



# Chapter 10: Storage and File Structure

**Database System Concepts, 6<sup>th</sup> Ed.**

©Silberschatz, Korth and Sudarshan

See [www.db-book.com](http://www.db-book.com) for conditions on re-use



# Chapter 10: Storage and File Structure

- Overview of Physical Storage Media
- Magnetic Disks
- RAID
- Tertiary Storage
- File Organization
- Organization of Records in Files
- Data-Dictionary Storage



# Classification of Physical Storage Media

- Storage media are classified by:
  - Speed with which data can be accessed
  - Cost per unit of data
  - The medium's reliability
    - data loss on power failure or system crash
    - physical failure of the storage device
- In addition to the speed and cost, storage can be differentiated into:
  - **volatile storage:** loses contents when power is switched off
  - **non-volatile storage:**
    - 4 Contents persist even when power is switched off.
    - 4 Includes secondary and tertiary storage, as well as battery backed up main-memory.



# Physical Storage Media

- **Cache** – fastest and most costly form of storage; volatile; managed by the computer system hardware.
- **Main memory:**
  - fast access (10s to 100s of nanoseconds; 1 nanosecond =  $10^{-9}$  seconds)
  - generally too small (or too expensive) to store the entire database
    - 4 capacities of up to a few Gigabytes widely used currently
    - 4 Capacities have gone up and per-byte costs have decreased steadily and rapidly (roughly factor of 2 every 2 to 3 years)
  - **Volatile** — contents of main memory are usually lost if a power failure or system crash occurs.



# Physical Storage Media (Cont.)

- **Flash memory**
- Data survives power failure. Two types: *NAND* and *NOR* flash.
  - NAND flash has a much higher storage capacity for a given cost, and is widely used in embedded devices such as cameras, music players, cell phones, laptop computers and USB Keys
  - Data can be written at a location only once, but location can be erased and written to again
    - 4 Can support only a limited number (10K – 1M) of write/erase cycles.
    - 4 Erasing of memory has to be done to an entire bank of memory
  - Reads are roughly as fast as main memory
  - But writes are slow (few microseconds), erase is slower
  - Also used as a replacement for magnetic disks for storing moderate amounts of data called *solid-state drives*.
  - Increasingly used in server systems to improve performance by caching frequently used data, since it provides faster access than disk, with larger storage capacity than main memory (for a given cost).



# Physical Storage Media (Cont.)

- **Magnetic-disk**

- Data is stored on spinning disk, and read/written magnetically
- Primary medium for the long-term storage of data; typically stores entire database.
- Data must be moved from disk to main memory for access, and written back for storage
  - 4 Much slower access than main memory
- **direct-access** – possible to read data on disk in any order, unlike magnetic tape
- Capacities range up to roughly 1.5 TB as of 2009
  - 4 Much larger capacity and lesser cost/byte than main memory/flash memory
  - 4 Growing constantly and rapidly with technology improvements (factor of 2 to 3 every 2 years)
- Survives power failures and system crashes
  - 4 disk failure can destroy data, but is rare



# Physical Storage Media (Cont.)

- **Optical storage**
  - non-volatile, data is read optically from a spinning disk using a laser
  - CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms
  - Blu-ray disks: 27 GB to 54 GB
  - Write-one, read-many (WORM) optical disks used for archival storage (CD-R, DVD-R, DVD+R)
  - Multiple write versions also available (CD-RW, DVD-RW, DVD+RW, and DVD-RAM)
  - Reads and writes are slower than magnetic disk
  - **Juke-box** systems, with large numbers of removable disks, a few drives, and a mechanism for automatic loading/unloading of disks available for storing large volumes of data



