

COMS W4111 - Introduction to Databases

(Module II)

DBMS Architecture and Implementation: Disks, I/O, Indexes

Donald F. Ferguson (dff@cs.columbia.edu)

Contents

Contents

- Reminder of DBMS implementation and Module II.
- Disks and Data Input/Output

Module II – DBMS Architecture and Implementation

Overview and Reminder

Database Management System Reminder

What is a database? In essence a database is nothing more than a collection of information that exists over a long period of time, often many years. In common parlance, the term *database* refers to a collection of data that is managed by a DBMS. The DBMS is expected to:

- 
1. Allow users to create new databases and specify their *schemas* (logical structure of the data), using a specialized *data-definition language*.

Covered for the relational model.

Database Systems: The Complete Book (2nd Edition)

by [Hector Garcia-Molina](#) (Author), [Jeffrey D. Ullman](#) (Author), [Jennifer Widom](#) (Author)

Database Management System Reminder

- 
- 
2. Give users the ability to *query* the data (a “query” is database lingo for a question about the data) and modify the data, using an appropriate language, often called a *query language* or *data-manipulation language*.
 3. Support the storage of very large amounts of data — many terabytes or more — over a long period of time, allowing efficient access to the data for queries and database modifications.
 4. Enable *durability*, the recovery of the database in the face of failures, errors of many kinds, or intentional misuse.
 5. Control access to data from many users at once, without allowing unexpected interactions among users (called *isolation*) and without actions on the data to be performed partially but not completely (called *atomicity*).

Focus for next part of course.

Database Systems: The Complete Book (2nd Edition)

by [Hector Garcia-Molina](#) (Author), [Jeffrey D. Ullman](#) (Author), [Jennifer Widom](#) (Author)



Purpose of Database Systems

In the early days, database applications were built directly on top of file systems, which leads to:

- Data redundancy and inconsistency: data is stored in multiple file formats resulting in duplication of information in different files
- Difficulty in accessing data
 - Need to write a new program to carry out each new task
- Data isolation
 - Multiple files and formats
- Integrity problems
 - Integrity constraints (e.g., account balance > 0) become “buried” in program code rather than being stated explicitly
 - Hard to add new constraints or change existing ones



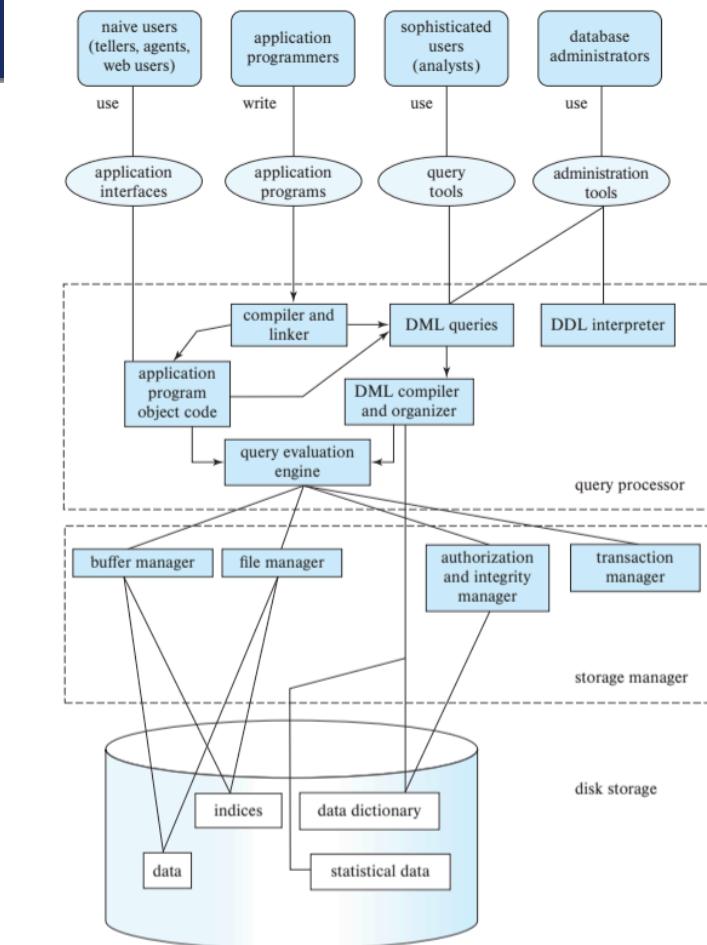
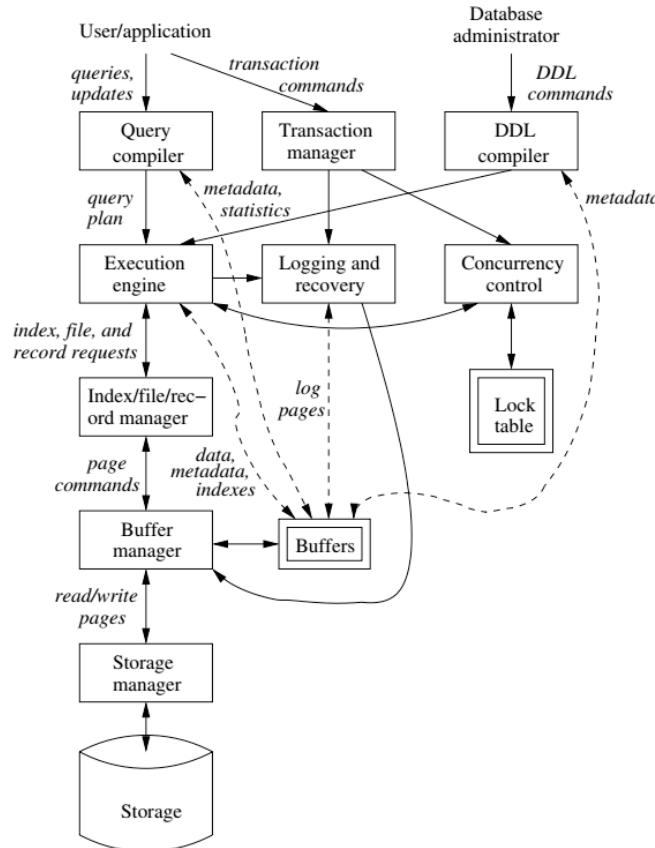
Purpose of Database Systems (Cont.)

- Atomicity of updates
 - Failures may leave database in an inconsistent state with partial updates carried out
 - Example: Transfer of funds from one account to another should either complete or not happen at all
- Concurrent access by multiple users
 - Concurrent access needed for performance
 - Uncontrolled concurrent accesses can lead to inconsistencies
 - Ex: Two people reading a balance (say 100) and updating it by withdrawing money (say 50 each) at the same time
- Security problems
 - Hard to provide user access to some, but not all, data

Database systems offer solutions to all the above problems

In Module I, we explored how users interact with the (some) of the functions through DDL and DML. In this module, we will explore *how* DBMS implement the capabilities “under the covers.”

DBMS Subsystems



Disks, I/O, Indexes

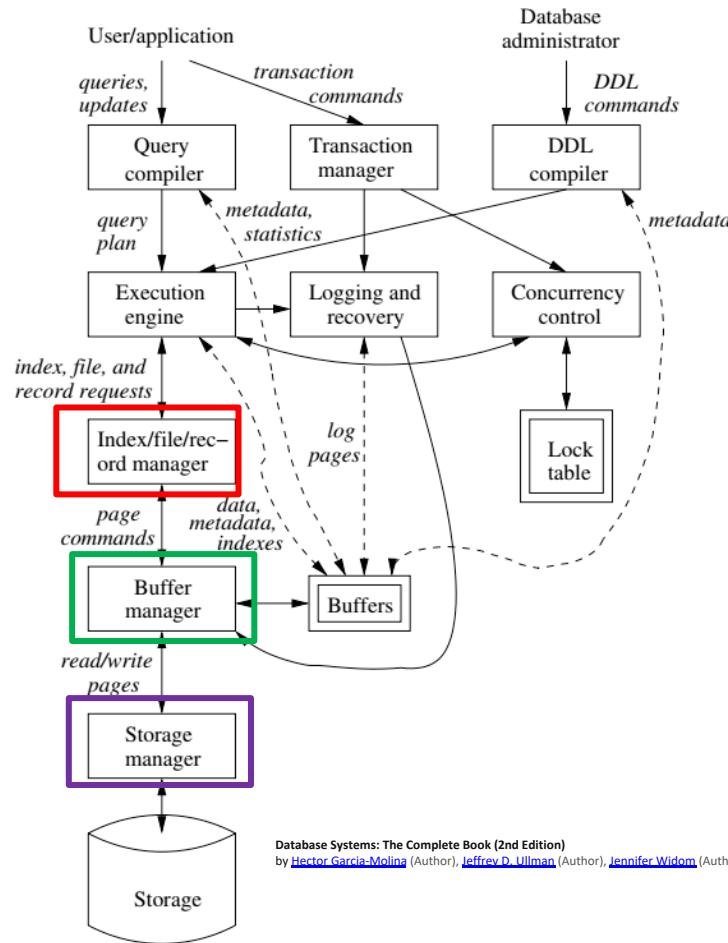
(Database Systems Concepts, V7, Ch. 12)

Disks as Far as the Eye Can See



Data Management

- Find things quickly.
- Access things quickly.
- Load/save things quickly.



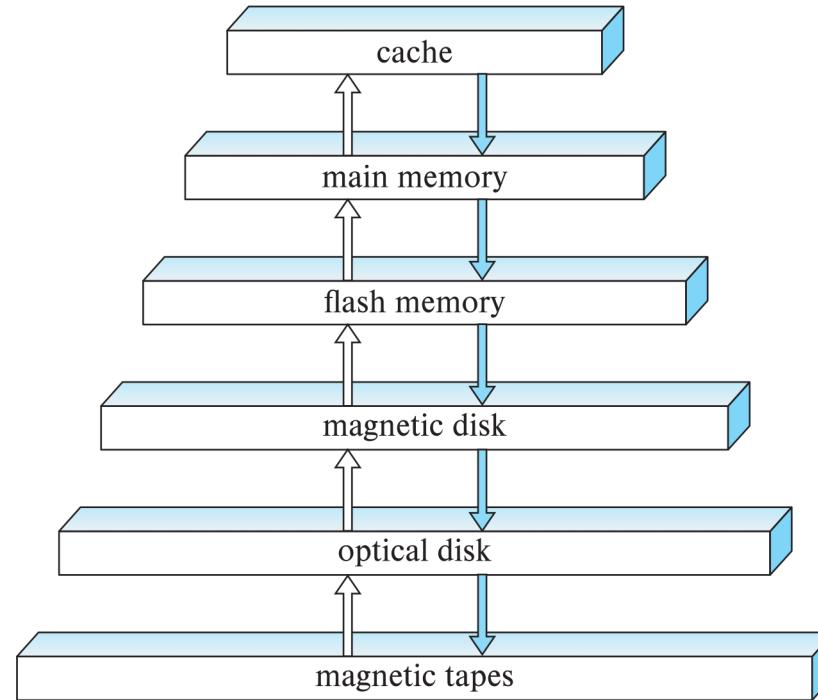


Classification of Physical Storage Media

- Can differentiate storage into:
 - **volatile storage:** loses contents when power is switched off
 - **non-volatile storage:**
 - Contents persist even when power is switched off.
 - Includes secondary and tertiary storage, as well as battery-backed up main-memory.
- Factors affecting choice of storage media include
 - Speed with which data can be accessed
 - Cost per unit of data
 - Reliability



Storage Hierarchy



Memory Hierarchy

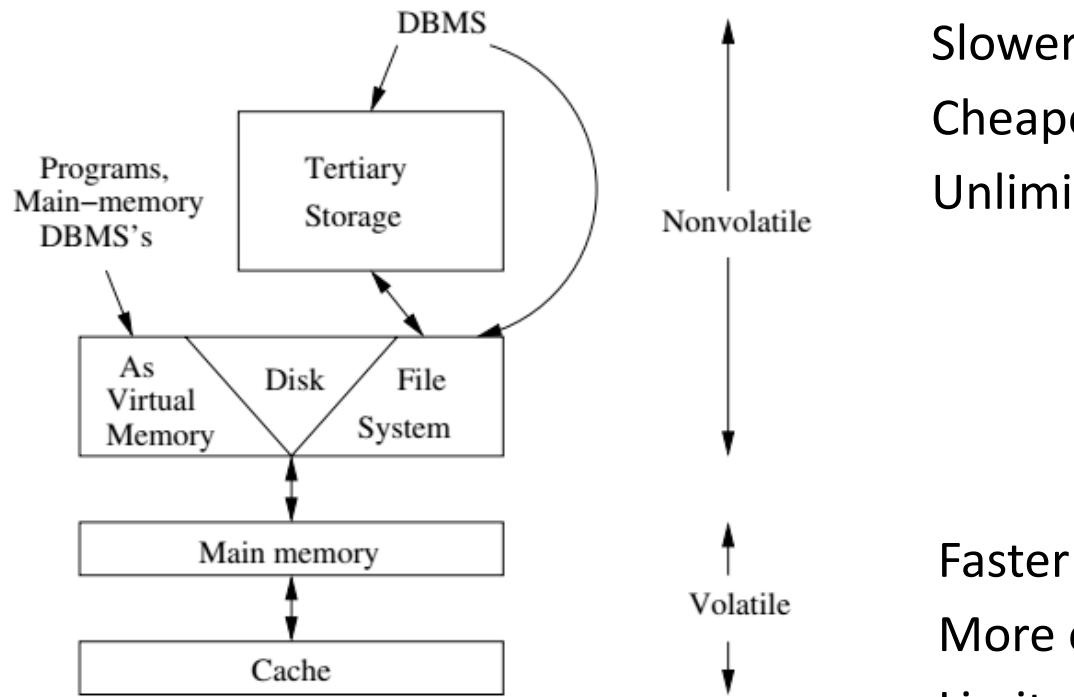


Figure 13.1: The memory hierarchy

Memory Hierarchy (Very Old Numbers – Still Directionally Valid)

Storage Technology

Price, Performance & Capacity

Technologies	Capacity (GB)	Latency (microS)	IOPs	Cost/IOPS (\$)	Cost/GB (\$)
Cloud Storage	Unlimited	60,000	20	17c/GB	0.15/month
Capacity HDDs	2,500	12,000	250	1.67	0.15
Performance HDDs	300	7,000	500	1.52	1.30
SSDs (write)	64	300	5000	0.20	13
SSDs (read only)	64	45	30,000	0.03	13
DRAM	8	0.005	500,000	0.001	52

- These numbers are ancient.
- Looking for more modern numbers.
- But, does give an idea of
 - Price
 - Performance
- The general observation is that
 - Performance goes up 10X/level.
 - Price goes up 10x per level.
- Note: One major change is improved price performance of SSD relative to HDD for large data.



Storage Hierarchy (Cont.)

- **primary storage:** Fastest media but volatile (cache, main memory).
- **secondary storage:** next level in hierarchy, non-volatile, moderately fast access time
 - Also called **on-line storage**
 - E.g., flash memory, magnetic disks
- **tertiary storage:** lowest level in hierarchy, non-volatile, slow access time
 - also called **off-line storage** and used for **archival storage**
 - e.g., magnetic tape, optical storage
 - Magnetic tape
 - Sequential access, 1 to 12 TB capacity
 - A few drives with many tapes
 - Juke boxes with petabytes (1000's of TB) of storage

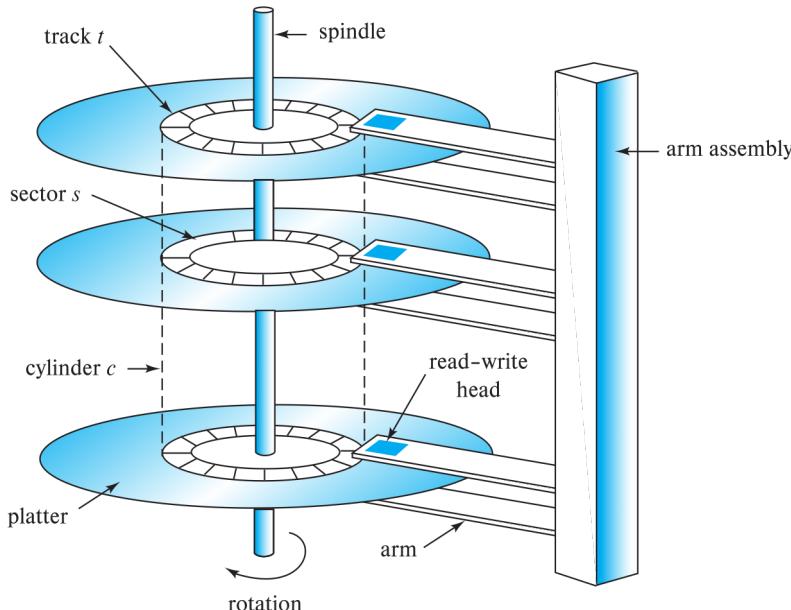


Storage Interfaces

- Disk interface standards families
 - SATA (Serial ATA)
 - SATA 3 supports data transfer speeds of up to 6 gigabits/sec
 - SAS (Serial Attached SCSI)
 - SAS Version 3 supports 12 gigabits/sec
 - NVMe (Non-Volatile Memory Express) interface
 - Works with PCIe connectors to support lower latency and higher transfer rates
 - Supports data transfer rates of up to 24 gigabits/sec
- Disks usually connected directly to computer system
- In **Storage Area Networks (SAN)**, a large number of disks are connected by a high-speed network to a number of servers
- In **Network Attached Storage (NAS)** networked storage provides a file system interface using networked file system protocol, instead of providing a disk system interface



