condition (say SSN = 123) ?
- Simplest case
  - Each relation is mapped to a file
  - A file contains a sequence of records
  - Each record corresponds to a logical tuple
- Next:
  - How are tuples/records stored within a block ?

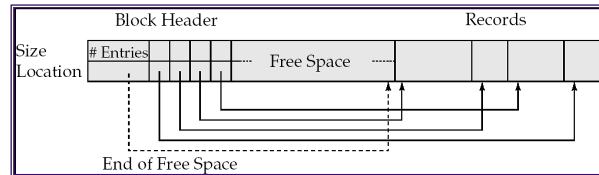
Fixed Length Records

- n = number of bytes per record
- Store record *i* at position:
  - n * (i – 1)
- Records may cross blocks
  - Not desirable
  - Stagger so that that doesn't happen
- Inserting a tuple ?
  - Depends on the policy used
  - One option: Simply append at the end of the record
- Deletions ?
  - Option 1: Rearrange
  - Option 2: Keep a *free list* and use for next insert

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Variable-length Records

Slotted page structure



- *Indirection:*
  - The records may move inside the page, but the outside world is oblivious to it
  - Why ?
    - The headers are used as a indirection mechanism
    - *Record ID 1000 is the 5th entry in the page number X*

File Organization

- Which block of a file should a record go to ?
  - Anywhere ?
    - How to search for "SSN = 123" ?
    - Called "heap" organization
  - Sorted by SSN ?
    - Called "sequential" organization
    - Keeping it sorted would be painful
    - How would you search ?
  - Based on a "hash" key
    - Called "hashing" organization
    - Store the record with SSN = x in the block number x%1000
    - Why ?

Sequential File Organization

- Keep sorted by some search key
- Insertion
  - Find the block in which the tuple should be
  - If there is free space, insert it
  - Otherwise, must create overflow pages
- Deletions
  - Delete and keep the free space
  - Databases tend to be insert heavy, so free space gets used fast
- Can become *fragmented*
  - Must reorganize once in a while

Sequential File Organization

- What if I want to find a particular record by value ?
  - *Account info for SSN = 123*
- Binary search
  - Takes *log(n)* number of disk accesses
    - Random accesses
  - Too much
    - n = 1,000,000,000 -- log(n) = 30
    - Recall each random access approx 10 ms
    - 300 ms to find just one account information
    - < 4 requests satisfied per second

Outline

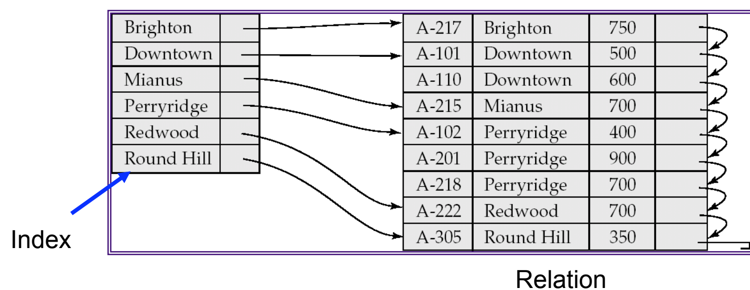
- Storage hierarchy
- Disks
- RAID
- Buffer Manager
- File Organization
- Indexes
- Etc…

Index

- A data structure for efficient search through large databaess
- Two key ideas:
  - The records are mapped to the disk blocks in specific ways
    - Sorted, or hash-based
  - Auxiliary data structures are maintained that allow quick search
- Think library index/catalogue
- Search key:
  - Attribute or set of attributes used to look up records
  - E.g. SSN for a persons table
- Two types of indexes
  - Ordered indexes
  - Hash-based indexes