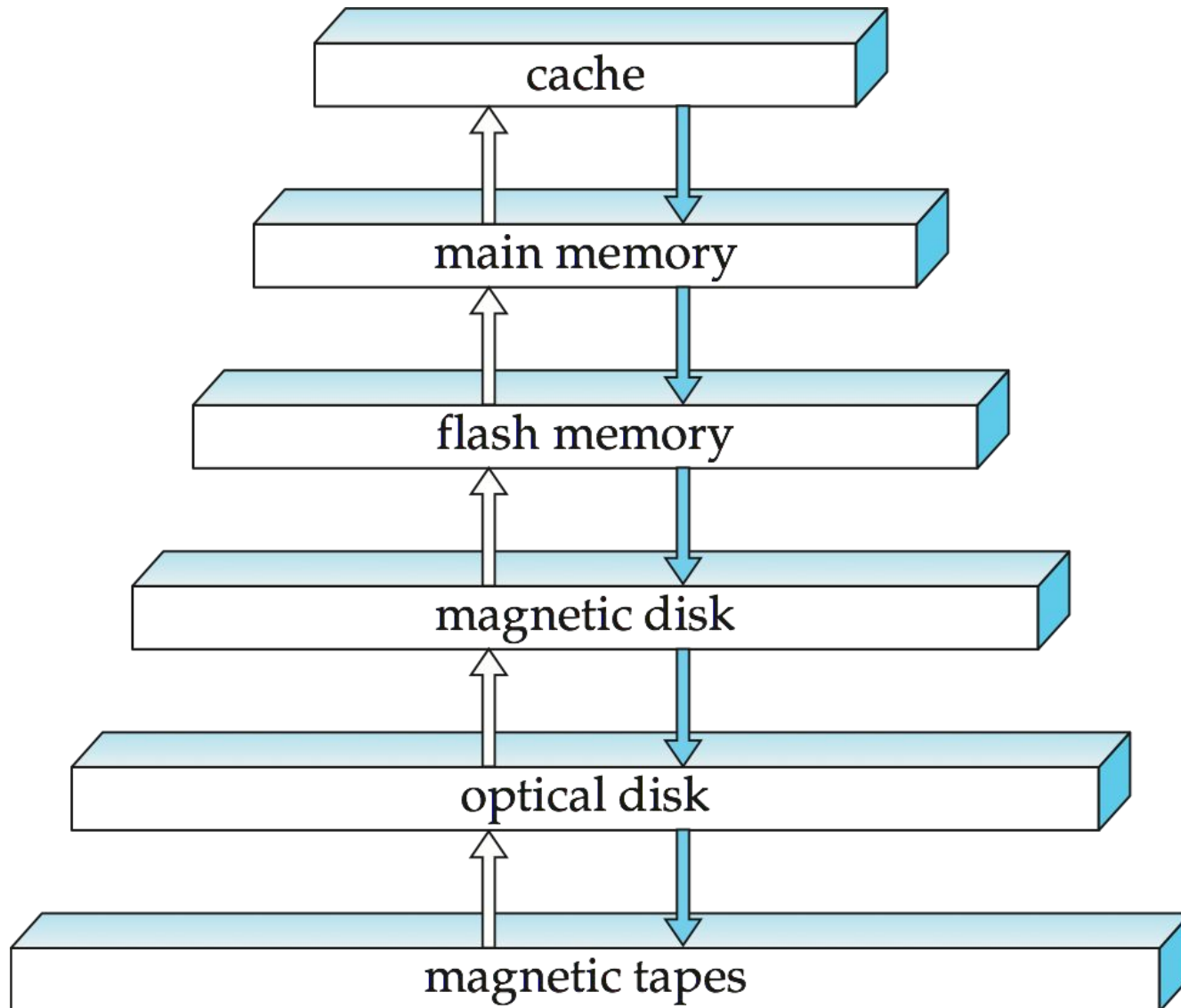
# Physical Storage Media (Cont.)

- **Tape storage**
  - non-volatile, used primarily for backup (to recover from disk failure), and for archival data
  - **sequential-access** – much slower than disk
  - very high capacity (40 to 300 GB tapes available)
  - tape can be removed from drive  $\Rightarrow$  storage costs much cheaper than disk, but drives are expensive
  - Tape jukeboxes available for storing massive amounts of data
    - 4 hundreds of terabytes (1 terabyte =  $10^9$  bytes) to even multiple **petabytes** (1 petabyte =  $10^{12}$  bytes)