Magnetic Hard Disk Mechanism

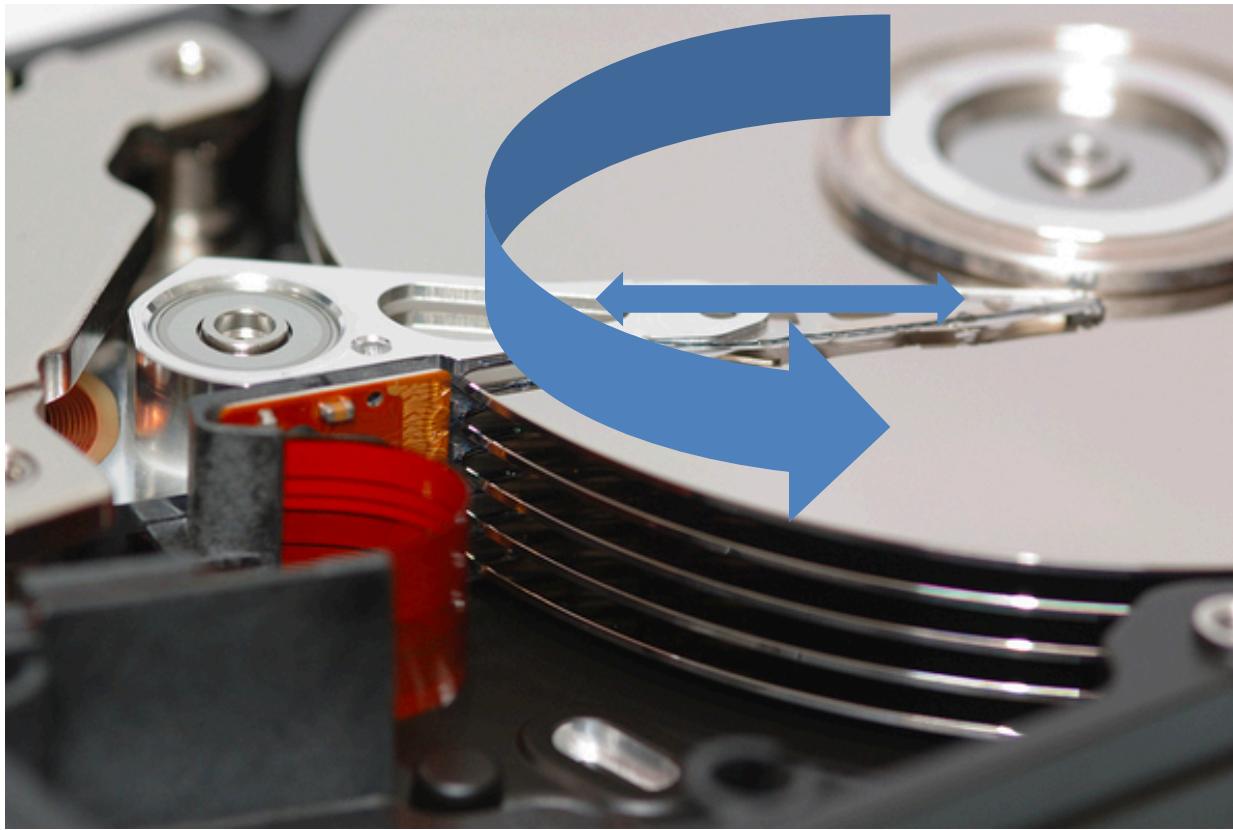


Schematic diagram of magnetic disk drive



Photo of magnetic disk drive

Hard Disk Drive



Disk Configuration

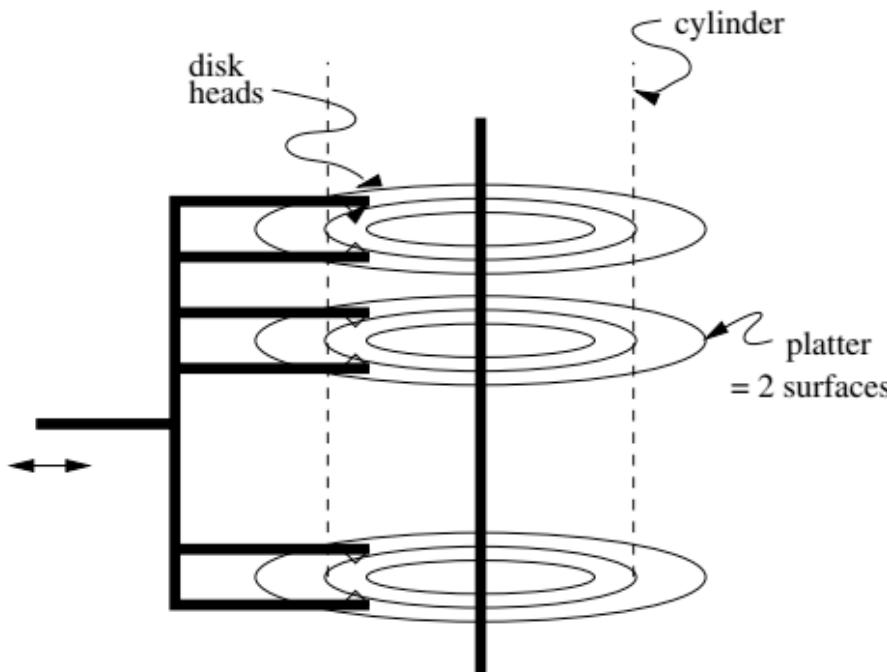


Figure 13.2: A typical disk

Components of disk I/O delay

Seek: Move head to cylinder/track.

Rotation: Wait for sector to get under head

Transfer: Move data from disk to memory.

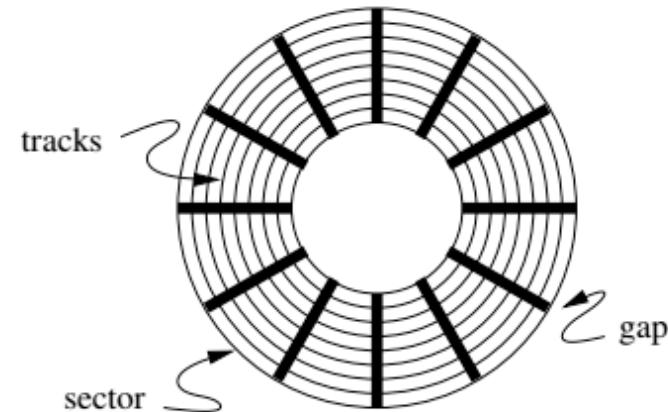
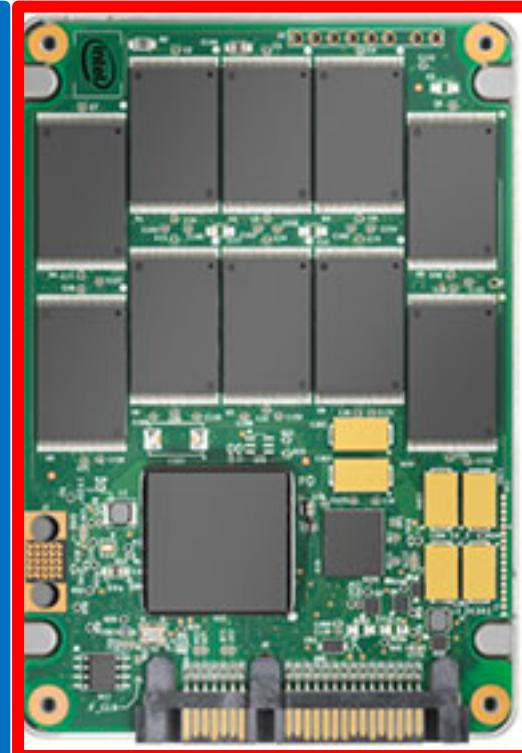


Figure 13.3: Top view of a disk surface

Database Systems: The Complete Book (2nd Edition)
by [Hector Garcia-Molina](#) (Author), [Jeffrey D. Ullman](#) (Author), [Jennifer Widom](#) (Author)

Hard Disk versus Solid State Disk

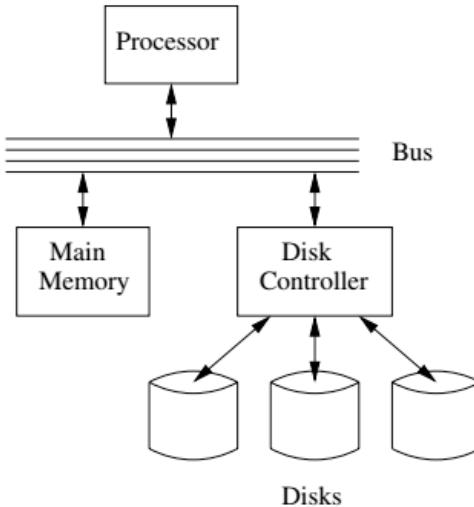
Hard
Disk
Drive



Solid
State
Drive



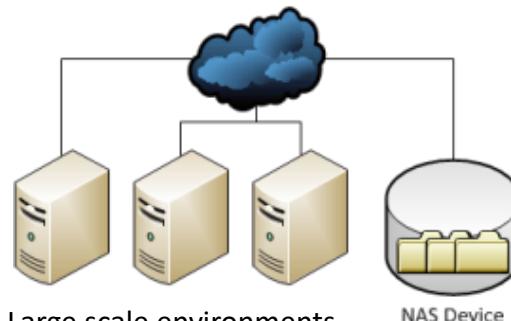
I/O Architecture



How we normally think of disks and I/O.

Network Attached Storage

- Shared storage over shared network
- File system
- Easier management

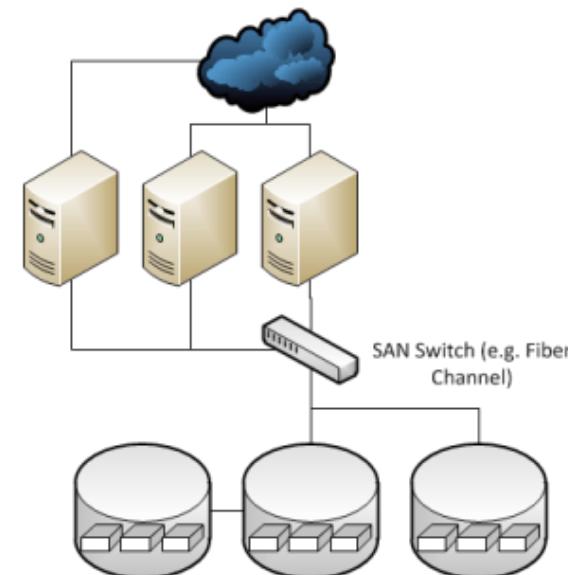


Large scale environments

- The bus-controller connection is over some kind of network.
- The disk controller is at the disks, and basically a “computer” with SW.
- Network is either
 - Standard communication network, or
 - Highly optimized I/O network.

Storage Area Network

- Shared storage over dedicated network
- Raw storage
- Fast, but costly





Magnetic Disks

- **Read-write head**
- Surface of platter divided into circular **tracks**
 - Over 50K-100K tracks per platter on typical hard disks
- Each track is divided into **sectors**.
 - A sector is the smallest unit of data that can be read or written.
 - Sector size typically 512 bytes
 - Typical sectors per track: 500 to 1000 (on inner tracks) to 1000 to 2000 (on outer tracks)
- To read/write a sector
 - disk arm swings to position head on right track
 - platter spins continually; data is read/written as sector passes under head
- Head-disk assemblies
 - multiple disk platters on a single spindle (1 to 5 usually)
 - one head per platter, mounted on a common arm.
- **Cylinder i** consists of i^{th} track of all the platters



Magnetic Disks (Cont.)

- **Disk controller** – interfaces between the computer system and the disk drive hardware.
 - accepts high-level commands to read or write a sector
 - initiates actions such as moving the disk arm to the right track and actually reading or writing the data
 - Computes and attaches **checksums** to each sector to verify that data is read back correctly
 - If data is corrupted, with very high probability stored checksum won't match recomputed checksum
 - Ensures successful writing by reading back sector after writing it
 - Performs **remapping of bad sectors**



Performance Measures of Disks

- **Access time** – the time it takes from when a read or write request is issued to when data transfer begins. Consists of:
 - **Seek time** – time it takes to reposition the arm over the correct track.
 - Average seek time is 1/2 the worst case seek time.
 - Would be 1/3 if all tracks had the same number of sectors, and we ignore the time to start and stop arm movement
 - 4 to 10 milliseconds on typical disks
 - **Rotational latency** – time it takes for the sector to be accessed to appear under the head.
 - 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)
 - Average latency is 1/2 of the above latency.
 - Overall latency is 5 to 20 msec depending on disk model
- **Data-transfer rate** – the rate at which data can be retrieved from or stored to the disk.
 - 25 to 200 MB per second max rate, lower for inner tracks



Performance Measures (Cont.)

- **Disk block** is a logical unit for storage allocation and retrieval
 - 4 to 16 kilobytes typically
 - Smaller blocks: more transfers from disk
 - Larger blocks: more space wasted due to partially filled blocks
- **Sequential access pattern**
 - Successive requests are for successive disk blocks
 - Disk seek required only for first block
- **Random access pattern**
 - Successive requests are for blocks that can be anywhere on disk
 - Each access requires a seek
 - Transfer rates are low since a lot of time is wasted in seeks
- **I/O operations per second (IOPS)**
 - Number of random block reads that a disk can support per second
 - 50 to 200 IOPS on current generation magnetic disks



Performance Measures (Cont.)

- **Mean time to failure (MTTF)** – the average time the disk is expected to run continuously without any failure.
 - Typically 3 to 5 years
 - Probability of failure of new disks is quite low, corresponding to a “theoretical MTTF” of 500,000 to 1,200,000 hours for a new disk
 - E.g., an MTTF of 1,200,000 hours for a new disk means that given 1000 relatively new disks, on an average one will fail every 1200 hours
 - MTTF decreases as disk ages



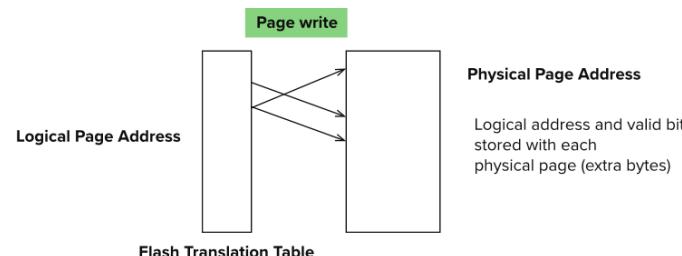
Flash Storage

- NOR flash vs NAND flash
- NAND flash
 - used widely for storage, cheaper than NOR flash
 - requires page-at-a-time read (page: 512 bytes to 4 KB)
 - 20 to 100 microseconds for a page read
 - Not much difference between sequential and random read
 - Page can only be written once
 - Must be erased to allow rewrite
- **Solid state disks**
 - Use standard block-oriented disk interfaces, but store data on multiple flash storage devices internally
 - Transfer rate of up to 500 MB/sec using SATA, and up to 3 GB/sec using NVMe PCIe



Flash Storage (Cont.)

- Erase happens in units of **erase block**
 - Takes 2 to 5 millisecs
 - Erase block typically 256 KB to 1 MB (128 to 256 pages)
- **Remapping** of logical page addresses to physical page addresses avoids waiting for erase
- **Flash translation table** tracks mapping
 - also stored in a label field of flash page
 - remapping carried out by **flash translation layer**



- After 100,000 to 1,000,000 erases, erase block becomes unreliable and cannot be used
 - **wear leveling**



SSD Performance Metrics

- Random reads/writes per second
 - Typical 4 KB reads: 10,000 reads per second (10,000 IOPS)
 - Typical 4KB writes: 40,000 IOPS
 - SSDs support parallel reads
 - Typical 4KB reads:
 - 100,000 IOPS with 32 requests in parallel (QD-32) on SATA
 - 350,000 IOPS with QD-32 on NVMe PCIe
 - Typical 4KB writes:
 - 100,000 IOPS with QD-32, even higher on some models
- Data transfer rate for sequential reads/writes
 - 400 MB/sec for SATA3, 2 to 3 GB/sec using NVMe PCIe
- **Hybrid disks:** combine small amount of flash cache with larger magnetic disk



Storage Class Memory

- 3D-XPoint memory technology pioneered by Intel
- Available as Intel Optane
 - SSD interface shipped from 2017
 - Allows lower latency than flash SSDs
 - Non-volatile memory interface announced in 2018
 - Supports direct access to words, at speeds comparable to main-memory speeds

Redundant Array of Independent Disks (RAID)



“RAID (redundant array of independent disks) is a data [storage virtualization](#) technology that combines multiple physical [disk drive](#) components into a single logical unit for the purposes of [data redundancy](#), performance improvement, or both. (...)

[RAID 0](#) consists of [striping](#), without [mirroring](#) or [parity](#). (...)

[RAID 1](#) consists of data mirroring, without parity or striping. (...)

[RAID 2](#) consists of bit-level striping with dedicated [Hamming-code](#) parity. (...)

[RAID 3](#) consists of byte-level striping with dedicated parity. (...)

[RAID 4](#) consists of block-level striping with dedicated parity. (...)

[RAID 5](#) consists of block-level striping with distributed parity. (...)

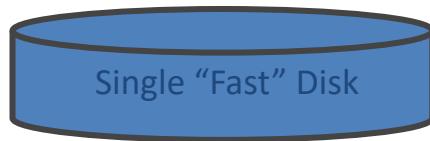
[RAID 6](#) consists of block-level striping with double distributed parity. (...)

Nested RAID

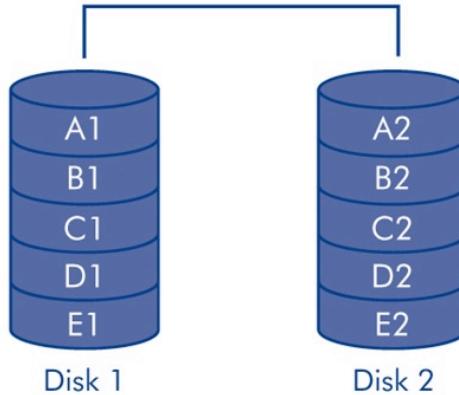
- RAID 0+1: creates two stripes and mirrors them. (...)
- RAID 1+0: creates a striped set from a series of mirrored drives. (...)
- **[JBOD RAID N+N](#)**: With JBOD (*just a bunch of disks*), (...”)