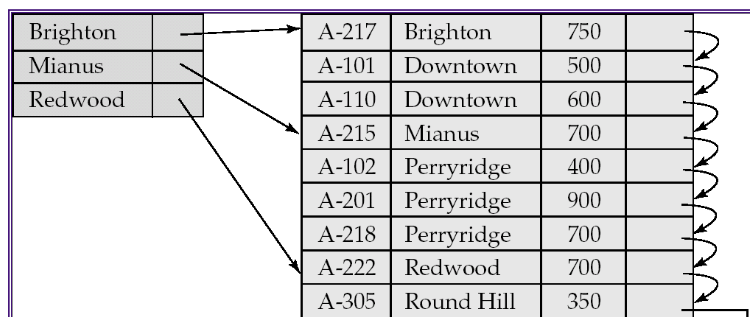
Ordered Indexes

- Primary index
  - The relation is sorted on the search key of the index
- Secondary index
  - It is not
- Can have only one primary index on a relation



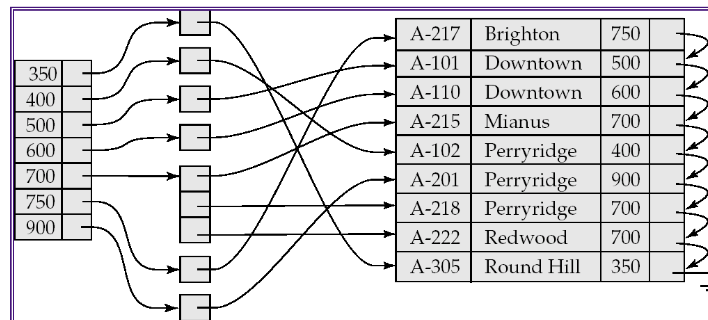
Primary *Sparse* Index

- Every key doesn't have to appear in the index
- Allows for very small indexes
  - Better chance of fitting in memory
  - Tradeoff: Must access the relation file even if the record is not present



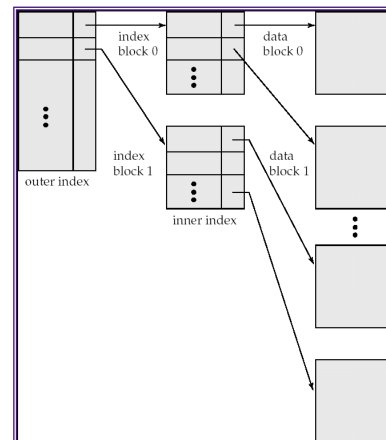
Secondary Index

- Relation sorted on *branch*
- But we want an index on *balance*
- Must be dense
  - Every search key must appear in the index



Multi-level Indexes

- What if the index itself is too big for memory ?
- Relation size = n = 1,000,000,000
- Block size = 100 tuples per block
- So, number of pages = 10,000,000
- Keeping one entry per page takes too much space
- Solution
  - Build an index on the index itself



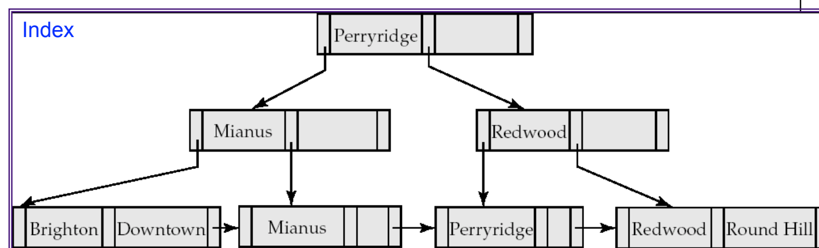
Multi-level Indexes

- How do you search through a multi-level index ?
- What about keeping the index up-to-date ?
  - Tuple insertions and deletions
    - This is a static structure
    - Need overflow pages to deal with insertions
  - Works well if no inserts/deletes
  - Not so good when inserts and deletes are common

Outline

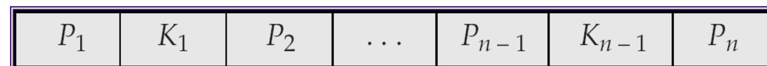
- Storage hierarchy
- Disks
- RAID
- Buffer Manager
- File Organization
- Indexes
- B+-Tree Indexes
- Etc..