# Storage Hierarchy



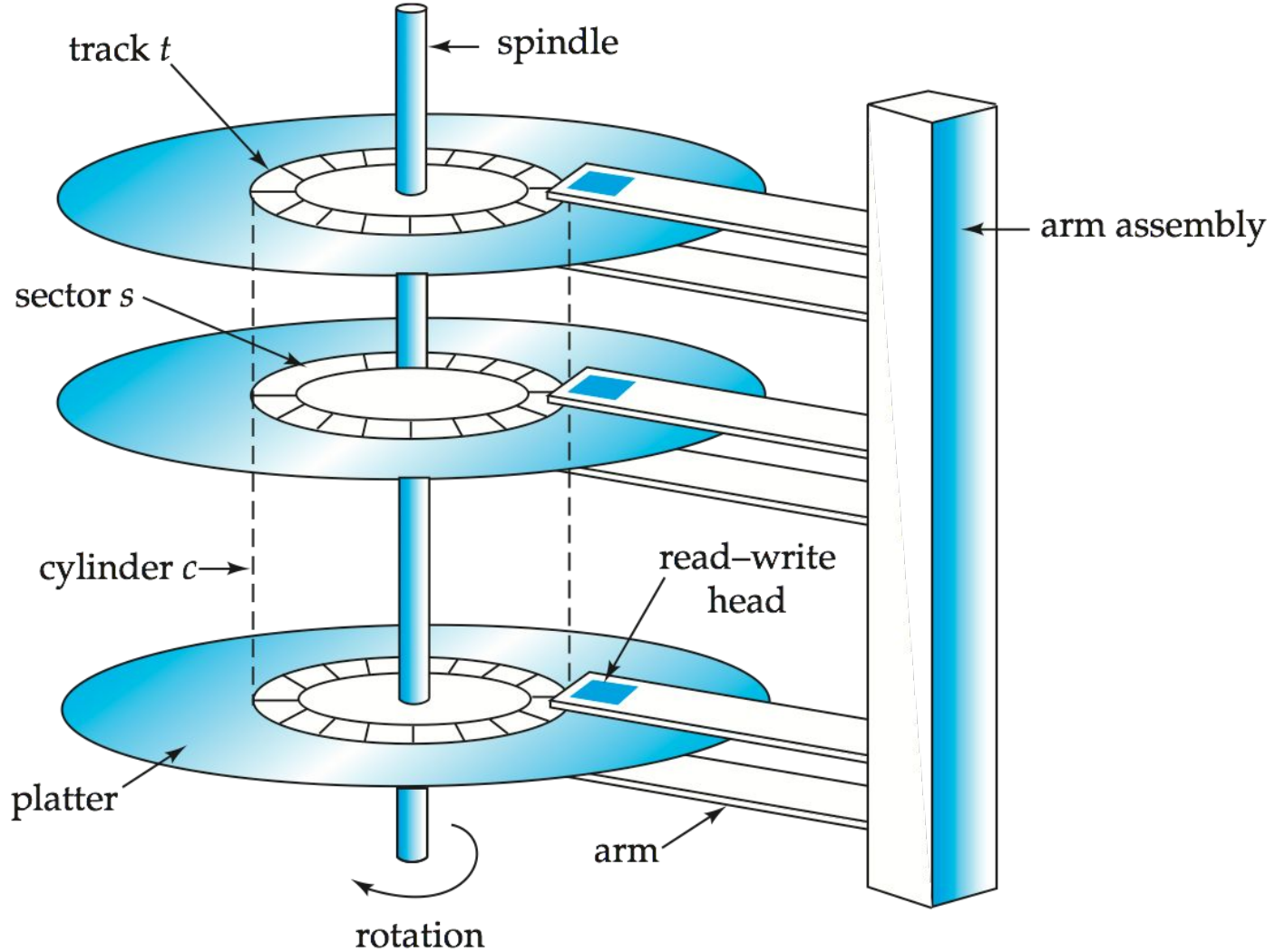


# Storage Hierarchy (Cont.)

- **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**
  - E.g. magnetic tape, optical storage



# Magnetic Hard Disk Mechanism



**NOTE:** Diagram is schematic, and simplifies the structure of actual disk drives



# Magnetic Disks

- **Read-write head**
  - Positioned very close to the platter surface (almost touching it)
  - Reads or writes magnetically encoded information.
- 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  $i^{\text{th}}$  track of all the platters