RAID-0 and RAID-1

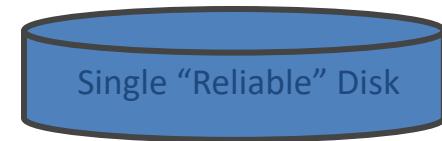
Two physical disks make
one single, logical **fast** disk



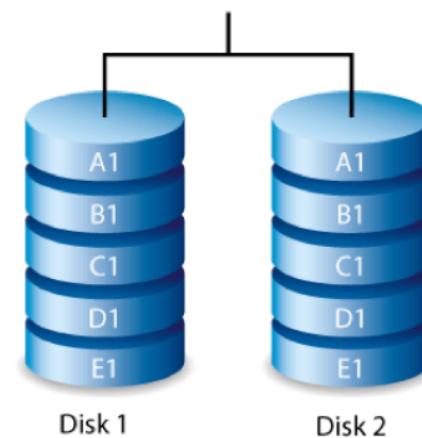
RAID 0



Two physical disks make
one single, logical **reliable** disk

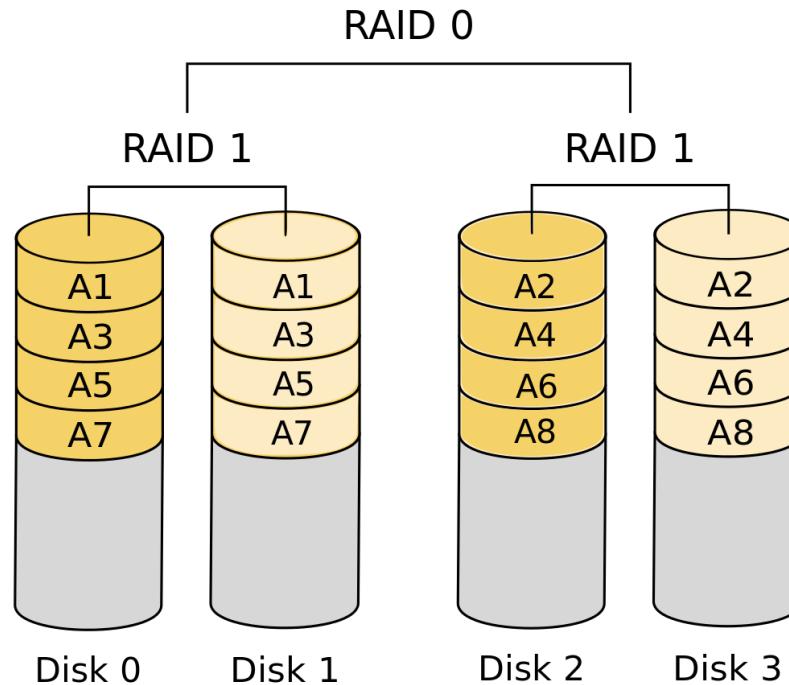


RAID 1



Mixed RAID Modes

RAID 1+0



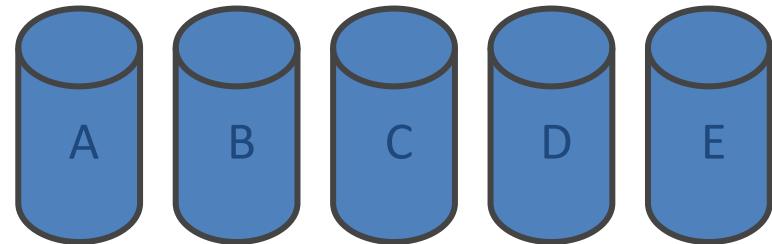
Stripe
And
Mirror

RAID-5

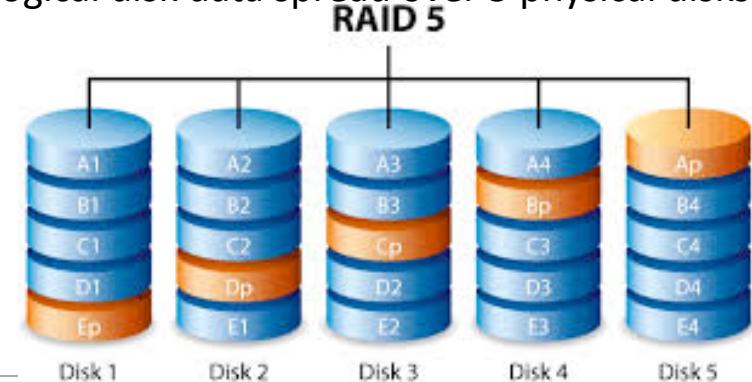
- Improved performance through parallelism
 - Rotation/seek
 - Transfer
- Availability uses *parity blocks*
 - Suppose I have 4 different data blocks on the logical drive A: A1, A2, A3, A4.
 - Parity function: $\text{Ap} = P(A1, A2, A3, A4)$
 - Recovery function: $A2 = R(\text{Ap}, A1, A3, A4)$
- During normal operations:
 - Read processing simply retrieves block.
 - Write processing of A2 updates A2 and Ap
- If an individual disk fails, the RAID
 - Read
 - Continues to function for reads on non-missing blocks.
 - Implements read on missing block by recalculating value.
 - Write
 - Updates block and parity block for non-missing blocks.
 - Computes missing block, and calculates parity based on old and new value.
 - Over time
 - “Hot Swap” the failed disk.
 - Rebuild the missing data from values and parity.



Is actually 5 smaller “logical” disks.



Logical disk data spread over 5 physical disks.



Very Simple Parity Example

- Even-Odd Parity
 - $b[i]$ is an array of bits (0 or 1)
 - $P(b[i]) =$
 - 0 if an even number of bits = 1. $\{P([0,1,1,0,1,1])=0$
 - 1 if an odd number of bits = 1. $\{P(0,0,1,0,1,1)=1$
 - Given an array with one missing bit and the parity bit, I can re-compute the missing bit.
 - Case 1: $[0,?,1,0,1,1]$ has $P=0$. There must be an EVEN number of ones and $?=1$.
 - Case 2: $[0,?,1,0,1,1]$ has $P=1$. There must be an ODD number of ones and $?=0$.
- Block Parity applies this to a set of blocks bitwise

$$\left. \begin{array}{l} - A1 = [0, 1, 0, 0, 1, 1] \\ - A2 = [1, 1, 1, 0, 0, 0] \\ - A3 = [0, 0, 0, 1, 0, 1] \\ - Pa = [1, 0, 1, 1, 1, 0] \end{array} \right\} \rightarrow$$

If I am missing a block and have the parity block, I can re-compute the missing block bitwise from remaining blocks and parity block.



RAID

- **RAID: Redundant Arrays of Independent Disks**
 - disk organization techniques that manage a large numbers of disks, providing a view of a single disk of
 - **high capacity** and **high speed** by using multiple disks in parallel,
 - **high reliability** by storing data redundantly, so that data can be recovered even if a disk fails
- The chance that some disk out of a set of N disks will fail is much higher than the chance that a specific single disk will fail.
 - E.g., a system with 100 disks, each with MTTF of 100,000 hours (approx. 11 years), will have a system MTTF of 1000 hours (approx. 41 days)
 - Techniques for using redundancy to avoid data loss are critical with large numbers of disks



Improvement of Reliability via Redundancy

- **Redundancy** – store extra information that can be used to rebuild information lost in a disk failure
- E.g., **Mirroring** (or **shadowing**)
 - Duplicate every disk. Logical disk consists of two physical disks.
 - Every write is carried out on both disks
 - Reads can take place from either disk
 - If one disk in a pair fails, data still available in the other
 - Data loss would occur only if a disk fails, and its mirror disk also fails before the system is repaired
 - Probability of combined event is very small
 - Except for dependent failure modes such as fire or building collapse or electrical power surges
- **Mean time to data loss** depends on mean time to failure, and **mean time to repair**
 - E.g., MTTF of 100,000 hours, mean time to repair of 10 hours gives mean time to data loss of 500×10^6 hours (or 57,000 years) for a mirrored pair of disks (ignoring dependent failure modes)



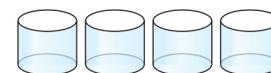
Improvement in Performance via Parallelism

- Two main goals of parallelism in a disk system:
 1. Load balance multiple small accesses to increase throughput
 2. Parallelize large accesses to reduce response time.
- Improve transfer rate by striping data across multiple disks.
- **Bit-level striping** – split the bits of each byte across multiple disks
 - In an array of eight disks, write bit i of each byte to disk i .
 - Each access can read data at eight times the rate of a single disk.
 - But seek/access time worse than for a single disk
 - Bit level striping is not used much any more
- **Block-level striping** – with n disks, block i of a file goes to disk $(i \bmod n) + 1$
 - Requests for different blocks can run in parallel if the blocks reside on different disks
 - A request for a long sequence of blocks can utilize all disks in parallel



RAID Levels

- Schemes to provide redundancy at lower cost by using disk striping combined with parity bits
 - Different RAID organizations, or RAID levels, have differing cost, performance and reliability characteristics
- **RAID Level 0:** Block striping; non-redundant.
 - Used in high-performance applications where data loss is not critical.
- **RAID Level 1:** Mirrored disks with block striping
 - Offers best write performance.
 - Popular for applications such as storing log files in a database system.



(a) RAID 0: nonredundant striping



(b) RAID 1: mirrored disks



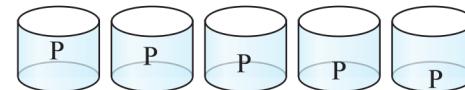
RAID Levels (Cont.)

- **Parity blocks:** Parity block j stores XOR of bits from block j of each disk
 - When writing data to a block j , parity block j must also be computed and written to disk
 - Can be done by using old parity block, old value of current block and new value of current block (2 block reads + 2 block writes)
 - Or by recomputing the parity value using the new values of blocks corresponding to the parity block
 - More efficient for writing large amounts of data sequentially
 - To recover data for a block, compute XOR of bits from all other blocks in the set including the parity block



RAID Levels (Cont.)

- **RAID Level 5:** Block-Interleaved Distributed Parity; partitions data and parity among all $N + 1$ disks, rather than storing data in N disks and parity in 1 disk.
 - E.g., with 5 disks, parity block for n th set of blocks is stored on disk $(n \bmod 5) + 1$, with the data blocks stored on the other 4 disks.



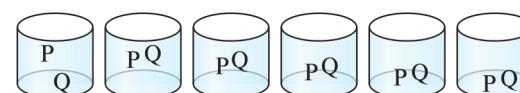
(c) RAID 5: block-interleaved distributed parity

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 Levels (Cont.)

- **RAID Level 5 (Cont.)**
 - Block writes occur in parallel if the blocks and their parity blocks are on different disks.
- **RAID Level 6: P+Q Redundancy** scheme; similar to Level 5, but stores two error correction blocks (P, Q) instead of single parity block to guard against multiple disk failures.
 - Better reliability than Level 5 at a higher cost
 - Becoming more important as storage sizes increase



(d) RAID 6: P + Q redundancy



RAID Levels (Cont.)

- Other levels (not used in practice):
 - RAID Level 2: Memory-Style Error-Correcting-Codes (ECC) with bit striping.
 - RAID Level 3: Bit-Interleaved Parity
 - RAID Level 4: Block-Interleaved Parity; uses block-level striping, and keeps a parity block on a separate **parity disk** for corresponding blocks from N other disks.
 - RAID 5 is better than RAID 4, since with RAID 4 with random writes, parity disk gets much higher write load than other disks and becomes a bottleneck



Choice of RAID Level

- Factors in choosing RAID level
 - Monetary cost
 - Performance: Number of I/O operations per second, and bandwidth during normal operation
 - Performance during failure
 - Performance during rebuild of failed disk
 - Including time taken to rebuild failed disk
- RAID 0 is used only when data safety is not important
 - E.g., data can be recovered quickly from other sources



Choice of RAID Level (Cont.)

- Level 1 provides much better write performance than level 5
 - Level 5 requires at least 2 block reads and 2 block writes to write a single block, whereas Level 1 only requires 2 block writes
- Level 1 had higher storage cost than level 5
- Level 5 is preferred for applications where writes are sequential and large (many blocks), and need large amounts of data storage
- RAID 1 is preferred for applications with many random/small updates
- Level 6 gives better data protection than RAID 5 since it can tolerate two disk (or disk block) failures
 - Increasing in importance since latent block failures on one disk, coupled with a failure of another disk can result in data loss with RAID 1 and RAID 5.



Hardware Issues

- **Software RAID:** RAID implementations done entirely in software, with no special hardware support
- **Hardware RAID:** RAID implementations with special hardware
 - Use non-volatile RAM to record writes that are being executed
 - Beware: power failure during write can result in corrupted disk
 - E.g., failure after writing one block but before writing the second in a mirrored system
 - Such corrupted data must be detected when power is restored
 - Recovery from corruption is similar to recovery from failed disk
 - NV-RAM helps to efficiently detect potentially corrupted blocks
 - Otherwise all blocks of disk must be read and compared with mirror/parity block



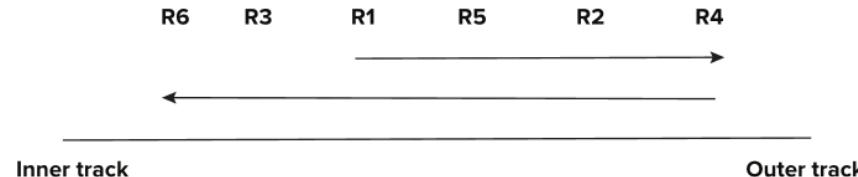
Hardware Issues (Cont.)

- **Latent failures:** data successfully written earlier gets damaged
 - can result in data loss even if only one disk fails
- **Data scrubbing:**
 - continually scan for latent failures, and recover from copy/parity
- **Hot swapping:** replacement of disk while system is running, without power down
 - Supported by some hardware RAID systems,
 - reduces time to recovery, and improves availability greatly
- Many systems maintain **spare disks** which are kept online, and used as replacements for failed disks immediately on detection of failure
 - Reduces time to recovery greatly
- Many hardware RAID systems ensure that a single point of failure will not stop the functioning of the system by using
 - Redundant power supplies with battery backup
 - Multiple controllers and multiple interconnections to guard against controller/interconnection failures



Optimization of Disk-Block Access

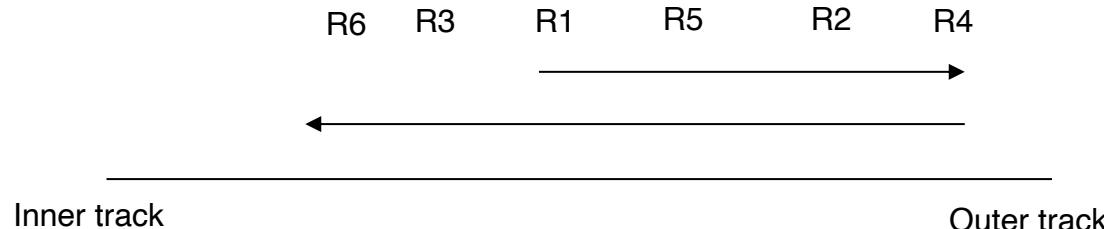
- **Buffering:** in-memory buffer to cache disk blocks
- **Read-ahead:** Read extra blocks from a track in anticipation that they will be requested soon
- **Disk-arm-scheduling** algorithms re-order block requests so that disk arm movement is minimized
 - **elevator algorithm**





Optimization of Disk-Block Access

- **Buffering:** in-memory buffer to cache disk blocks
- **Read-ahead:** Read extra blocks from a track in anticipation that they will be requested soon
- **Disk-arm-scheduling** algorithms re-order block requests so that disk arm movement is minimized
 - **elevator algorithm**



Disk Scheduling

- An example – *Elevator Algorithm*
 - Assume
 - The seek time from track n to m is $\text{ABS}(n-m)$.
 - The I/O request queue is for sectors on tracks 1, 9, 2, 8, 3.
 - The head is currently on track 5.
 - Total seek time for FIFO processing
 - Seek order is 5 → 1 → 9 → 2 → 8 → 3
 - Total seek time is $\text{ABS}(5-1) + \text{ABS}(1-9) + \text{ABS}(9-2) + \text{ABS}(2-8) + \text{ABS}(8-3) = 30$
 - If the head goes all the way up and all the way down, like an elevator,
 - The seek order 5 → 8 → 9 → 3 → 2 → 1
 - Total seek time is $\text{ABS}(5-8) + \text{ABS}(8-9) + \text{ABS}(9-3) + \text{ABS}(3-2) + \text{ABS}(2-1) = 12$
- There is a rich body of research and algorithms on optimizing
 - Blocks on tracks/sectors.
 - Seek/rotation optimization.

Data Access Patterns

- Think about accessing tuples in a database. The access patterns are not completely random. For example,
 - SELECT ... ORDER BY ... → *I look for things in sequence.*
 - SELECT ... WHERE last_name="Smith" → *I look for sets of things.*
 - CREATE VIEW AS SELECT ... FROM JOIN ... ON A.x=B.y →
I look for things in pairs of sets.
- Optimizing I/O performance
 - Group records into blocks if accessed together.
 - Group blocks onto the same sector/cylinder if accessed together/sequentially → Eliminates seek time.
 - Schedule access:
 - If I have to read cylinders 1, 9, 2, 8, 3
 - A better ordering is 1,2,3,9,8.
 - Stripe across disks: Read one block from 10 disks instead one 10 from 1.
 - Pre-fetch and preload.

Logical Block Addressing (https://gerardnico.com/wiki/data_storage/lba)

3 - The LBA scheme

LBA	C	H	S
0	0	0	0
1	0	0	1
2	0	0	2
3	0	0	3
4	0	0	4
5	0	0	5
6	0	0	6
7	0	0	7
8	0	0	8
9	0	0	9
10	0	1	0
11	0	1	1
12	0	1	2
13	0	1	3
14	0	1	4
15	0	1	5
16	0	1	6
17	0	1	7
18	0	1	8
19	0	1	9
Cylinder 0			

LBA	C	H	S
20	1	0	0
21	1	0	1
22	1	0	2
23	1	0	3
24	1	0	4
25	1	0	5
26	1	0	6
27	1	0	7
28	1	0	8
29	1	0	9
30	1	1	0
31	1	1	1
32	1	1	2
33	1	1	3
34	1	1	4
35	1	1	5
36	1	1	6
37	1	1	7
38	1	1	8
39	1	1	9
Cylinder 1			

- CHS addresses can be converted to LBA addresses using the following formula:

$$\text{LBA} = ((\text{C} \times \text{HPC}) + \text{H}) \times \text{SPT} + \text{S} - 1$$

where,

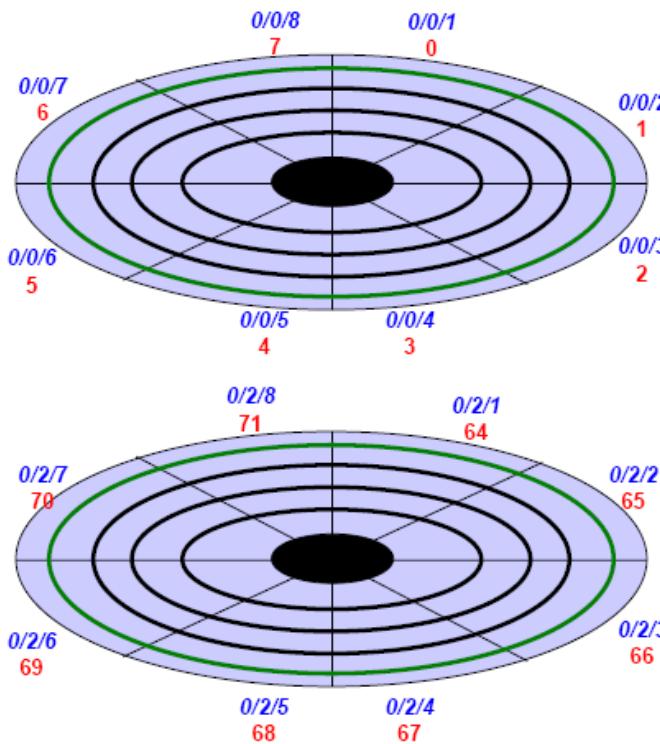
- C, H and S are the cylinder number, the head number, and the sector number
- LBA is the logical block address
- HPC is the number of heads per cylinder
- SPT is the number of sectors per track

Devices have configuration and metadata APIs that allow storage manager to

- Map between LBA and CHS
- To optimize block placement
- Based on access patterns, statistics, data schema, etc.

Another View

(http://www.c-jump.com/CIS24/Slides/DiskDrives/D01_0150_chs_vs_lba.htm)



Simple example: 4 sides, 8 sectors/side, 4 tracks/side (cylinders) yields 128 total sectors...

The green track is cylinder 0. On the first side (side 0), sectors are numbered 0/0/1 thru 0/0/8 in CHS format, or 0 thru 7 in LBA.

On cylinder 1 of the first side, sectors are numbered 1/0/1 thru 1/0/8 (8 thru 15).

The last sector on side 0 is numbered 4/0/8 (31).