Example B+-Tree Index



B^+-Tree Node Structure

- Typical node



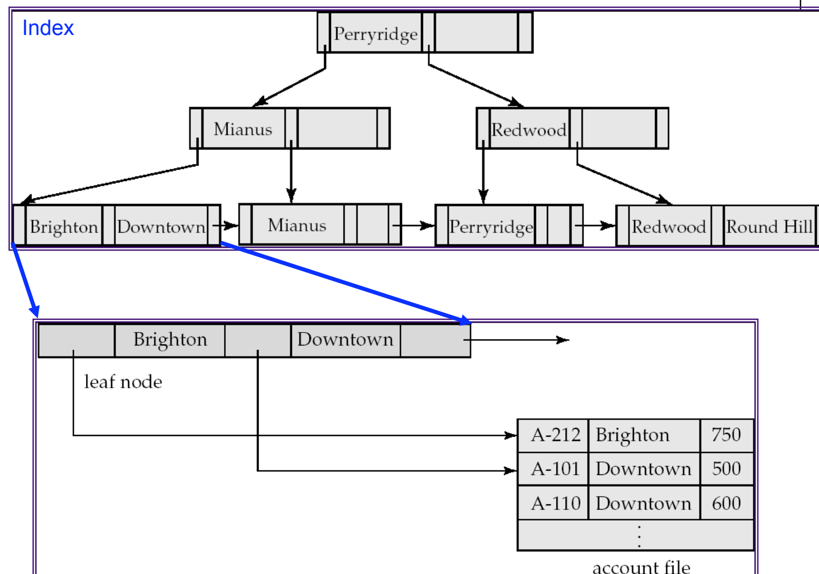
  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

Properties of B+-Trees

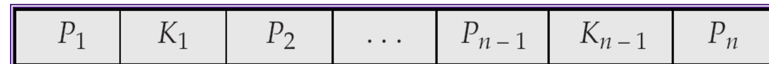
- It is balanced
  - Every path from the root to a leaf is same length
- Leaf nodes (at the bottom)
  - *P1* contains the pointers to tuple(s) with key *K1*
  - …
  - *Pn* is a pointer to the *next* leaf node
  - Must contain at least n/2 entries

Example B+-Tree Index



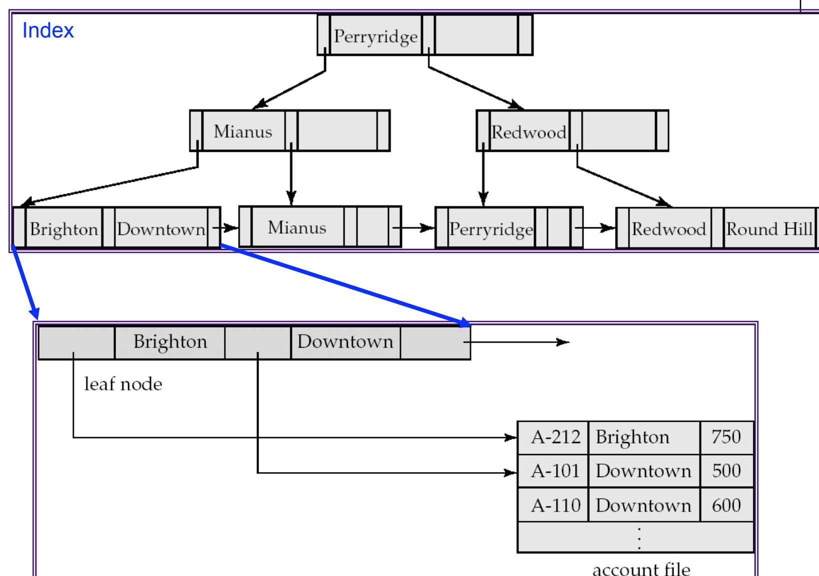
Properties

- Interior nodes



  - All tuples in the subtree pointed to by *P1,* have search key < *K1*
  - To find a tuple with key *K1'* < *K1,* follow *P1*
  - …
  - Finally, search keys in the tuples contained in the subtree pointed to by *Pn,* are all larger than *Kn-1*
  - Must contain at least n/2 entries (unless root)