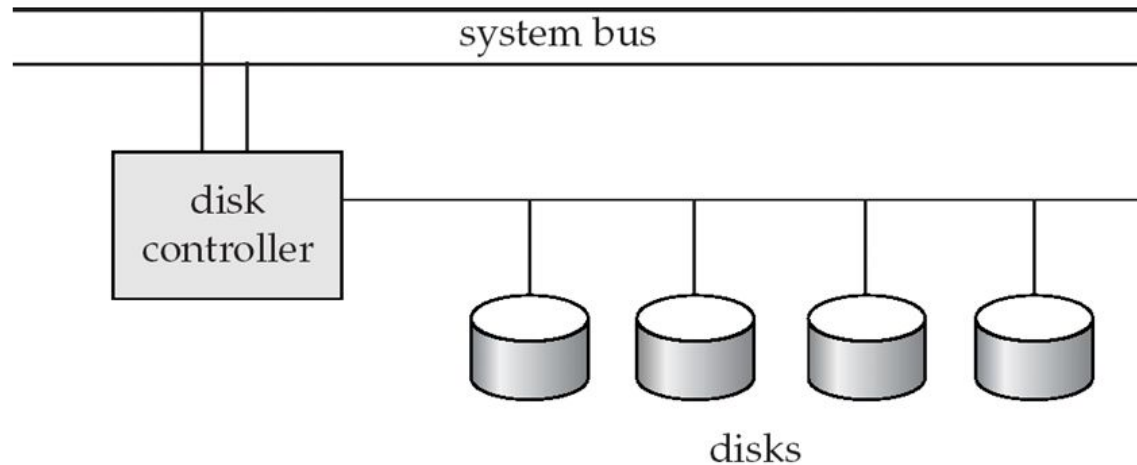


# Magnetic Disks (Cont.)

- **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
    - 4 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**



# Disk Subsystem



- Multiple disks connected to a computer system through a controller
  - Controllers functionality (checksum, bad sector remapping) often carried out by individual disks; reduces load on controller
- Disk interface standards families
  - **ATA** (AT adaptor) range of standards
  - **SATA** (Serial ATA)
  - **SCSI** (Small Computer System Interconnect) range of standards
  - **SAS** (Serial Attached SCSI)
  - The Fibre Channel interface
  - Several variants of each standard (different speeds and capabilities)



# Disk Subsystem

- While disks are usually connected directly by cables to the disk interface of the computer system, they can be situated remotely and connected by a high-speed network to the disk controller.
- In **Storage Area Networks (SAN)**, a large number of disks are connected by a high-speed network to a number of server computers. The disks are usually organized locally using a storage organization technique called **RAID** to give the servers a logical view of a very large and very reliable disk. .
- Remote access to disks across a storage area network means that disks can be shared by multiple computers that could run different parts of an application in parallel.
- Remote access also means that disks containing important data can be kept in a central server room where they can be monitored and maintained by system administrators, instead of being scattered in different parts of an organization.
- In **Network Attached Storage (NAS)** networked storage provides a file system interface using networked file system protocol, instead of providing a disk system interface



# 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.
    - 4 The **average seek time** is the average of the seek times, measured over a sequence of (uniformly distributed) random requests.
    - 4 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 4 to 10 milliseconds on typical disks
  - **Rotational latency** – time it takes for the sector to be accessed to appear under the head.
    - 4 Average latency is  $1/2$  of the worst case latency.
    - 4 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)
  - So, **Access time** (seek + rotational latency) ranges from 8 to 20 ms
- **Data-transfer rate** – the rate at which data can be retrieved from or stored to the disk.
  - 25 to 100 MB per second max rate, lower for inner tracks