Second side (side 1) starts: 0/1/1 32
Second side ends: 4/1/8 63

Third side (side 2) starts: 0/2/1 64
Third side ends: 4/2/8 95

Fourth side (side 3) starts: 0/3/1 96
Fourth side ends: 4/3/8 127

Data Storage Structures

(Database Systems Concepts, V7, Ch. 13)



File Organization

- The database is stored as a collection of *files*. Each file is a sequence of *records*. A record is a sequence of fields.
- One approach
 - Assume record size is fixed
 - Each file has records of one particular type only
 - Different files are used for different relations
- This case is easiest to implement; will consider variable length records later
- We assume that records are smaller than a disk block

Terminology

- A tuple in a relation maps to a *record*. Records may be
 - *Fixed length*
 - *Variable length*
 - *Variable format* (which we will see in Neo4J, DynamoDB, etc).
- A *block*
 - Is the unit of transfer between disks and memory (buffer pools).
 - Contains multiple records, usually but not always from the same relation.
- The *database address space* contains
 - All of the blocks and records that the database manages
 - Including blocks/records containing data
 - And blocks/records containing free space.

Fixed Length Record

- Sample DDL

```
CREATE TABLE `products` (
  `product_id` char(8) NOT NULL,
  `product_name` varchar(16) NOT NULL,
  `product_description` char(8) NOT NULL,
  `product_brand` enum('IBM','HP','Acer','Lenovo','Some really long brand
name') NOT NULL,
  PRIMARY KEY (`product_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

- Why
 - Is this fixed length? product_brand and product_name are clearly variable length.
 - Isn't char(8) too short for a product description?

Hypothetical Explanations

“In computing, **internationalization and localization** are means of adapting computer software to different languages, regional differences and technical requirements of a target locale.¹¹ Internationalization is the process of designing a software application so that it can be adapted to various languages and regions without engineering changes. Localization is the process of adapting internationalized software for a specific region or language by adding locale-specific components and translating text. Localization (which is potentially performed multiple times, for different locales) uses the infrastructure or flexibility provided by internationalization (which is ideally performed only once, or as an integral part of ongoing development).” (https://en.wikipedia.org/wiki/Internationalization_and_localization)

- Retrieving a product description may require two elements
 - Language code, e.g. (EN, DE, ES, ...)
 - Description ID, e.g. 01105432
 - (EN, 01105432) and (DE, 01105432) are the same description in different languages (English and German)
 - The description ID is an attribute of the product.
 - The language is an attribute or property of the user, install location, etc.

Hypothetical Explanations

"In contrast, different locales have varying internal representations and (in some) uses the same, or as an option)

are to be adapted by adapting elements and (in some) uses the same, or as an option)

- This concept is
 - An element of data modeling and design
 - Not DBMS implementation or data management.
- I introduce the concept because
 - Important, general data model topic we have not covered.
 - Pattern is a reason why many records we think would be variable are often fixed.
 - VARCHAR field in a tuple is often
 - A pointer to a value in some other record.
-]

ent

, etc.

Hypothetical Explanation

- `product_name` ***varchar(16) NOT NULL***
 - (NB: Assumes that the product_name is not localized).
 - Length can be 0 .. 15 characters.
 - But,
 - If the value is almost always between 12 and 16 characters.
 - The database engine may decide to store in a fixed length area.
 - Because the relatively small savings available from treating like a variable length record
 - Does not justify the added complexity involved in variable size record management.
- `product_brand` enum('IBM','HP','Acer','Lenovo','Some really long brand name')
 - The DB engine would/could represent with a 1 byte code in a lookup table.
 - This enables simplified, fixed length record management
 - And saves space because
 - 1,000 products in the catalog from 'Some really long brand'
 - Requires 1,000 bytes instead of 27,000 bytes.
 - Repeated values:
 - The same value occurring repeatedly in multiple distinct tuples is common, even when not an enum, e.g.
 - Country names, City names, street names, last names, first names, ...
 - Foreign keys in many-to-many associative entities.
 - DBMS storage management engine may detect the situation, and replace values with symbol lookups.



Fixed-Length Records

- Simple approach:
 - Store record i starting from byte $n * (i - 1)$, where n is the size of each record.
 - Record access is simple but records may cross blocks
 - Modification: do not allow records to cross block boundaries

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000



Fixed-Length Records

- Deletion of record i : alternatives:
 - **move records $i + 1, \dots, n$ to $i, \dots, n - 1$**
 - move record n to i
 - do not move records, but link all free records on a *free list*

Record 3 deleted

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000



Fixed-Length Records

- Deletion of record i : alternatives:
 - move records $i + 1, \dots, n$ to $i, \dots, n - 1$
 - **move record n to i**
 - do not move records, but link all free records on a *free list*

Record 3 deleted and replaced by record 11

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000



Fixed-Length Records

- Deletion of record i : alternatives:
 - move records $i + 1, \dots, n$ to $i, \dots, n - 1$
 - move record n to i
 - **do not move records, but link all free records on a *free list***

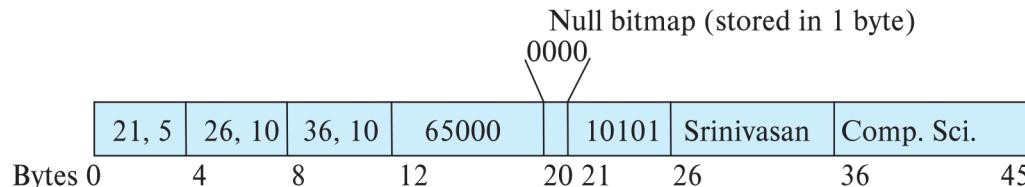
header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

The diagram illustrates a linked list of free records. Four arrows point from the empty fourth column of records 4, 5, 6, and 7 to a common pointer at the bottom right, which then points to a blank line representing the end of the list.



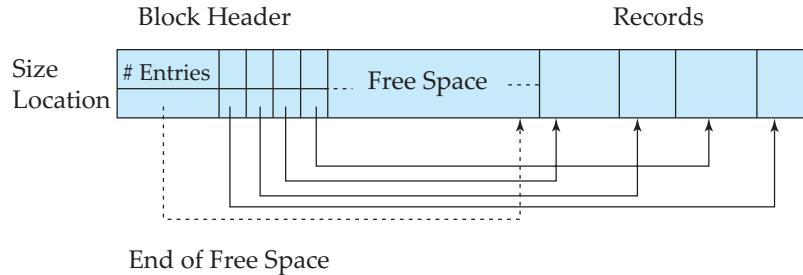
Variable-Length Records

- Variable-length records arise in database systems in several ways:
 - Storage of multiple record types in a file.
 - Record types that allow variable lengths for one or more fields such as strings (**varchar**)
 - Record types that allow repeating fields (used in some older data models).
- Attributes are stored in order
- Variable length attributes represented by fixed size (offset, length), with actual data stored after all fixed length attributes
- Null values represented by null-value bitmap





Variable-Length Records: Slotted Page Structure



- **Slotted page** header contains:
 - number of record entries
 - end of free space in the block
 - location and size of each record
- Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.
- Pointers should not point directly to record — instead they should point to the entry for the record in header.



Storing Large Objects

- E.g., blob/clob types
- Records must be smaller than pages
- Alternatives:
 - Store as files in file systems
 - Store as files managed by database
 - Break into pieces and store in multiple tuples in separate relation
 - PostgreSQL TOAST



Organization of Records in Files

- **Heap** – record can be placed anywhere in the file where there is space
- **Sequential** – store records in sequential order, based on the value of the search key of each record
- In a **multitable clustering file organization** records of several different relations can be stored in the same file
 - Motivation: store related records on the same block to minimize I/O
- **B⁺-tree file organization**
 - Ordered storage even with inserts/deletes
 - More on this in Chapter 14
- **Hashing** – a hash function computed on search key; the result specifies in which block of the file the record should be placed
 - More on this in Chapter 14



Heap File Organization

- Records can be placed anywhere in the file where there is free space
- Records usually do not move once allocated
- Important to be able to efficiently find free space within file
- **Free-space map**
 - Array with 1 entry per block. Each entry is a few bits to a byte, and records fraction of block that is free
 - In example below, 3 bits per block, value divided by 8 indicates fraction of block that is free

4	2	1	4	7	3	6	5	1	2	0	1	1	0	5	6
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

- Can have second-level free-space map
- In example below, each entry stores maximum from 4 entries of first-level free-space map

4	7	2	6
---	---	---	---

- Free space map written to disk periodically, OK to have wrong (old) values for some entries (will be detected and fixed)



Sequential File Organization

- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a search-key

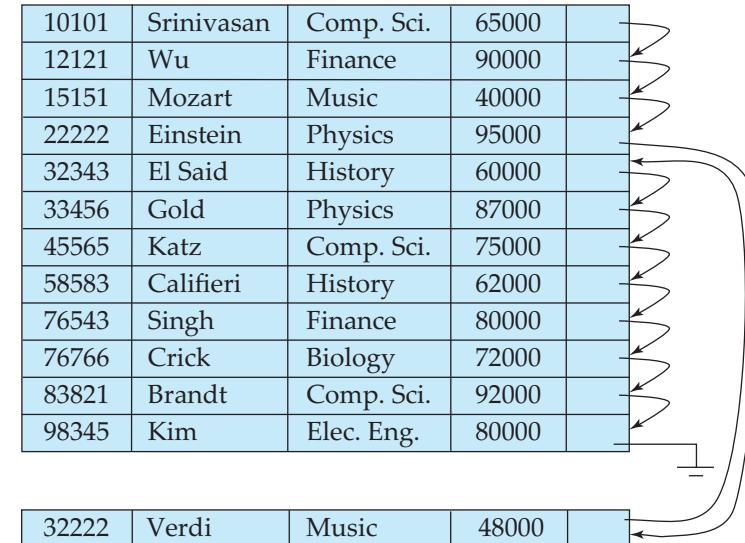
10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

A diagram illustrating a sequential file organization. It shows a table with 12 rows, each representing a record. The records are ordered by a search-key. Arrows point from the right edge of each row to a vertical line at the bottom, indicating the sequential nature of the file.



Sequential File Organization (Cont.)

- Deletion – use pointer chains
- Insertion – locate the position where the record is to be inserted
 - if there is free space insert there
 - if no free space, insert the record in an **overflow block**
 - In either case, pointer chain must be updated
- Need to reorganize the file from time to time to restore sequential order





Multitable Clustering File Organization

Store several relations in one file using a **multitable clustering** file organization

department

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Comp. Sci.	Taylor	100000
Physics	Watson	70000

instructor

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

multitable clustering
of *department* and
instructor

Comp. Sci.	Taylor	100000	
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
Physics	Watson	70000	
33456	Gold	Physics	87000



Multitable Clustering File Organization (cont.)

- good for queries involving *department* \bowtie *instructor*, and for queries involving one single department and its instructors
- bad for queries involving only *department*
- results in variable size records
- Can add pointer chains to link records of a particular relation



Partitioning

- **Table partitioning:** Records in a relation can be partitioned into smaller relations that are stored separately
- E.g., *transaction* relation may be partitioned into *transaction_2018*, *transaction_2019*, etc.
- Queries written on *transaction* must access records in all partitions
 - Unless query has a selection such as *year=2019*, in which case only one partition is needed
- Partitioning
 - Reduces costs of some operations such as free space management
 - Allows different partitions to be stored on different storage devices
 - E.g., *transaction* partition for current year on SSD, for older years on magnetic disk



Data Dictionary Storage

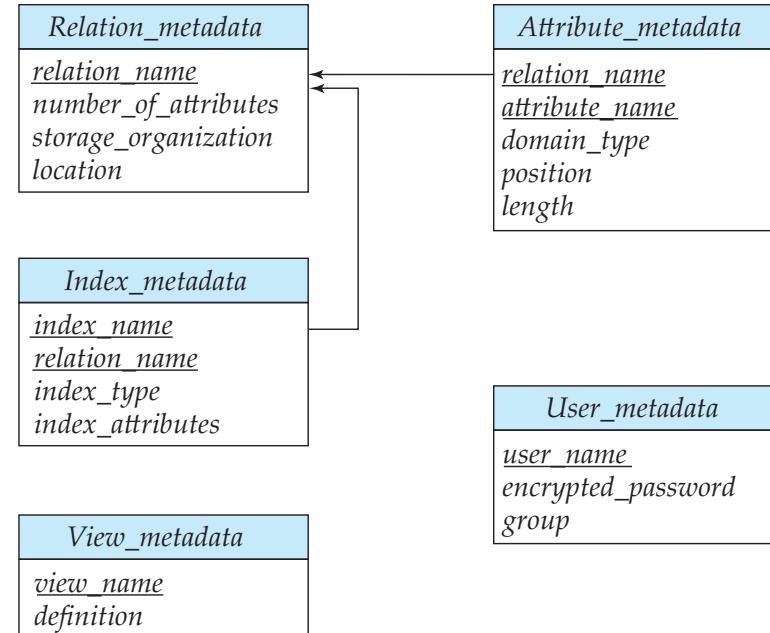
The **Data dictionary** (also called **system catalog**) stores **metadata**; that is, data about data, such as

- Information about relations
 - names of relations
 - names, types and lengths of attributes of each relation
 - names and definitions of views
 - integrity constraints
- User and accounting information, including passwords
- Statistical and descriptive data
 - number of tuples in each relation
- Physical file organization information
 - How relation is stored (sequential/hash/...)
 - Physical location of relation
- Information about indices (Chapter 14)



Relational Representation of System Metadata

- Relational representation on disk
- Specialized data structures designed for efficient access, in memory





Column-Oriented Storage

- Also known as **columnar representation**
- Store each attribute of a relation separately
- Example

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000



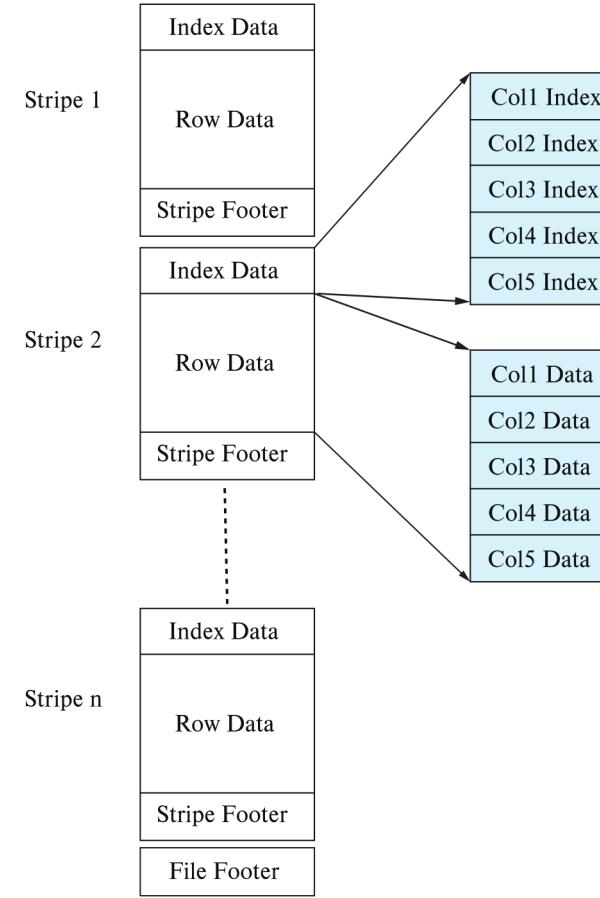
Columnar Representation

- Benefits:
 - Reduced IO if only some attributes are accessed
 - Improved CPU cache performance
 - Improved compression
 - **Vector processing** on modern CPU architectures
- Drawbacks
 - Cost of tuple reconstruction from columnar representation
 - Cost of tuple deletion and update
 - Cost of decompression
- Columnar representation found to be more efficient for decision support than row-oriented representation
- Traditional row-oriented representation preferable for transaction processing
- Some databases support both representations
 - Called **hybrid row/column stores**



Columnar File Representation

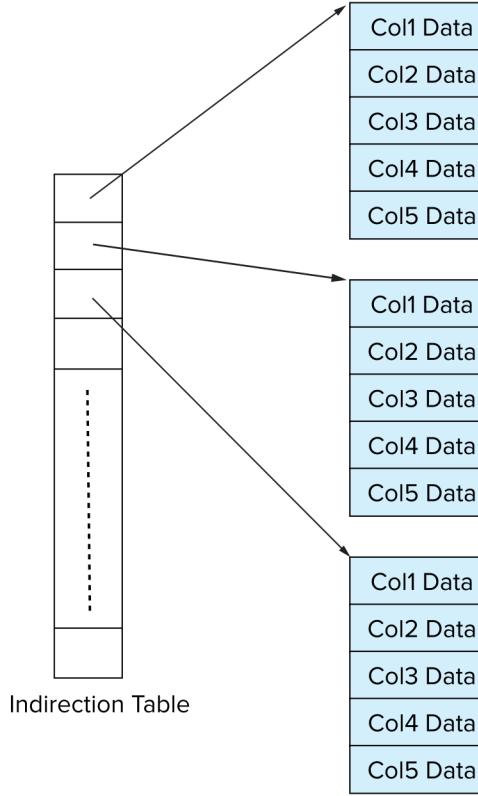
- ORC and Parquet: file formats with columnar storage inside file
- Very popular for big-data applications
- Orc file format shown on right:





Storage Organization in Main-Memory Databases

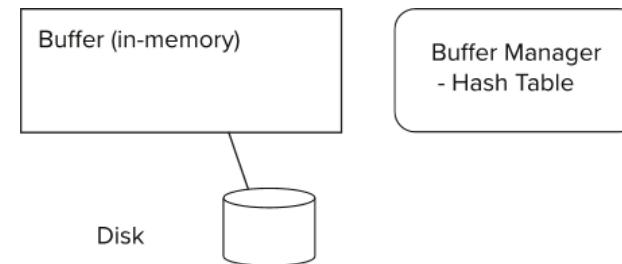
- Can store records directly in memory without a buffer manager
- Column-oriented storage can be used in-memory for decision support applications
 - Compression reduces memory requirement





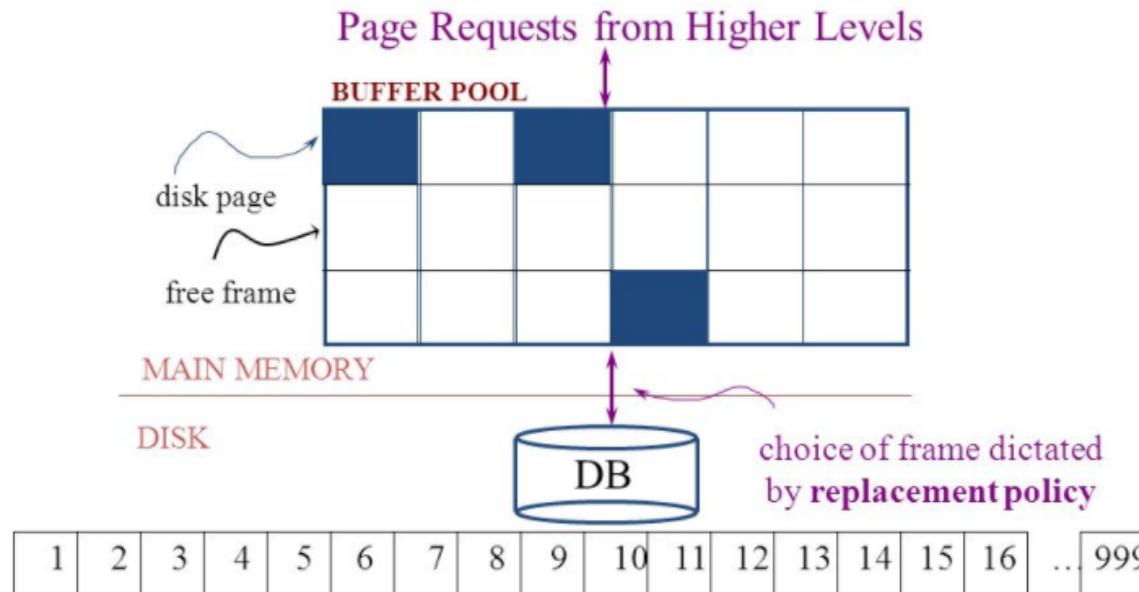
Storage Access

- Blocks are units of both storage allocation and data transfer.
- Database system seeks to minimize the number of block transfers between the disk and memory. We can reduce the number of disk accesses by keeping as many blocks as possible in main memory.
- **Buffer** – portion of main memory available to store copies of disk blocks.
- **Buffer manager** – subsystem responsible for allocating buffer space in main memory.



The Logical Concept

- The DBMS and queries can only manipulate in-memory blocks and records.
- A very, very, very small fraction of all blocks fit in memory.





Buffer Manager

- Programs call on the buffer manager when they need a block from disk.
 - If the block is already in the buffer, buffer manager returns the address of the block in main memory
 - If the block is not in the buffer, the buffer manager
 - Allocates space in the buffer for the block
 - Replacing (throwing out) some other block, if required, to make space for the new block.
 - Replaced block written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
 - Reads the block from the disk to the buffer, and returns the address of the block in main memory to requester.



Buffer Manager

- **Buffer replacement strategy** (details coming up!)
- **Pinned block:** memory block that is not allowed to be written back to disk
 - **Pin** done before reading/writing data from a block
 - **Unpin** done when read /write is complete
 - Multiple concurrent pin/unpin operations possible
 - Keep a pin count, buffer block can be evicted only if pin count = 0
- **Shared and exclusive locks on buffer**
 - Needed to prevent concurrent operations from reading page contents as they are moved/reorganized, and to ensure only one move/reorganize at a time
 - Readers get shared lock, updates to a block require exclusive lock
 - **Locking rules:**
 - Only one process can get exclusive lock at a time
 - Shared lock cannot be concurrently with exclusive lock
 - Multiple processes may be given shared lock concurrently



Buffer-Replacement Policies

- Most operating systems replace the block **least recently used** (LRU strategy)
 - Idea behind LRU – use past pattern of block references as a predictor of future references
 - LRU can be bad for some queries
- Queries have well-defined access patterns (such as sequential scans), and a database system can use the information in a user's query to predict future references
- Mixed strategy with hints on replacement strategy provided by the query optimizer is preferable
- Example of bad access pattern for LRU: when computing the join of 2 relations r and s by a nested loops