Example B+-Tree Index



B+-Trees - Searching

- How to search ?
  - Follow the pointers
- Logarithmic
  - $log_{B/2}(N)$, where *B = Number of entries per block*
  - *B* is also called the order of the B+-Tree Index
    - Typically 100 or so
- If a relation contains1,000,000,000 entries, takes only 4 random accesses
- The top levels are typically in memory
  - So only requires 1 or 2 random accesses per request

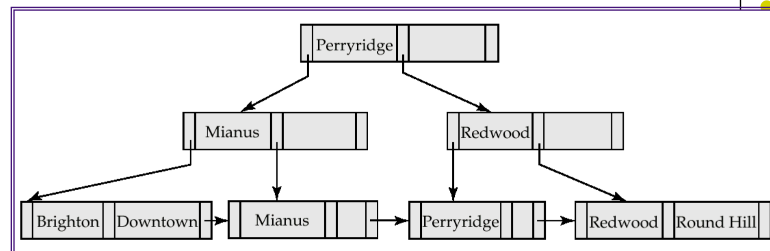
Tuple Insertion

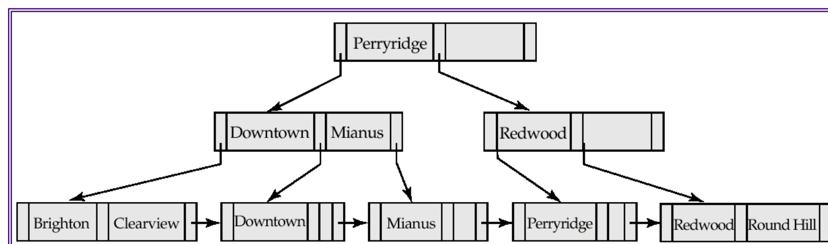
- Find the leaf node where the search key should go
- If already present
  - Insert record in the file. Update the bucket if necessary
    - This would be needed for secondary indexes
- If not present
  - Insert the record in the file
  - Adjust the index
    - Add a new *(Ki, Pi)* pair to the leaf node
    - Recall the keys in the nodes are sorted
  - What if there is no space ?

Tuple Insertion

- Splitting a node
  - Node has too many key-pointer pairs
    - Needs to store *n,* only has space for *n-1*
  - Split the node into two nodes
    - Put about half in each
  - Recursively go up the tree
    - May result in splitting all the way to the root
    - In fact, may end up adding a *level* to the tree
  - Pseudocode in the book !!

B+-Trees: Insertion



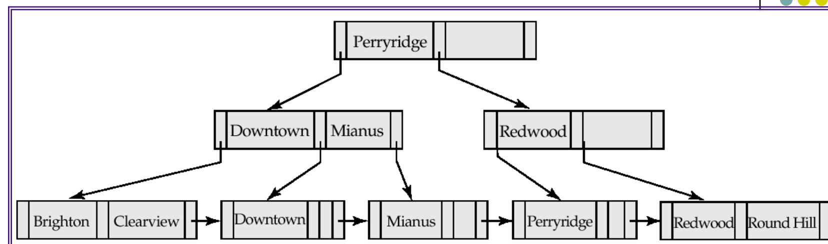


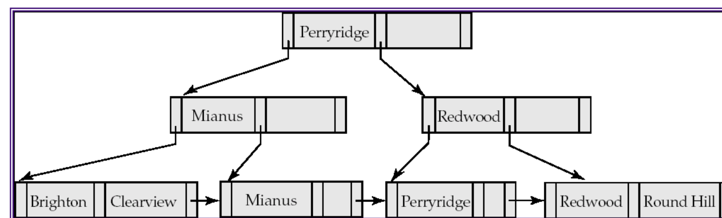
B+-Tree before and after insertion of “Clearview”