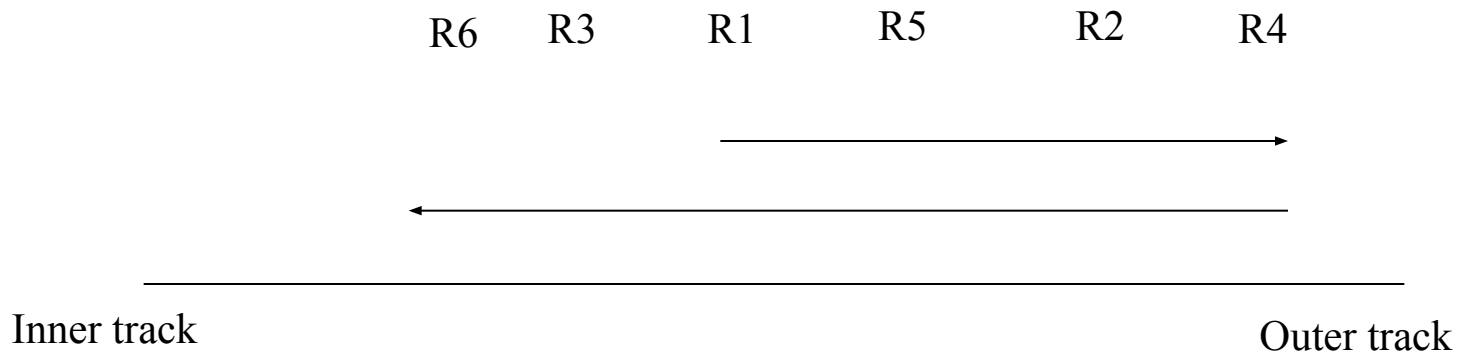
# Performance Measures (Cont.)

- Multiple disks may share a controller, so rate that controller can handle is also important
  - 4 E.g. SATA: 150 MB/sec, SATA-II 3Gb (300 MB/sec)
  - 4 Ultra 320 SCSI: 320 MB/s, SAS (3 to 6 Gb/sec)
  - 4 Fiber Channel (FC 2Gb or 4Gb interface): 256 to 512 MB/s
- **Mean time to failure (MTTF)** – A measure of reliability; 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 (about 57 to 136 years ! ) for a new disk (Vendors claim).
    - 4 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 (50 days)
  - MTTF decreases as disk ages



# Optimization of Disk-Block Access

- **Block** – a contiguous sequence of sectors from a single track
  - data is transferred between disk and main memory in blocks
  - sizes range from 512 bytes to several kilobytes
    - 4 Smaller blocks: more transfers from disk
    - 4 Larger blocks: more space wasted due to partially filled blocks
    - 4 Typical block sizes today range from 4 to 16 kilobytes
- **Disk-arm-scheduling** algorithms order pending accesses to tracks so that disk arm movement is minimized
  - **elevator algorithm:**





# RAID (Why?)

- The data-storage requirements of some applications (in particular Web, database, and multimedia applications) have been growing so fast that a large number of disks are needed to store their data, even though disk-drive capacities have been growing very fast.
- Having a large number of disks in a system presents opportunities for improving the rate at which data can be read or written, if the disks are operated in parallel.
- Again, several independent reads or writes can also be performed in parallel.
- Furthermore, this setup offers the potential for improving the reliability of data storage, because redundant information can be stored on multiple disks.
- Thus, failure of one disk does not lead to loss of data.
- A variety of disk-organization techniques, collectively called **redundant arrays of independent disks (RAID)**, have been proposed to achieve improved performance and reliability.



# 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
    - 4 **high capacity** and **high speed** by using multiple disks in parallel,
    - 4 **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
  - Originally a cost-effective alternative to large, expensive disks
    - I in RAID originally stood for “inexpensive”
    - Today RAIDs are used for their higher reliability and higher performance rate, rather than for economic reasons.
      - 4 The “I” is interpreted as independent



# Improvement of Reliability via Redundancy

- **Redundancy** – store extra information that that is not needed normally but can be used to rebuild information lost in a disk failure
- E.g., **Mirroring** (or **shadowing**): simplest (but most expensive)
  - Duplicate every disk. Logical disk consists of two physical disks.
  - Every write is carried out on both disks
    - 4 Reads can take place from either disk
  - If one disk in a pair fails, data still available in the other
    - 4 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  $500 \times 10^6$  hours (or 57,000 years!) for a mirrored pair of disks (ignoring dependent failure modes)