```
for each tuple  $tr$  of  $r$  do  
  for each tuple  $ts$  of  $s$  do  
    if the tuples  $tr$  and  $ts$  match ...
```



Buffer-Replacement Policies (Cont.)

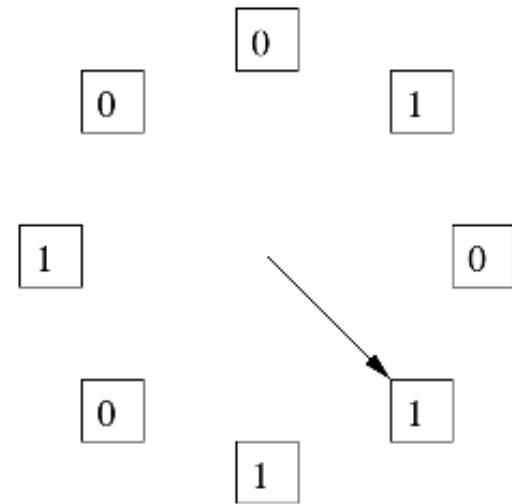
- **Toss-immediate** strategy – frees the space occupied by a block as soon as the final tuple of that block has been processed
- **Most recently used (MRU) strategy** – system must pin the block currently being processed. After the final tuple of that block has been processed, the block is unpinned, and it becomes the most recently used block.
- Buffer manager can use statistical information regarding the probability that a request will reference a particular relation
 - E.g., the data dictionary is frequently accessed. Heuristic: keep data-dictionary blocks in main memory buffer
- Operating system or buffer manager may reorder writes
 - Can lead to corruption of data structures on disk
 - E.g., linked list of blocks with missing block on disk
 - File systems perform consistency check to detect such situations
 - Careful ordering of writes can avoid many such problems

Replacement Policy

- The *replacement policy* is one of the most important factors in database management system implementation and configuration.
- A very simple, introductory explanation is (https://en.wikipedia.org/wiki/Cache_replacement_policies).
 - There are a lot of possible policies.
 - The *most* efficient caching algorithm would be to always discard the information that will not be needed for the longest time in the future. This optimal result is referred to as Bélády's optimal algorithm/simply optimal replacement policy or the clairvoyant algorithm.
- All implementable policies are an attempt to approximate knowledge of the future based on knowledge of the past.
- Least Recently Used is based on the simplest assumption
 - The information that will not be needed for the longest time.
 - Is the information that has not been accessed for the longest time.

The “Clock Algorithm”

- LRU is (perceived to be) expensive
 - Maintain timestamp for each block.
 - Update and resort blocks on access.
- The “Clock Algorithm” is a less expensive approximation.
 - Arrange the frames (places blocks can go) into a logical circle like the seconds on a clock face.
 - Each frame is marked 0 or 1.
 - Set to 1 when block added to frame.
 - Or when application accesses a block in frame.
 - Replacement choice
 - Sweep second hand clockwise one frame at a time.
 - If bit is 0, choose for replacement.
 - If bit is 1, set bit to zero and go to next frame.
- The basic idea is. On a clock face
 - If the second hand is currently at 27 seconds.
 - The 28 second tick mark is “the least recently touched mark.”



Replacement Algorithm

The algorithms are more sophisticated in the real world, e.g.

- “Scans” are common, e.g. go through a large query result in order (will be more clear when discussing cursors).
 - The engine knows the current position in the result set.
 - Uses the sort order to determine which records will be accessed soon.
 - Tags those blocks as not replaceable.
 - (A form of clairvoyance).
- Not all users/applications are equally “important.”
 - Classify users/applications into priority 1, 2 and 3.
 - Sub-allocate the buffer pool into pools P1, P2 and P3.
 - Apply LRU within pools and adjust pool sizes based on relative importance.
 - This prevents
 - A high access rate, low-priority application from taking up a lot of frames
 - Result in low access, high priority applications not getting buffer hits.



Optimization of Disk Block Access (Cont.)

- Buffer managers support **forced output** of blocks for the purpose of recovery (more in Chapter 19)
- **Nonvolatile write buffers** speed up disk writes by writing blocks to a non-volatile RAM or flash buffer immediately
 - *Writes can be reordered to minimize disk arm movement*
- **Log disk** – a disk devoted to writing a sequential log of block updates
 - Used exactly like nonvolatile RAM
 - Write to log disk is very fast since no seeks are required
- **Journaling file systems** write data in-order to NV-RAM or log disk
 - Reordering without journaling: risk of corruption of file system data

Indexesres

(Database Systems Concepts, V7, Ch. 14)



Outline

- Basic Concepts
- Ordered Indices
- B⁺-Tree Index Files
- B-Tree Index Files
- Hashing
- Write-optimized indices
- Spatio-Temporal Indexing



Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
 - E.g., author catalog in library
- **Search Key** - attribute or set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

search-key	pointer
------------	---------
- Index files are typically much smaller than the original file
- Two basic kinds of indices:
 - **Ordered indices:** search keys are stored in sorted order
 - **Hash indices:** search keys are distributed uniformly across “buckets” using a “hash function”.



Index Evaluation Metrics

- Access types supported efficiently. E.g.,
 - Records with a specified value in the attribute
 - Records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead



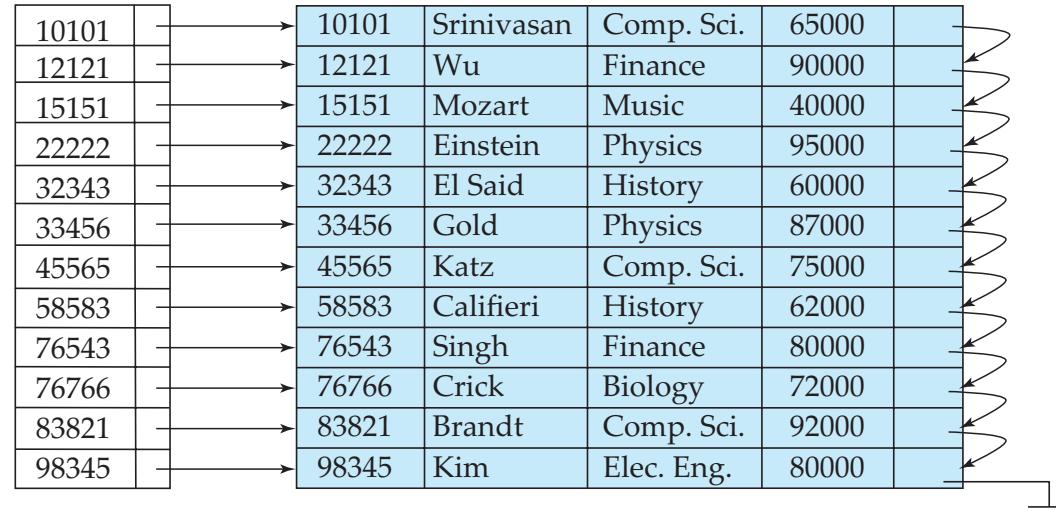
Ordered Indices

- In an **ordered index**, index entries are stored sorted on the search key value.
- **Clustering index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
 - Also called **primary index**
 - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index:** an index whose search key specifies an order different from the sequential order of the file. Also called **nonclustering index**.
- **Index-sequential file:** sequential file ordered on a search key, with a clustering index on the search key.



Dense Index Files

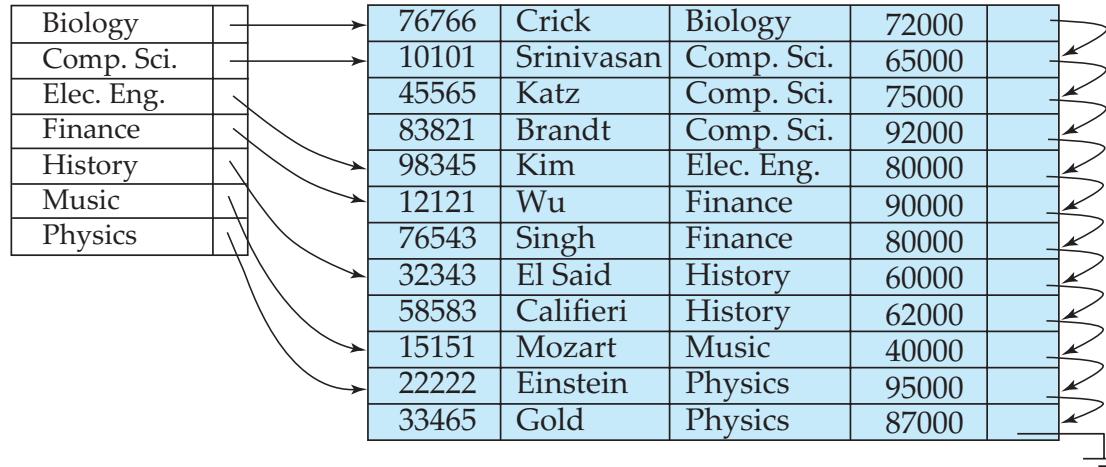
- **Dense index** — Index record appears for every search-key value in the file.
- E.g. index on *ID* attribute of *instructor* relation





Dense Index Files (Cont.)

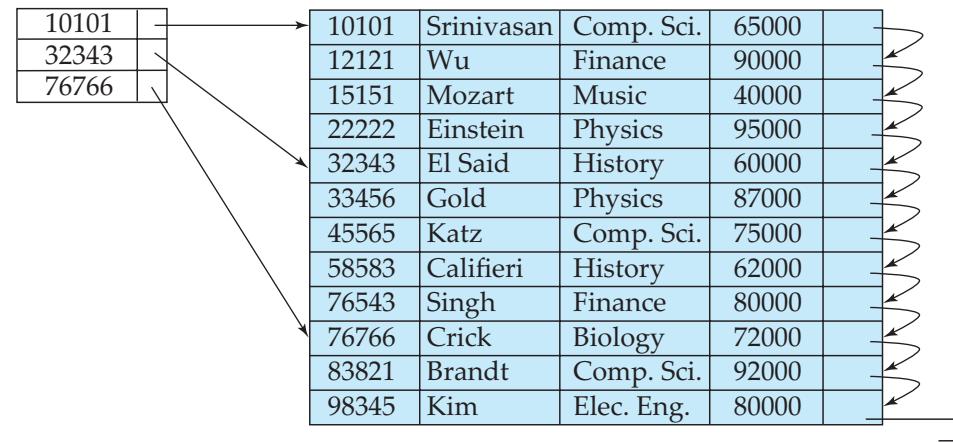
- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*





Sparse Index Files

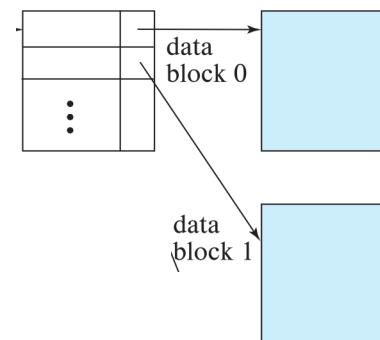
- **Sparse Index:** contains index records for only some search-key values.
 - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value K we:
 - Find index record with largest search-key value $< K$
 - Search file sequentially starting at the record to which the index record points





Sparse Index Files (Cont.)

- Compared to dense indices:
 - Less space and less maintenance overhead for insertions and deletions.
 - Generally slower than dense index for locating records.
- **Good tradeoff:**
 - for clustered index: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.

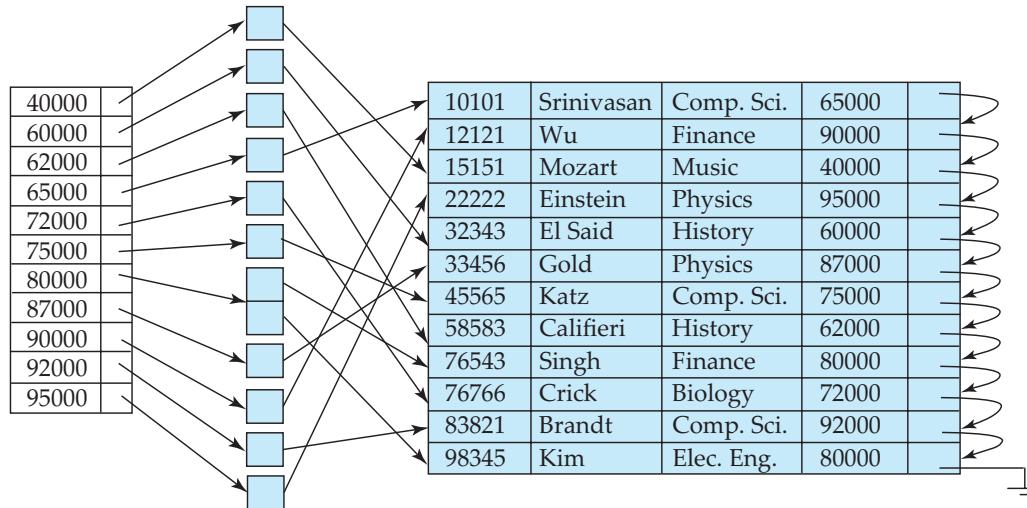


- For unclustered index: sparse index on top of dense index (multilevel index)



Secondary Indices Example

- Secondary index on salary field of instructor



- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense

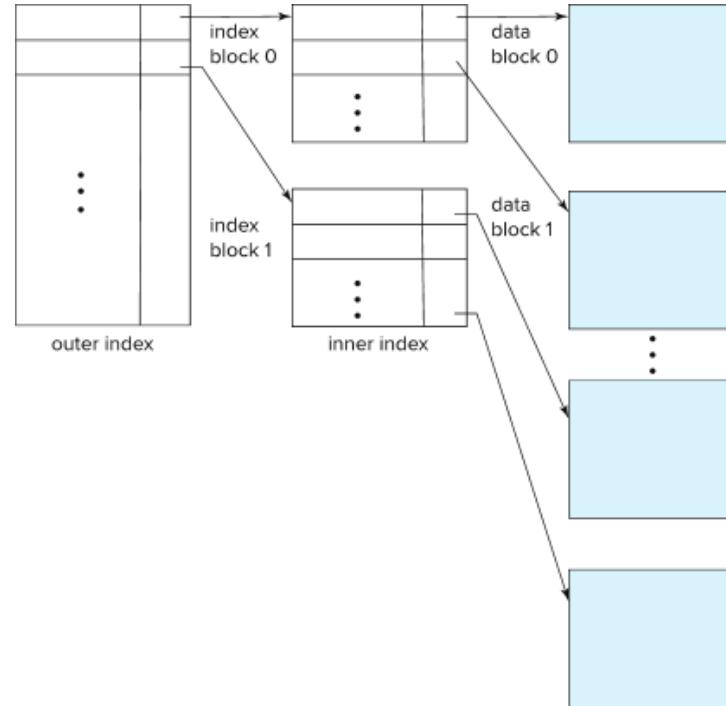


Multilevel Index

- If index does not fit in memory, access becomes expensive.
- Solution: treat index kept on disk as a sequential file and construct a sparse index on it.
 - outer index – a sparse index of the basic index
 - inner index – the basic index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.



Multilevel Index (Cont.)



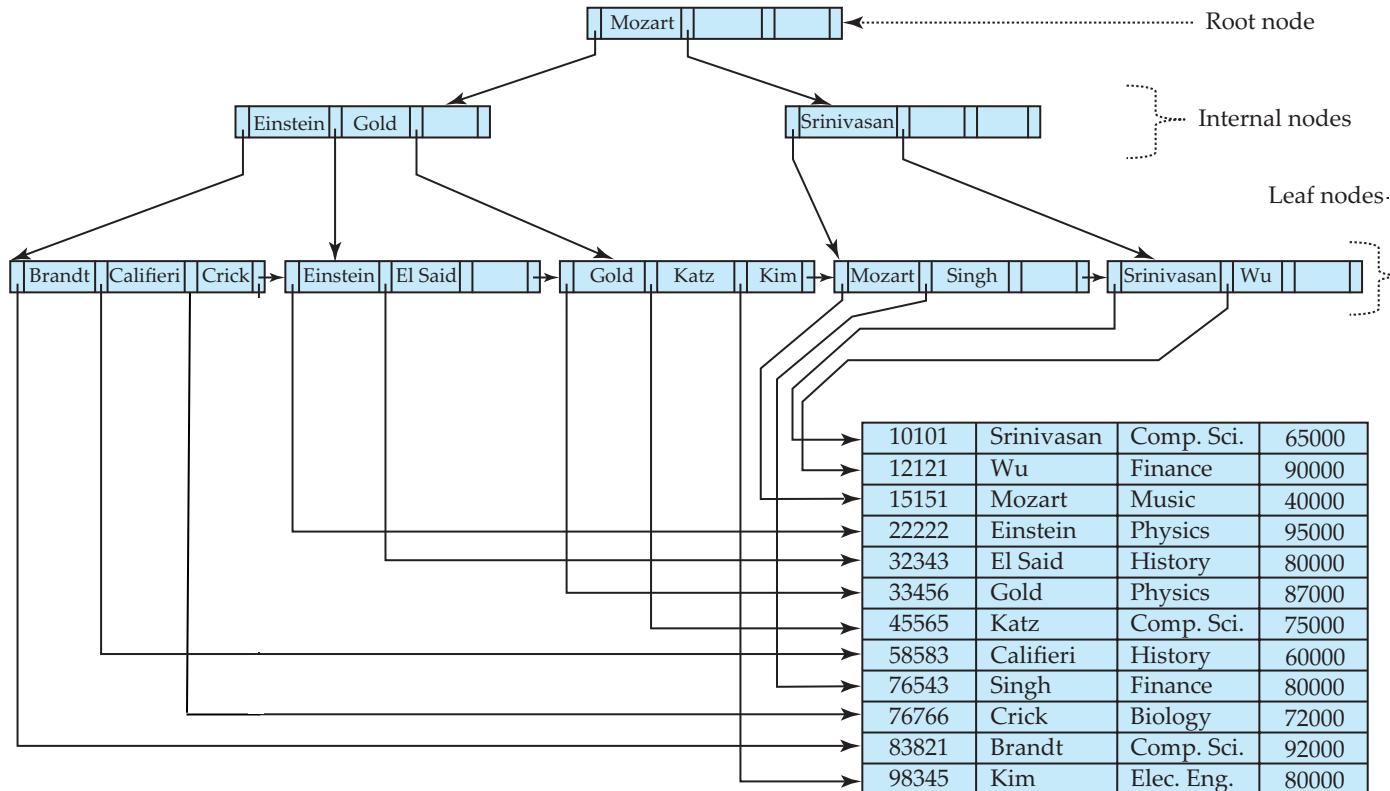


Indices on Multiple Keys

- **Composite search key**
 - E.g., index on *instructor* relation on attributes (*name*, *ID*)
 - Values are sorted lexicographically
 - E.g. (John, 12121) < (John, 13514) and (John, 13514) < (Peter, 11223)
 - Can query on just *name*, or on (*name*, *ID*)



Example of B⁺-Tree





B⁺-Tree Index Files (Cont.)

A B⁺-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
- A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values
- Special cases:
 - If the root is not a leaf, it has at least 2 children.
 - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n-1)$ values.



B⁺-Tree Node Structure

- Typical node

P_1	K_1	P_2	\dots	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	---------	-----------	-----------	-------

- 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

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

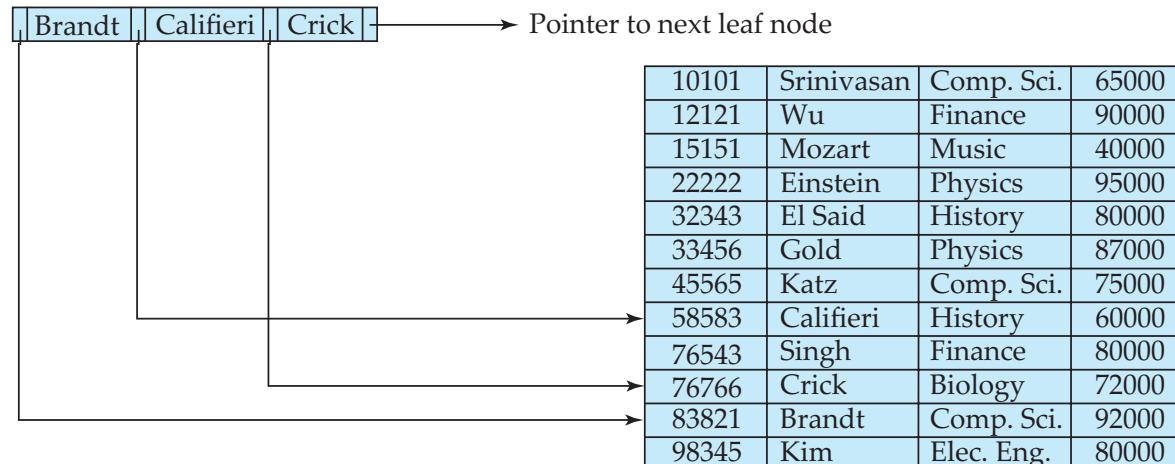
(Initially assume no duplicate keys, address duplicates later)



Leaf Nodes in B⁺-Trees

Properties of a leaf node:

- For $i = 1, 2, \dots, n-1$, pointer P_i points to a file record with search-key value K_i ,
- If L_i, L_j are leaf nodes and $i < j$, L_i 's search-key values are less than or equal to L_j 's search-key values
- P_n points to next leaf node in search-key order





Non-Leaf Nodes in B⁺-Trees

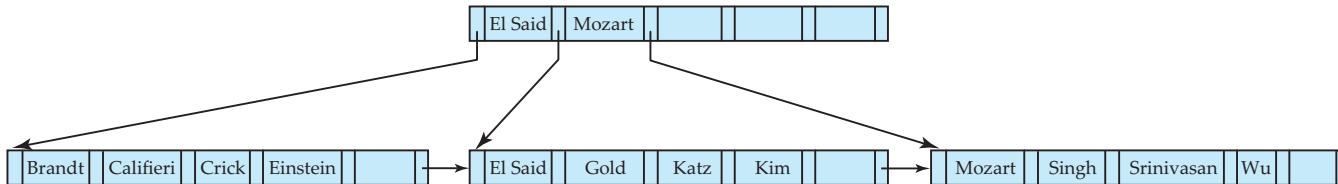
- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:
 - All the search-keys in the subtree to which P_1 points are less than K_1
 - For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i
 - All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}
 - General structure

P_1	K_1	P_2	\dots	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	---------	-----------	-----------	-------



Example of B⁺-tree

- B⁺-tree for *instructor* file ($n = 6$)



- Leaf nodes must have between 3 and 5 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with $n = 6$).
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil (n/2) \rceil$ and n with $n = 6$).
- Root must have at least 2 children.



Observations about B+-trees

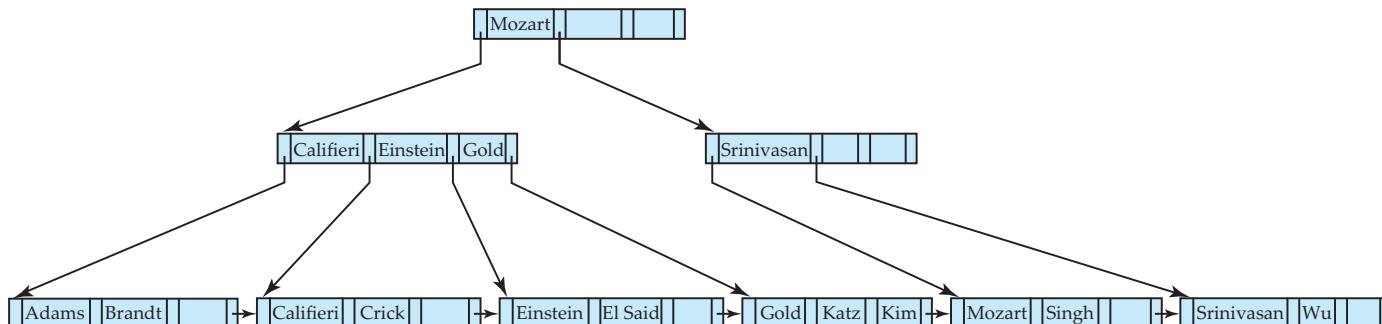
- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close.
- The non-leaf levels of the B+-tree form a hierarchy of sparse indices.
- The B+-tree contains a relatively small number of levels
 - Level below root has at least $2 * \lceil n/2 \rceil$ values
 - Next level has at least $2 * \lceil n/2 \rceil * \lceil n/2 \rceil$ values
 - .. etc.
 - If there are K search-key values in the file, the tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
 - thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see).



Queries on B⁺-Trees

function *find*(*v*)

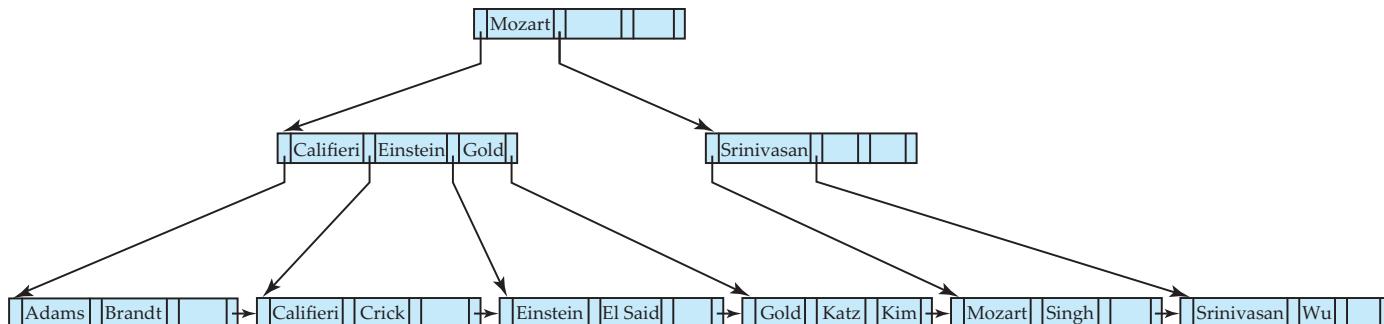
1. *C*=root
2. **while** (*C* is not a leaf node)
 1. Let *i* be least number s.t. $V \leq K_i$.
 2. **if** there is no such number *i* **then**
 3. Set *C* = last non-null pointer in *C*
 4. **else if** (*v* = *C.K_i*) Set *C* = *P_{i+1}*
 5. **else set** *C* = *C.P_i*
3. **if** for some *i*, *K_i* = *V* **then** return *C.P_i*
4. **else** return null /* no record with search-key value *v* exists. */





Queries on B⁺-Trees (Cont.)

- **Range queries** find all records with search key values in a given range
 - See book for details of **function** *findRange(lb, ub)* which returns set of all such records
 - Real implementations usually provide an iterator interface to fetch matching records one at a time, using a *next()* function





Queries on B⁺-Trees (Cont.)

- If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{n/2}(K) \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes
 - and n is typically around 100 (40 bytes per index entry).
- With 1 million search key values and $n = 100$
 - at most $\log_{50}(1,000,000) = 4$ nodes are accessed in a lookup traversal from root to leaf.
- Contrast this with a balanced binary tree with 1 million search key values
 - around 20 nodes are accessed in a lookup
 - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds



Non-Unique Keys

- If a search key a_i is not unique, create instead an index on a composite key (a_i, A_p) , which is unique
 - A_p could be a primary key, record ID, or any other attribute that guarantees uniqueness
- Search for $a_i = v$ can be implemented by a range search on composite key, with range $(v, -\infty)$ to $(v, +\infty)$