Updates on B+-Trees: Deletion

- Find the record, delete it.
- Remove the corresponding (search-key, pointer) pair from a leaf node
  - Note that there might be another tuple with the same search-key
  - In that case, this is not needed
- Issue:
  - The leaf node now may contain too few entries
    - Why do we care ?
  - Solution:
    1. See if you can borrow some entries from a sibling
    2. If all the siblings are also just barely full, then *merge (opposite of split)*
  - May end up merging all the way to the root
  - In fact, may reduce the height of the tree by one

Examples of B+-Tree Deletion

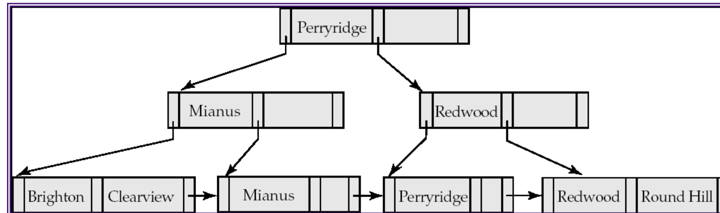


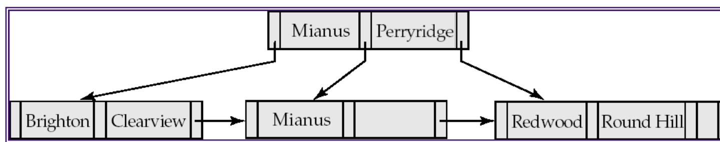


Before and after deleting "Downtown"

- Deleting "Downtown" causes merging of under-full leaves
  - leaf node can become empty only for n=3!

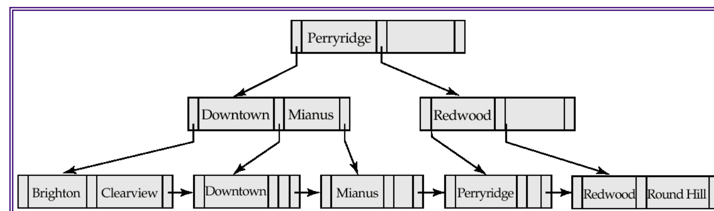
Examples of B+-Tree Deletion

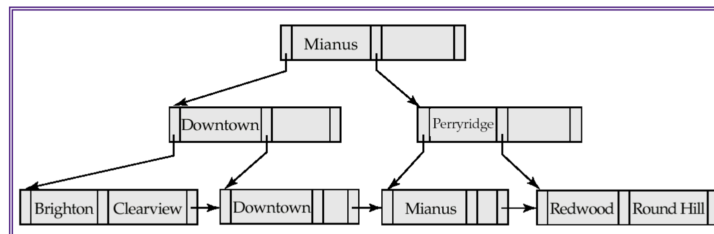




Deletion of “Perryridge” from result of previous example

Example of B+-tree Deletion





Before and after deletion of “Perryridge” from earlier example

B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 3: 133^3 = 2,352,637 entries
  - Height 4: 133^4 = 312,900,700 entries
- 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

B+ Trees: Summary

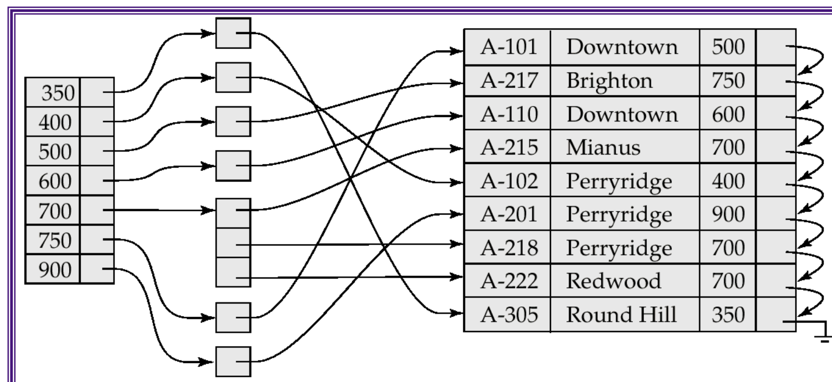
- Searching:
  - $\log_d(n)$ – Where *d* is the order, and *n* is the number of entries
- Insertion:
  - Find the leaf to insert into
  - If full, split the node, and adjust index accordingly
  - Similar cost as searching
- Deletion
  - Find the leaf node
  - Delete
  - May not remain half-full; must adjust the index accordingly