# Improvement in Performance via Parallelism

- Let's consider the benefit of parallel access to multiple disks.
  - With disk mirroring, the rate at which read requests can be handled is doubled, since read requests can be sent to either disk (as long as both disks in a pair are functional, as is almost always the case). The transfer rate of each read is the same as in a single-disk system, but the number of reads per unit time has doubled.
  - With multiple disks, Improve transfer rate by striping data across multiple disks can be achieved.
  - **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
- 4 Bit level striping is not used much any more



# Improvement in Performance via Parallelism

- **Block-level striping** – stripes blocks across multiple disks. It treats the array of disks as a single large disk and it gives blocks logical numbers (assume the block numbers start from 0).
  - with  $n$  disks, logical block  $i$  of a disk array goes to disk  $(i \bmod n) + 1$  ; it uses the  $\lfloor i/n \rfloor$  th physical block of the disk. Example: with 8 disks, logical block 0 is stored in physical block 0 of disk 1, while logical block 11 is stored in physical block 1 of disk 4.
  - When reading a large file, block-level striping fetches  $n$  blocks at a time in parallel from the  $n$  disks, giving a high data-transfer rate for *large reads*.
  - When a single block is read, the data-transfer rate is the same as on one disk, but the remaining  $n - 1$  disks are free to perform other actions.
- 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.



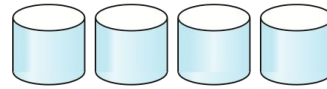
# RAID Levels

- **Mirroring** provides high reliability, but it is expensive.
- **Striping** provides high data-transfer rates, but does not improve reliability.
- Various alternative schemes aim to provide redundancy at lower cost by combining disk striping with “parity” bits.
- Different RAID organizations, or RAID levels, have differing cost, performance and reliability characteristics
- For all levels, the next figures depict four disks’ worth of data, and the extra disks depicted are used to store redundant information for failure recovery. (P = error-correcting bits, and C = a second copy of the data.)

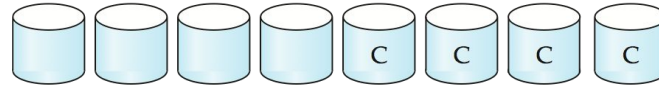




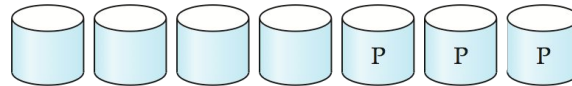
# Figure 10.03



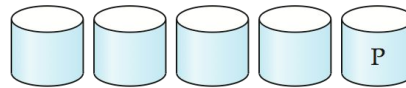
(a) RAID 0: nonredundant striping



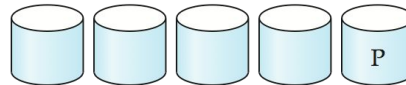
(b) RAID 1: mirrored disks



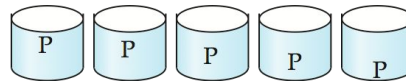
(c) RAID 2: memory-style error-correcting codes



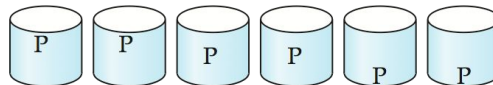
(d) RAID 3: bit-interleaved parity



(e) RAID 4: block-interleaved parity



(f) RAID 5: block-interleaved distributed parity

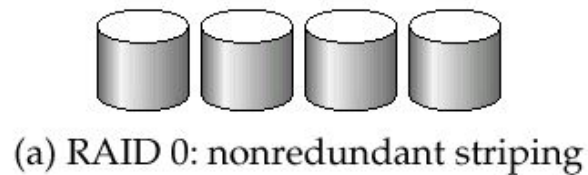


(g) RAID 6: P + Q redundancy



# RAID Levels

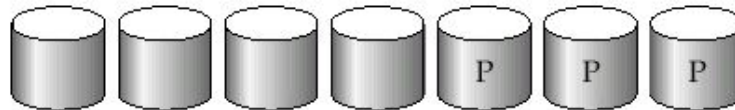
- **RAID Level 0:** Block striping; non-redundant.
  - Refers to disk arrays with striping at the level of blocks, but without any redundancy (such as mirroring or parity bits)
  - Used in high-performance applications where data loss is not critical.
- **RAID Level 1:** Mirrored disks with block striping
  - Offers best performance but most costly
  - Popular for applications such as storing log files in a database system.
  - Some vendors use the term **RAID level 1+0** or **RAID level 10** to refer to mirroring with striping, and use the term **RAID level 1** to refer to mirroring without striping.