- But more I/O operations are needed to fetch the actual records
 - If the index is clustering, all accesses are sequential
 - If the index is non-clustering, each record access may need an I/O operation



Updates on B+-Trees: Insertion

Assume record already added to the file. Let

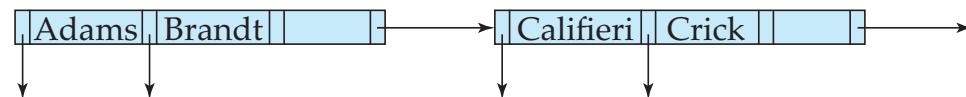
- pr be pointer to the record, and let
- v be the search key value of the record

1. Find the leaf node in which the search-key value would appear
 1. If there is room in the leaf node, insert (v, pr) pair in the leaf node
 2. Otherwise, split the node (along with the new (v, pr) entry) as discussed in the next slide, and propagate updates to parent nodes.



Updates on B⁺-Trees: Insertion (Cont.)

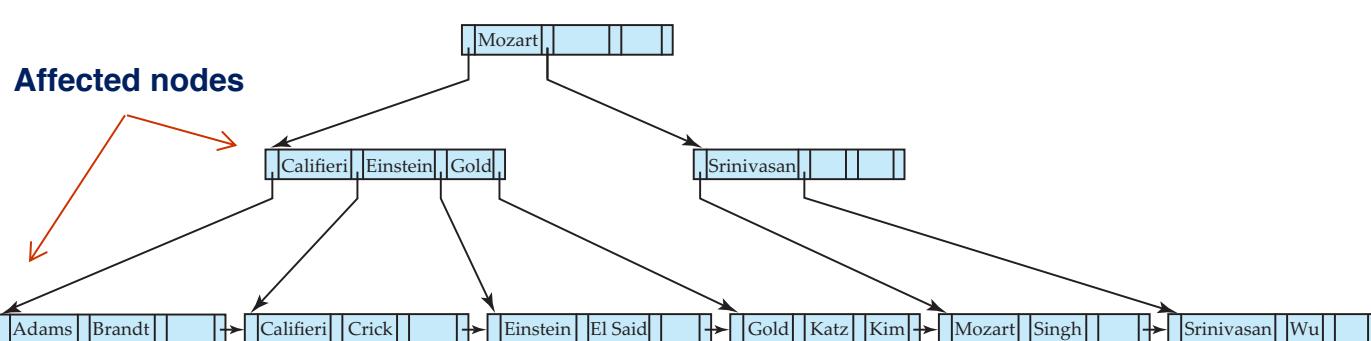
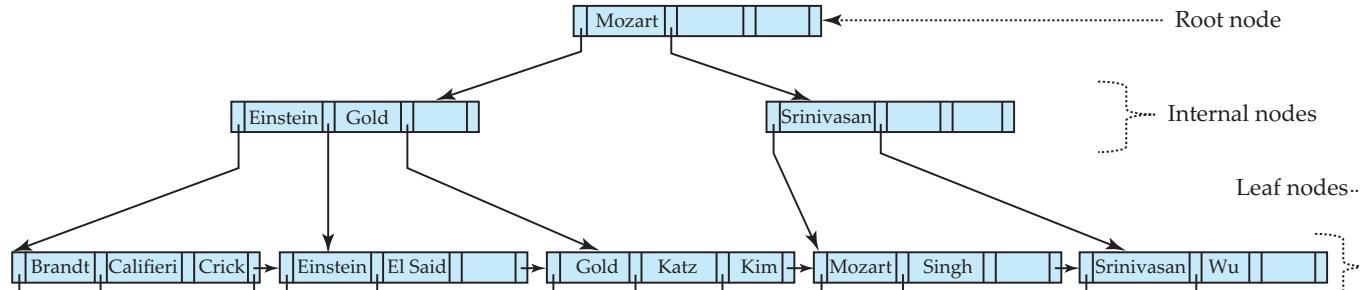
- Splitting a leaf node:
 - take the n (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
 - let the new node be p , and let k be the least key value in p . Insert (k, p) in the parent of the node being split.
 - If the parent is full, split it and **propagate** the split further up.
- Splitting of nodes proceeds upwards till a node that is not full is found.
- In the worst case the root node may be split increasing the height of the tree by 1.



Result of splitting node containing Brandt, Califieri and Crick on inserting Adams
Next step: insert entry with (Califieri, pointer-to-new-node) into parent



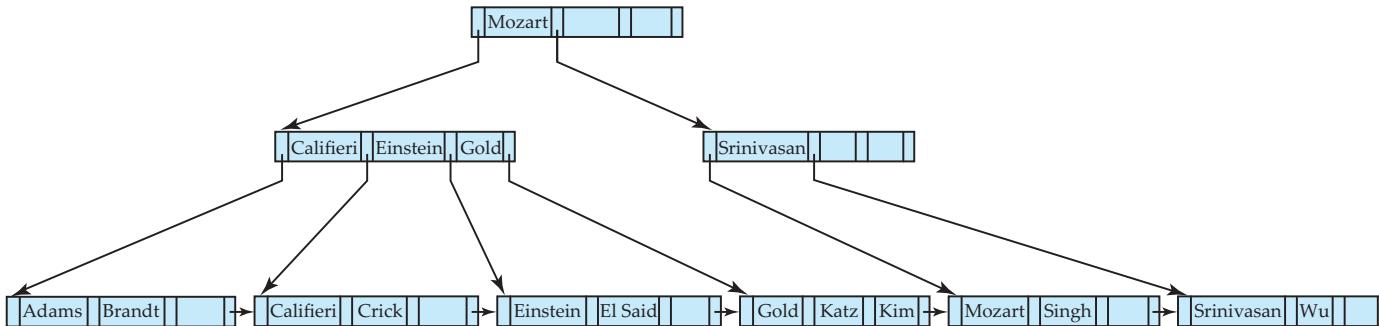
B+-Tree Insertion



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

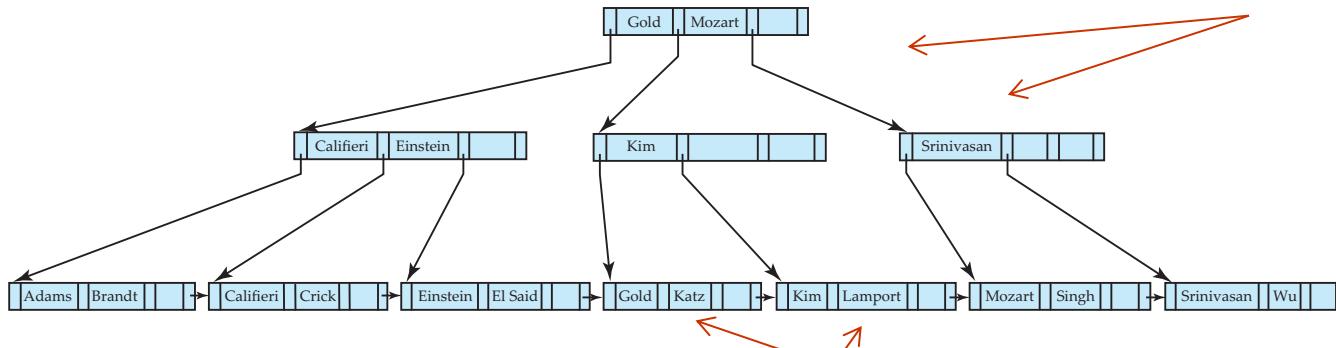


B⁺-Tree Insertion



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

Affected nodes

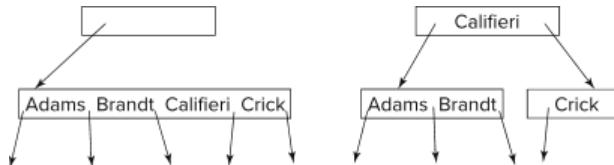


Affected nodes



Insertion in B⁺-Trees (Cont.)

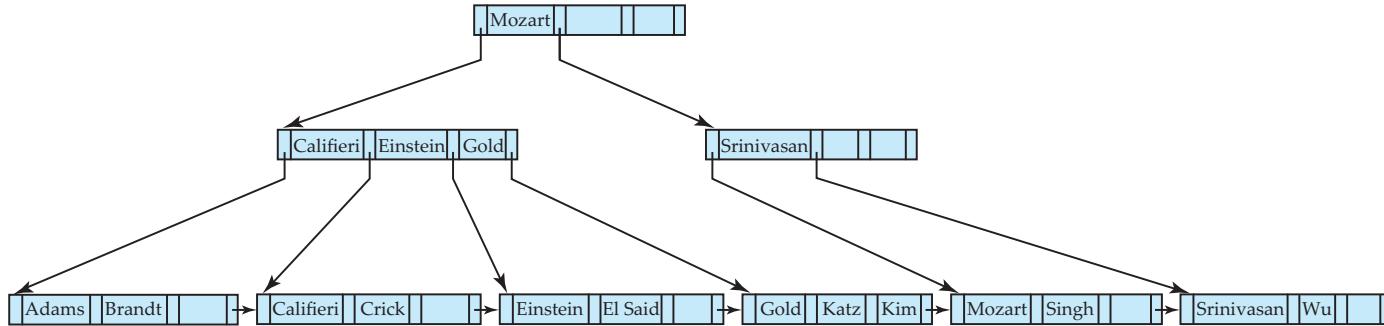
- Splitting a non-leaf node: when inserting (k,p) into an already full internal node N
 - Copy N to an in-memory area M with space for n+1 pointers and n keys
 - Insert (k,p) into M
 - Copy $P_1, K_1, \dots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ from M back into node N
 - Copy $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \dots, K_n, P_{n+1}$ from M into newly allocated node N'
 - Insert $(K_{\lceil n/2 \rceil}, N')$ into parent N
- Example



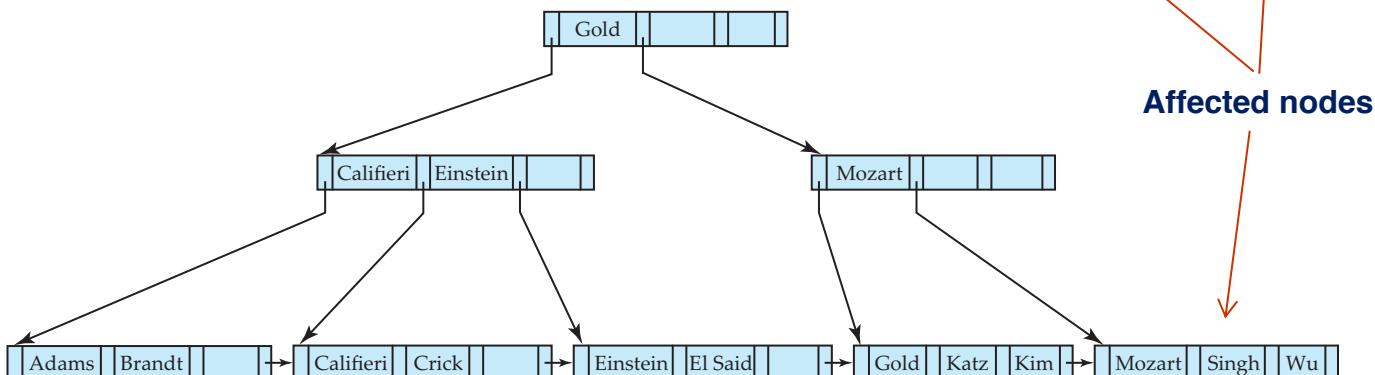
- **Read pseudocode in book!**



Examples of B+-Tree Deletion



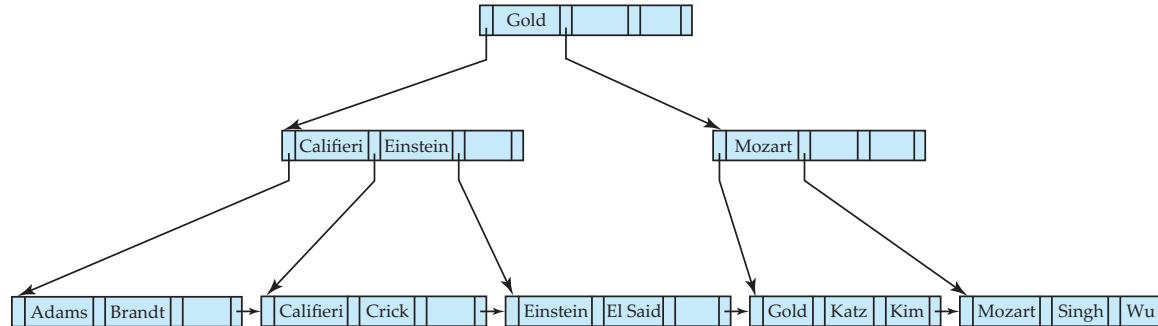
Before and after deleting “Srinivasan”



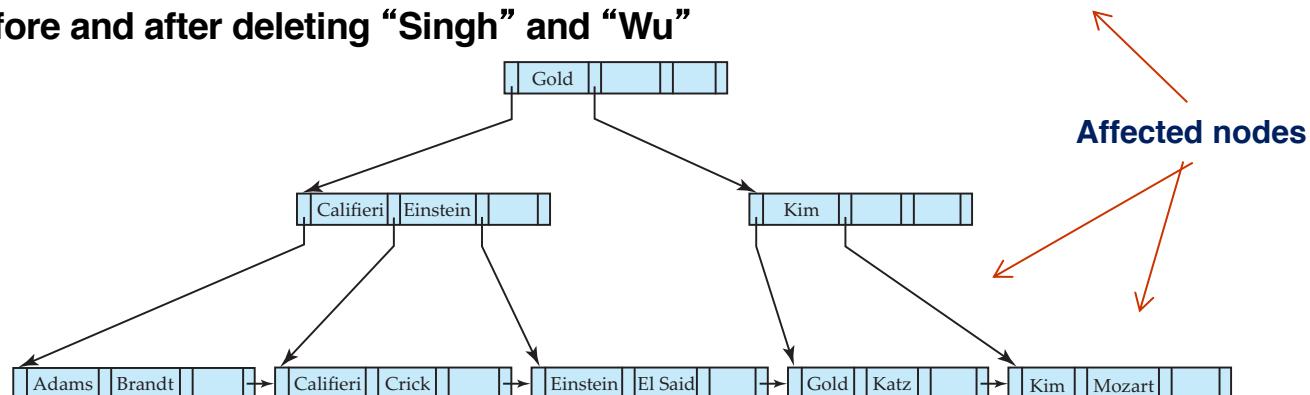
- Deleting “Srinivasan” causes **merging** of under-full leaves



Examples of B⁺-Tree Deletion (Cont.)



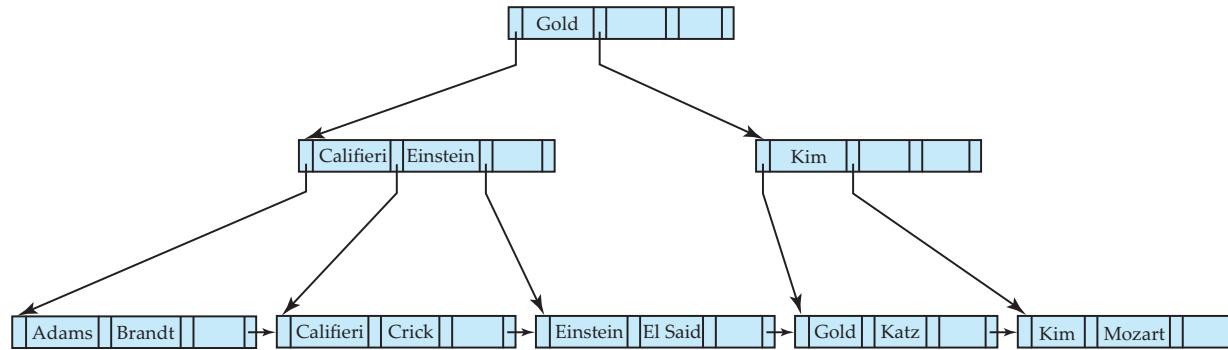
Before and after deleting “Singh” and “Wu”



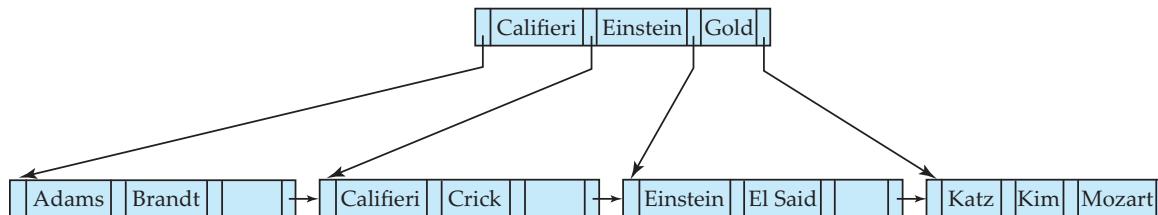
- Leaf containing Singh and Wu became underfull, and **borrowed a value** Kim from its left sibling
- Search-key value in the parent changes as a result



Example of B⁺-tree Deletion (Cont.)



Before and after deletion of “Gold”



- Node with Gold and Katz became underfull, and was merged with its sibling
- Parent node becomes underfull, and is merged with its sibling
 - Value separating two nodes (at the parent) is pulled down when merging
- Root node then has only one child, and is deleted



Updates on B⁺-Trees: Deletion

Assume record already deleted from file. Let V be the search key value of the record, and Pr be the pointer to the record.

- Remove (Pr, V) from the leaf node
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then ***merge siblings***:
 - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
 - Delete the pair (K_{i-1}, P_i) , where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.



Updates on B+-Trees: Deletion

- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then **redistribute pointers**:
 - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
 - Update the corresponding search-key value in the parent of the node.
- The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ or more pointers is found.
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.



Complexity of Updates

- Cost (in terms of number of I/O operations) of insertion and deletion of a single entry proportional to height of the tree
 - With K entries and maximum fanout of n , worst case complexity of insert/delete of an entry is $O(\log_{n/2}(K))$
- In practice, number of I/O operations is less:
 - Internal nodes tend to be in buffer
 - Splits/merges are rare, most insert/delete operations only affect a leaf node
- Average node occupancy depends on insertion order
 - 2/3rds with random, $1/2$ with insertion in sorted order



Non-Unique Search Keys

- Alternatives to scheme described earlier
 - Buckets on separate block (bad idea)
 - List of tuple pointers with each key
 - Extra code to handle long lists
 - Deletion of a tuple can be expensive if there are many duplicates on search key (why?)
 - Worst case complexity may be linear!
 - Low space overhead, no extra cost for queries
 - Make search key unique by adding a record-identifier
 - Extra storage overhead for keys
 - Simpler code for insertion/deletion
 - Widely used



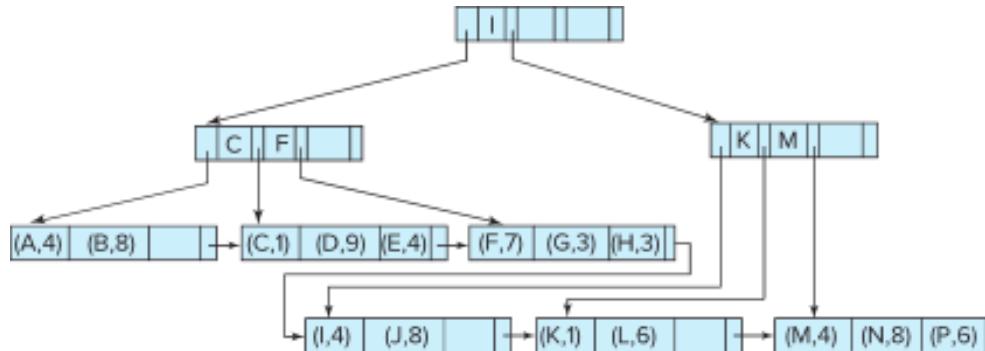
B⁺-Tree File Organization

- B⁺-Tree File Organization:
 - Leaf nodes in a B⁺-tree file organization store records, instead of pointers
 - Helps keep data records clustered even when there are insertions/deletions/updates
- Leaf nodes are still required to be half full
 - Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B⁺-tree index.



B+-Tree File Organization (Cont.)

- Example of B+-tree File Organization



- Good space utilization important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
 - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least $\lfloor 2n/3 \rfloor$ entries



Other Issues in Indexing

- **Record relocation and secondary indices**
 - If a record moves, all secondary indices that store record pointers have to be updated
 - Node splits in B⁺-tree file organizations become very expensive
 - *Solution:* use search key of B⁺-tree file organization instead of record pointer in secondary index
 - Add record-id if B⁺-tree file organization search key is non-unique
 - Extra traversal of file organization to locate record
 - Higher cost for queries, but node splits are cheap



Indexing Strings

- Variable length strings as keys
 - Variable fanout
 - Use space utilization as criterion for splitting, not number of pointers
- **Prefix compression**
 - Key values at internal nodes can be prefixes of full key
 - Keep enough characters to distinguish entries in the subtrees separated by the key value
 - E.g., “Silas” and “Silberschatz” can be separated by “Silb”
 - Keys in leaf node can be compressed by sharing common prefixes

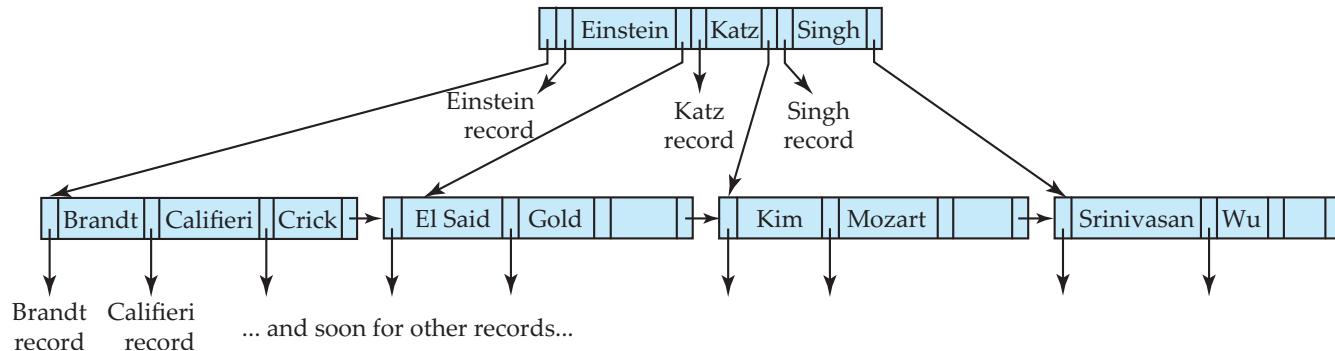


Bulk Loading and Bottom-Up Build

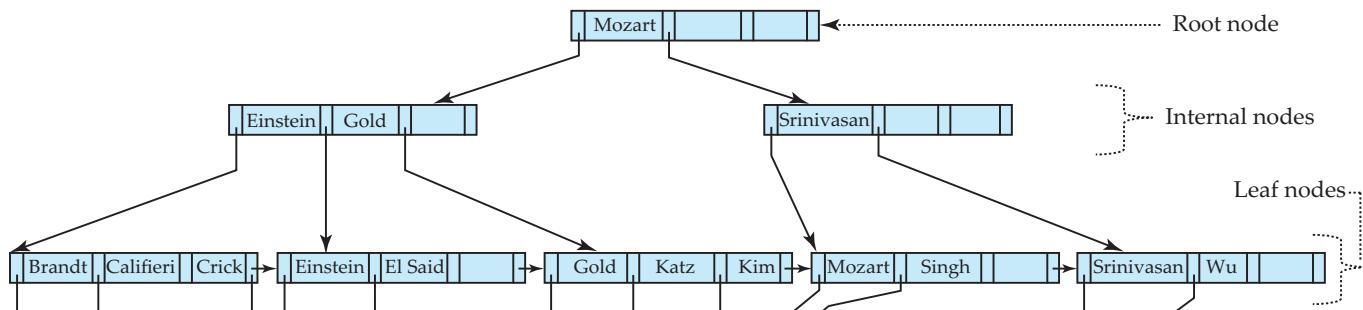
- Inserting entries one-at-a-time into a B⁺-tree requires ≥ 1 IO per entry
 - assuming leaf level does not fit in memory
 - can be very inefficient for loading a large number of entries at a time
(bulk loading)
- Efficient alternative 1:
 - sort entries first (using efficient external-memory sort algorithms discussed later in Section 12.4)
 - insert in sorted order
 - insertion will go to existing page (or cause a split)
 - much improved IO performance, but most leaf nodes half full
- Efficient alternative 2: **Bottom-up B⁺-tree construction**
 - As before sort entries
 - And then create tree layer-by-layer, starting with leaf level
 - details as an exercise
 - Implemented as part of bulk-load utility by most database systems



B-Tree Index File Example



B-tree (above) and B+-tree (below) on same data





Indexing on Flash

- Random I/O cost much lower on flash
 - 20 to 100 microseconds for read/write
- Writes are not in-place, and (eventually) require a more expensive erase
- Optimum page size therefore much smaller
- Bulk-loading still useful since it minimizes page erases
- Write-optimized tree structures (discussed later) have been adapted to minimize page writes for flash-optimized search trees



Indexing in Main Memory

- Random access in memory
 - Much cheaper than on disk/flash
 - But still expensive compared to cache read
 - Data structures that make best use of cache preferable
 - Binary search for a key value within a large B⁺-tree node results in many cache misses
- B⁺- trees with small nodes that fit in cache line are preferable to reduce cache misses
- Key idea: use large node size to optimize disk access, but structure data within a node using a tree with small node size, instead of using an array.



Hashing



Static Hashing

- A **bucket** is a unit of storage containing one or more entries (a bucket is typically a disk block).
 - we obtain the bucket of an entry from its search-key value using a **hash function**
- Hash function h is a function from the set of all search-key values K to the set of all bucket addresses B .
- Hash function is used to locate entries for access, insertion as well as deletion.
- Entries with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate an entry.
- In a **hash index**, buckets store entries with pointers to records
- In a **hash file-organization** buckets store records



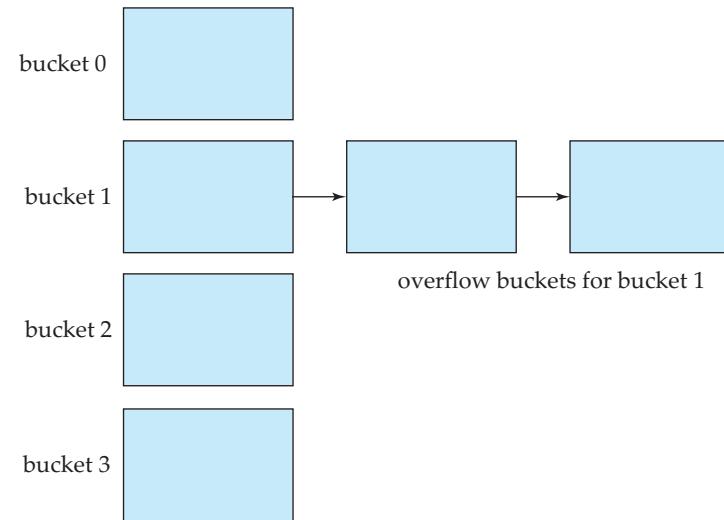
Handling of Bucket Overflows

- Bucket overflow can occur because of
 - Insufficient buckets
 - Skew in distribution of records. This can occur due to two reasons:
 - multiple records have same search-key value
 - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using **overflow buckets**.



Handling of Bucket Overflows (Cont.)

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed addressing** (also called **closed hashing** or **open hashing** depending on the book you use)
 - An alternative, called **open addressing** (also called **open hashing** or **closed hashing** depending on the book you use) which does not use overflow buckets, is not suitable for database applications.





Example of Hash File Organization

Hash file organization of *instructor* file, using *dept_name* as key.

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7



Deficiencies of Static Hashing

- In static hashing, function h maps search-key values to a fixed set of B of bucket addresses. Databases grow or shrink with time.
 - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
 - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
 - If database shrinks, again space will be wasted.
- One solution: periodic re-organization of the file with a new hash function
 - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically.



Dynamic Hashing

- Periodic rehashing
 - If number of entries in a hash table becomes (say) 1.5 times size of hash table,
 - create new hash table of size (say) 2 times the size of the previous hash table
 - Rehash all entries to new table
- Linear Hashing
 - Do rehashing in an incremental manner
- Extendable Hashing
 - Tailored to disk based hashing, with buckets shared by multiple hash values
 - Doubling of # of entries in hash table, without doubling # of buckets



Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
 - Hashing is generally better at retrieving records having a specified value of the key.
 - If range queries are common, ordered indices are to be preferred
- In practice:
 - PostgreSQL supports hash indices, but discourages use due to poor performance
 - Oracle supports static hash organization, but not hash indices
 - SQLServer supports only B⁺-trees



Multiple-Key Access

- Use multiple indices for certain types of queries.
- Example:

```
select ID  
from instructor  
where dept_name = "Finance" and salary = 80000
```

- Possible strategies for processing query using indices on single attributes:
 1. Use index on *dept_name* to find instructors with department name Finance; test *salary* = 80000
 2. Use index on *salary* to find instructors with a salary of \$80000; test *dept_name* = "Finance".
 3. Use *dept_name* index to find pointers to all records pertaining to the "Finance" department. Similarly use index on *salary*. Take intersection of both sets of pointers obtained.



Indices on Multiple Keys

- **Composite search keys** are search keys containing more than one attribute
 - E.g., $(dept_name, salary)$
- Lexicographic ordering: $(a_1, a_2) < (b_1, b_2)$ if either
 - $a_1 < b_1$, or
 - $a_1=b_1$ and $a_2 < b_2$



Indices on Multiple Attributes

Suppose we have an index on combined search-key
 $(dept_name, salary)$.

- With the **where** clause
 - where** $dept_name = \text{"Finance"} \text{ and } salary = 80000$
the index on $(dept_name, salary)$ can be used to fetch only records that satisfy both conditions.
 - Using separate indices is less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions.
- Can also efficiently handle
 - where** $dept_name = \text{"Finance"} \text{ and } salary < 80000$
- But cannot efficiently handle
 - where** $dept_name < \text{"Finance"} \text{ and } balance = 80000$
 - May fetch many records that satisfy the first but not the second condition



Other Features

- **Covering indices**
 - Add extra attributes to index so (some) queries can avoid fetching the actual records
 - Store extra attributes only at leaf
 - Why?
- Particularly useful for secondary indices
 - Why?



Creation of Indices

- Example
 - create index takes_pk on takes (ID,course_ID, year, semester, section)**
 - drop index takes_pk**
- Most database systems allow specification of type of index, and clustering.
- Indices on primary key created automatically by all databases
 - Why?
- Some database also create indices on foreign key attributes
 - Why might such an index be useful for this query:
 - $takes \bowtie \sigma_{name='Shankar'}(student)$
- Indices can greatly speed up lookups, but impose cost on updates
 - Index tuning assistants/wizards supported on several databases to help choose indices, based on query and update workload



Index Definition in SQL

- Create an index