More…

- Primary vs Secondary Indexes
- More B+-Trees
- Hash-based Indexes
  - Static Hashing
  - Extendible Hashing
  - Linear Hashing
- Grid-files
- R-Trees
- etc…

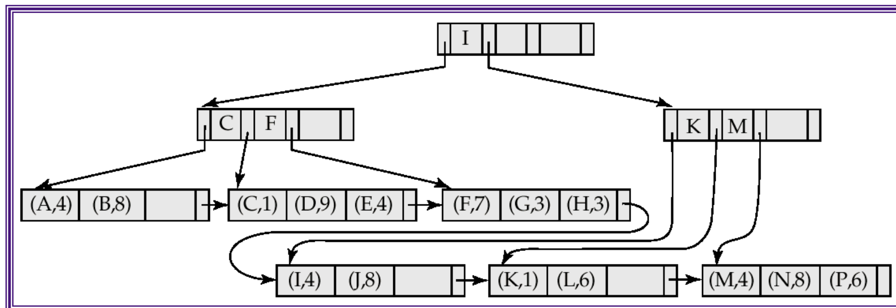
Secondary Index

- If relation not sorted by search key, called a *secondary index*
  - Not all tuples with the same search key will be together
  - Searching is more expensive



B+-Tree File Organization

- Store the records at the leaves
- Sorted order etc..



B-Tree

- Predates
- Different treatment of search keys
- Less storage
- Significantly harder to implement
- Not used.

Hash-based File Organization

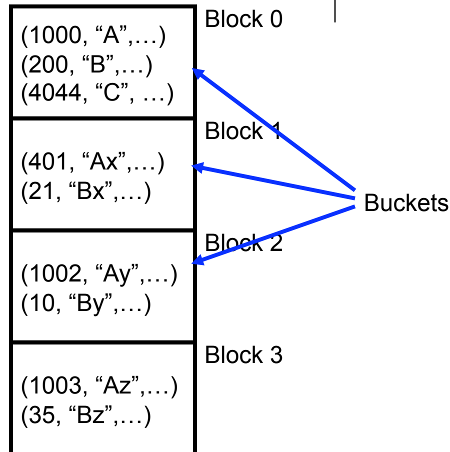
Store record with search key *k* in block number *h(k)*

e.g. for a person file,
h(SSN) = SSN % 4

Blocks called "*buckets*"

What if the block becomes full ?
Overflow pages

Uniformity property:
Don't want all tuples to map to the same bucket
h(SSN) = SSN % 2 would be bad



Hash-based File Organization

Hashed on "branch-name"

Hash function:

a = 1, b = 2, .., z = 26

h(abz)