# RAID Levels (Cont.)

- **RAID Level 2: Memory-Style Error-Correcting-Codes (ECC)** organization, employs parity bits with bit striping.
  - Requires three disks' overhead for four disks of data
- **RAID Level 3: Bit-Interleaved Parity**
  - A single parity bit is enough for error correction, not just detection, since disk controllers know whether a sector has been read correctly and thus which disk has failed
  - If one of the sectors gets damaged, for each bit in the sector, the system can figure out whether it is a 1 or a 0 by computing the parity of the corresponding bits from sectors in the other disks. If the parity of the remaining bits is equal to the stored parity, the missing bit is 0; otherwise, it is 1.



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



(d) RAID 3: bit-interleaved parity



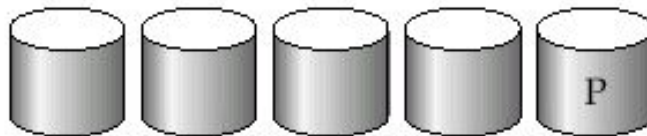
# RAID Levels (Cont.)

- **RAID Level 3: Bit-Interleaved Parity (Cont.)**
  - Subsumes Level 2 (provides all its benefits, at lower cost: only a one-disk overhead).
  - two benefits over level 1. It needs only one parity disk for several regular disks, whereas level 1 needs one mirror disk for every disk, and thus level 3 reduces the storage overhead.
  - Since reads and writes of a byte are spread out over multiple disks, with  $N$ -way striping of data, the transfer rate for reading or writing a single block is  $N$  times faster than a RAID level 1 organization using  $N$ -way striping.
  - On the other hand, RAID level 3 supports a lower number of I/O operations per second, since every disk has to participate in every I/O request.



## RAID Levels (Cont.)

- **RAID Level 4: Block-Interleaved Parity**; uses block-level striping, and keeps a parity block on a separate disk for corresponding blocks from  $N$  other disks.
  - When writing data block, corresponding block of parity bits must also be computed and written to parity disk
  - If one of the disks fails, the parity block can be used with the corresponding blocks from the other disks to restore the blocks of the failed disk.
  - A block read accesses only one disk, allowing other requests to be processed by the other disks. Thus, the data-transfer rate for each access is slower, but multiple read accesses can proceed in parallel, leading to a higher overall I/O rate.



(e) RAID 4: block-interleaved parity



# RAID Levels (Cont.)

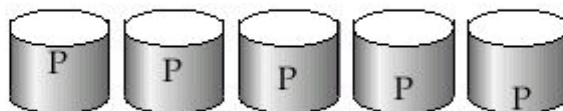
- **RAID Level 4 (Cont.)**

- The transfer rates for large reads is high, since all the disks can be read in parallel; large writes also have high transfer rates, since the data and parity can be written in parallel.
- Small independent writes, on the other hand, cannot be performed in parallel. A write of a block has to access the regular disk + the parity disk, since the parity block has to be updated. Parity block becomes a bottleneck for independent block writes
- Before writing a block, parity data must be computed
  - 4 both the old value of the parity block and the old value of the block being written have to be read for the new parity to be computed.
  - 4 Thus, a single write requires four disk accesses: 2 block reads + 2 block writes



## RAID Levels (Cont.)

- **RAID Level 5: Block-Interleaved Distributed Parity**; improves on level 4 by partitioning data and parity among all  $N + 1$  disks, rather than storing data in  $N$  disks and parity in 1 disk.
  - For each set of  $N$  logical blocks, one of the disks stores the parity, and the other  $N$  disks store the blocks.
  - All disks can participate in satisfying read requests, unlike RAID level 4, where the parity disk cannot participate, so level 5 increases the total number of requests that can be met in a given amount of time.