```
create index <index-name> on <relation-name>  
<attribute-list>)
```

E.g.,: **create index b-index on branch(branch_name)**

- Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key is a candidate key.
 - Not really required if SQL **unique** integrity constraint is supported
- To drop an index

```
drop index <index-name>
```

- Most database systems allow specification of type of index, and clustering.



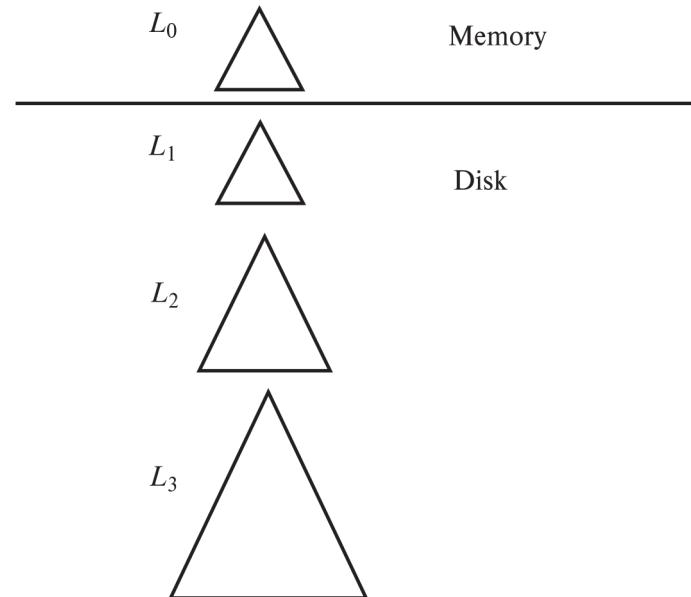
Write Optimized Indices

- Performance of B⁺-trees can be poor for write-intensive workloads
 - One I/O per leaf, assuming all internal nodes are in memory
 - With magnetic disks, < 100 inserts per second per disk
 - With flash memory, one page overwrite per insert
- Two approaches to reducing cost of writes
 - **Log-structured merge tree**
 - **Buffer tree**



Log Structured Merge (LSM) Tree

- Consider only inserts/queries for now
- Records inserted first into in-memory tree (L_0 tree)
- When in-memory tree is full, records moved to disk (L_1 tree)
 - B^+ -tree constructed using bottom-up build by merging existing L_1 tree with records from L_0 tree
- When L_1 tree exceeds some threshold, merge into L_2 tree
 - And so on for more levels
 - Size threshold for L_{i+1} tree is k times size threshold for L_i tree





LSM Tree (Cont.)

- Benefits of LSM approach
 - Inserts are done using only sequential I/O operations
 - Leaves are full, avoiding space wastage
 - Reduced number of I/O operations per record inserted as compared to normal B⁺-tree (up to some size)
- Drawback of LSM approach
 - Queries have to search multiple trees
 - Entire content of each level copied multiple times
- Stepped-merge index
 - Variant of LSM tree with multiple trees at each level
 - Reduces write cost compared to LSM tree
 - But queries are even more expensive
 - Bloom filters to avoid lookups in most trees
- Details are covered in Chapter 24



LSM Trees (Cont.)

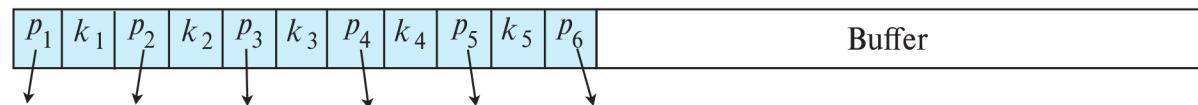
- Deletion handled by adding special “delete” entries
 - Lookups will find both original entry and the delete entry, and must return only those entries that do not have matching delete entry
 - When trees are merged, if we find a delete entry matching an original entry, both are dropped.
- Update handled using insert+delete
- LSM trees were introduced for disk-based indices
 - But useful to minimize erases with flash-based indices
 - The stepped-merge variant of LSM trees is used in many BigData storage systems
 - Google BigTable, Apache Cassandra, MongoDB
 - And more recently in SQLite4, LevelDB, and MyRocks storage engine of MySQL



Buffer Tree

- Alternative to LSM tree
- Key idea: each internal node of B⁺-tree has a buffer to store inserts
 - Inserts are moved to lower levels when buffer is full
 - With a large buffer, many records are moved to lower level each time
 - Per record I/O decreases correspondingly
- Benefits
 - Less overhead on queries
 - Can be used with any tree index structure
 - Used in PostgreSQL Generalized Search Tree (GiST) indices
- Drawback: more random I/O than LSM tree

Internal node





Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- Records in a relation are assumed to be numbered sequentially from, say, 0
 - Given a number n it must be easy to retrieve record n
 - Particularly easy if records are of fixed size
- Applicable on attributes that take on a relatively small number of distinct values
 - E.g., gender, country, state, ...
 - E.g., income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)
- A bitmap is simply an array of bits



Bitmap Indices (Cont.)

- In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute
 - Bitmap has as many bits as records
 - In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise
- Example

record number	<i>ID</i>	<i>gender</i>	<i>income_level</i>
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for <i>gender</i>		Bitmaps for <i>income_level</i>	
m	10010	L1	10100
f	01101	L2	01000
		L3	00001
		L4	00010
		L5	00000



Bitmap Indices (Cont.)

- Bitmap indices are useful for queries on multiple attributes
 - not particularly useful for single attribute queries
- Queries are answered using bitmap operations
 - Intersection (and)
 - Union (or)
- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
 - E.g., $100110 \text{ AND } 110011 = 100010$
 $100110 \text{ OR } 110011 = 110111$
 $\text{NOT } 100110 = 011001$
 - Males with income level L1: $10010 \text{ AND } 10100 = 10000$
 - Can then retrieve required tuples.
 - Counting number of matching tuples is even faster



Bitmap Indices (Cont.)

- Bitmap indices generally very small compared with relation size
 - E.g., if record is 100 bytes, space for a single bitmap is 1/800 of space used by relation.
 - If number of distinct attribute values is 8, bitmap is only 1% of relation size



Efficient Implementation of Bitmap Operations

- Bitmaps are packed into words; a single word and (a basic CPU instruction) computes and of 32 or 64 bits at once
 - E.g., 1-million-bit maps can be and-ed with just 31,250 instruction
- Counting number of 1s can be done fast by a trick:
 - Use each byte to index into a precomputed array of 256 elements each storing the count of 1s in the binary representation
 - Can use pairs of bytes to speed up further at a higher memory cost
 - Add up the retrieved counts
- Bitmaps can be used instead of Tuple-ID lists at leaf levels of B⁺-trees, for values that have a large number of matching records
 - Worthwhile if > 1/64 of the records have that value, assuming a tuple-id is 64 bits
 - Above technique merges benefits of bitmap and B⁺-tree indices



Spatial and Temporal Indices



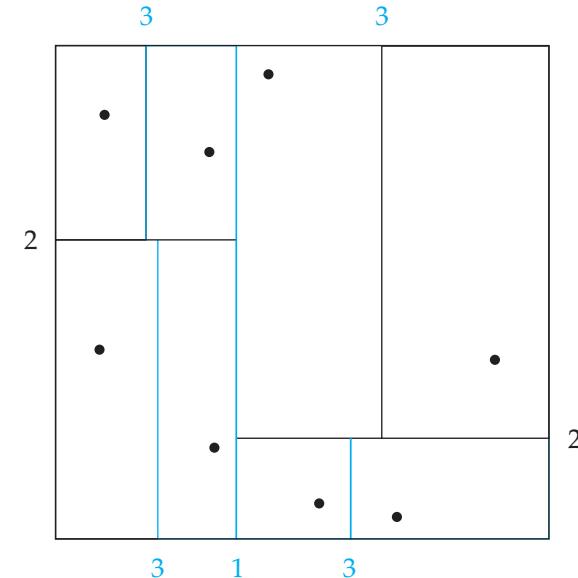
Spatial Data

- Databases can store data types such as lines, polygons, in addition to raster images
 - allows relational databases to store and retrieve spatial information
 - Queries can use spatial conditions (e.g. contains or overlaps).
 - queries can mix spatial and nonspatial conditions
- **Nearest neighbor queries**, given a point or an object, find the nearest object that satisfies given conditions.
- **Range queries** deal with spatial regions. e.g., ask for objects that lie partially or fully inside a specified region.
- Queries that compute intersections or **unions** of regions.
- **Spatial join** of two spatial relations with the location playing the role of join attribute.



Indexing of Spatial Data

- **k-d tree** - early structure used for indexing in multiple dimensions.
- Each level of a *k-d* tree partitions the space into two.
 - Choose one dimension for partitioning at the root level of the tree.
 - Choose another dimensions for partitioning in nodes at the next level and so on, cycling through the dimensions.
- In each node, approximately half of the points stored in the sub-tree fall on one side and half on the other.
- Partitioning stops when a node has less than a given number of points.

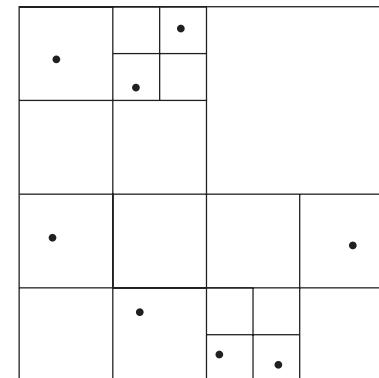


- The **k-d-B tree** extends the *k-d* tree to allow multiple child nodes for each internal node; well-suited for secondary storage.



Division of Space by Quadtrees

- Each node of a quadtree is associated with a rectangular region of space; the top node is associated with the entire target space.
- Each non-leaf nodes divides its region into four equal sized quadrants
 - correspondingly each such node has four child nodes corresponding to the four quadrants and so on
- Leaf nodes have between zero and some fixed maximum number of points (set to 1 in example).





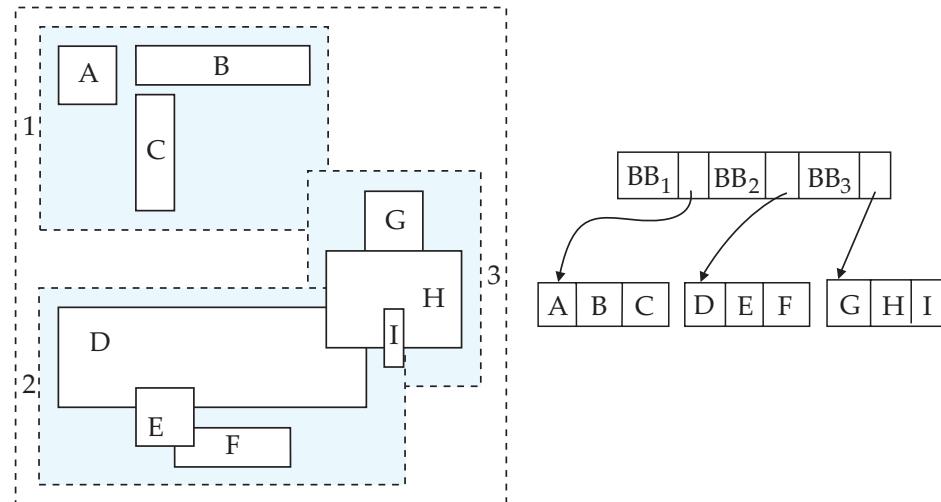
R-Trees

- **R-trees** are a N-dimensional extension of B+-trees, useful for indexing sets of rectangles and other polygons.
- Supported in many modern database systems, along with variants like R⁺-trees and R^{*}-trees.
- Basic idea: generalize the notion of a one-dimensional interval associated with each B+ -tree node to an N-dimensional interval, that is, an N-dimensional rectangle.
- Will consider only the two-dimensional case ($N = 2$)
 - generalization for $N > 2$ is straightforward, although R-trees work well only for relatively small N
- The **bounding box** of a node is a minimum sized rectangle that contains all the rectangles/polygons associated with the node
 - *Bounding boxes of children of a node are allowed to overlap*



Example R-Tree

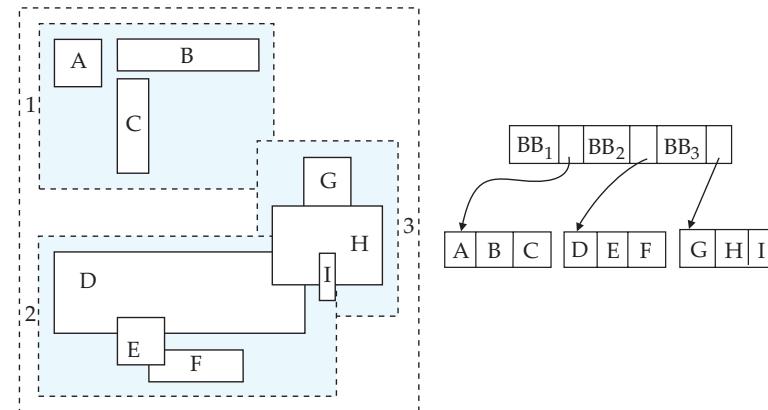
- A set of rectangles (solid line) and the bounding boxes (dashed line) of the nodes of an R-tree for the rectangles.
- The R-tree is shown on the right.





Search in R-Trees

- To find data items intersecting a given query point/region, do the following, starting from the root node:
 - If the node is a leaf node, output the data items whose keys intersect the given query point/region.
 - Else, for each child of the current node whose bounding box intersects the query point/region, recursively search the child
- Can be very inefficient in worst case since multiple paths may need to be searched, but works acceptably in practice.





Indexing Temporal Data

- Temporal data refers to data that has an associated time period (interval)
 - Example: a temporal version of the *course* relation

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>	<i>start</i>	<i>end</i>
BIO-101	Intro. to Biology	Biology	4	1985-01-01	9999-12-31
CS-201	Intro. to C	Comp. Sci.	4	1985-01-01	1999-01-01
CS-201	Intro. to Java	Comp. Sci.	4	1999-01-01	2010-01-01
CS-201	Intro. to Python	Comp. Sci.	4	2010-01-01	9999-12-31

- Time interval has a start and end time
 - End time set to infinity (or large date such as 9999-12-31) if a tuple is currently valid and its validity end time is not currently known
- Query may ask for all tuples that are valid at a point in time or during a time interval
 - Index on valid time period speeds up this task



Indexing Temporal Data (Cont.)

- To create a temporal index on attribute a :
 - Use spatial index, such as R-tree, with attribute a as one dimension, and time as another dimension
 - Valid time forms an interval in the time dimension
 - Tuples that are currently valid cause problems, since value is infinite or very large
 - Solution: store all current tuples (with end time as infinity) in a separate index, indexed on $(a, start\text{-}time)$
 - To find tuples valid at a point in time t in the current tuple index, search for tuples in the range $(a, 0)$ to (a, t)
- Temporal index on primary key can help enforce temporal primary key constraint

course_id	title	dept_name	credits	start	end
BIO-101	Intro. to Biology	Biology	4	1985-01-01	9999-12-31
CS-201	Intro. to C	Comp. Sci.	4	1985-01-01	1999-01-01
CS-201	Intro. to Java	Comp. Sci.	4	1999-01-01	2010-01-01
CS-201	Intro. to Python	Comp. Sci.	4	2010-01-01	9999-12-31