= (1 + 2 + 26) % 10

= 9

bucket 0

bucket 1

bucket 2

bucket 3

A-217	Brighton	750
A-305	Round Hill	350

bucket 4

A-222	Redwood	700

bucket 5

A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700

bucket 6

bucket 7

A-215	Mianus	700

bucket 8

A-101	Downtown	500
A-110	Downtown	600

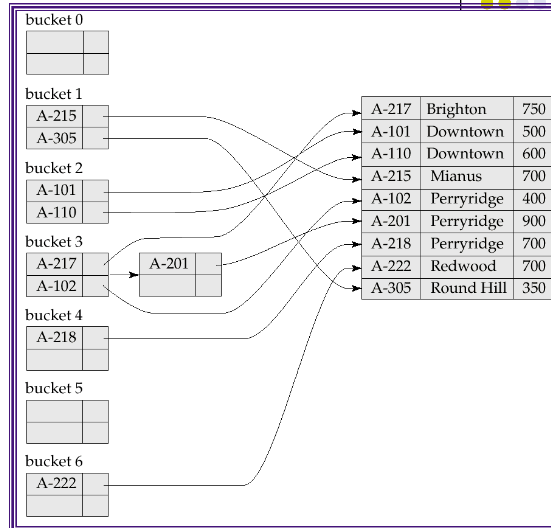
bucket 9

Hash Indexes

Extends the basic idea

Search:
Find the block with search key
Follow the pointer

Range search ?
a < X < b ?



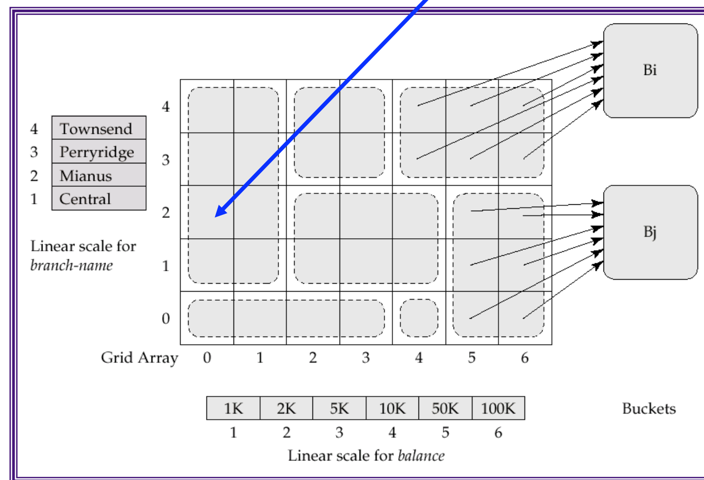
Hash Indexes

- Very fast search on equality
- Can’t search for “ranges” at all
  - Must scan the file
- Inserts/Deletes
  - Overflow pages can degrade the performance
- Two approaches
  - Dynamic hashing
  - Extendible hashing

Grid Files

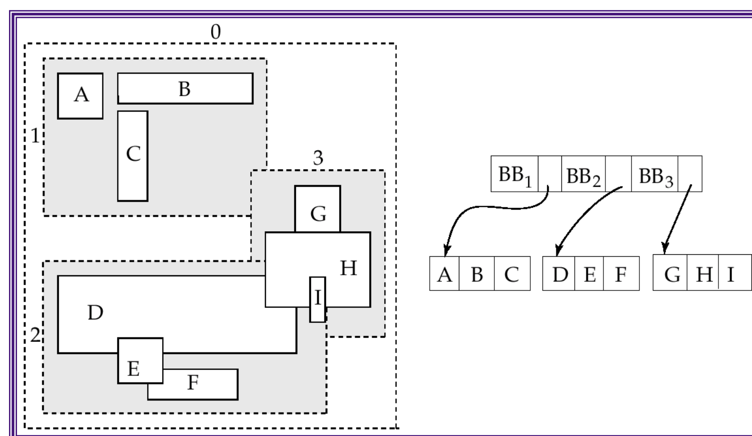
Multidimensional index structure
Can handle: *X = x1 and Y = y1*
a < X < b and c < Y < d

Stores pointers to tuples with :
branch-name between Mianus and Perryridge
and balance < 1k



R-Trees

For spatial data (e.g. maps, rectangles, GPS data etc)



Conclusions

- Indexing Goal: "Quickly find the tuples that match certain conditions"
- Equality and range queries most common
  - Hence B+-Trees the predominant structure for on-disk representation
  - Hashing is used more commonly for in-memory operations
- Many many more types of indexing structures exist
  - For different types of data
  - For different types of queries
    - E.g. "nearest-neighbor" queries