(f) RAID 5: block-interleaved distributed parity

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

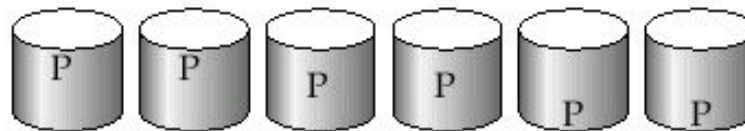


# RAID Levels (Cont.)

- **RAID Level 5 (Cont.)**

- Higher I/O rates than Level 4.
  - 4 Block writes occur in parallel if the blocks and their parity blocks are on different disks.
- Subsumes Level 4: provides same benefits, but avoids bottleneck of parity disk.

- **RAID Level 6: P+Q Redundancy** scheme; similar to Level 5, but stores extra redundant information to guard against multiple disk failures.
  - 2 bits of redundant data are stored for every 4 bits of data—unlike 1 parity bit in level 5—and the system can tolerate two disk failures.
  - Better reliability than Level 5 at a higher cost; not used as widely.



(g) RAID 6: P + Q redundancy

- You can also visit:
  - <https://www.javatpoint.com/dbms-raid>





# Choice of RAID Level

- Factors to be taken into account in choosing RAID level
  - **Monetary cost** of extra disk-storage requirements.
  - **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
- The time to rebuild the data of a failed disk can be significant, and it varies with the RAID level that is used.
- Rebuilding is easiest for RAID level 1, since data can be copied from another disk; for the other levels, we need to access all the other disks in the array to rebuild data of a failed disk.
- The **rebuild performance** of a RAID system may be an important factor if continuous availability of data is required, as it is in high-performance database systems.
- Furthermore, since rebuild time can form a significant part of the repair time, rebuild performance also influences the mean time to data loss.



# Choice of RAID Level

- RAID 0 is used in high-performance applications where data safety is not important
  - E.g. data can be recovered quickly from other sources
- Level 2 and 4 are subsumed by 3 and 5.
- Level 3 is not used anymore since bit-striping forces single block reads to access all disks, wasting disk arm movement, which block striping (level 5) avoids.
- Level 6 is not supported currently by many RAID implementations, but it offers better reliability than level 5 and can be used in applications where data safety is very important.
- The choice between RAID level 1 and level 5 is harder to make.
- RAID level 5 has a lower storage overhead than level 1, but has a higher time overhead for writes. For applications where data are read frequently, and written rarely, and uses large amounts of data, level 5 is the preferred choice.



# 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 preferred for moderate storage and high I/O requirements.
  - RAID level 1 is popular for applications such as storage of log files in a database system, since it offers the best write performance.
- Level 1 had higher storage cost than level 5
  - disk drive capacities increasing rapidly (50%/year) whereas disk access times have decreased much less (x 3 in 10 years)
  - I/O requirements have increased greatly, e.g. for Web servers
  - When enough disks have been bought to satisfy required rate of I/O, they often have spare storage capacity
    - 4 so there is often no extra monetary cost for Level 1!



# Choice of RAID Level (Cont.)

- RAID system designers have to make several other decisions as well.
- For example, how many disks should there be in an array?
  - If there are more disks in an array, data-transfer rates are higher, but the system will be more expensive.
- How many bits should be protected by each parity bit?
  - If there are more bits protected by a parity bit, the space overhead due to parity bits is lower, but there is an increased chance that a second disk will fail before the first failed disk is repaired, and that will result in data loss.



# 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
    - 4 E.g. failure after writing one block but before writing the second in a mirrored system
    - 4 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 detected 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



# Flash Storage

- NOR flash vs NAND flash
- NAND flash
  - used widely for storage, since it is much cheaper than NOR flash
  - requires page-at-a-time read (page: 512 bytes to 4 KB)
  - transfer rate around 20 MB/sec
  - **solid state disks**: use multiple flash storage devices to provide higher transfer rate of 100 to 200 MB/sec
  - erase is very slow (1 to 2 millisecs)
    - 4 erase block contains multiple pages
    - 4 **remapping** of logical page addresses to physical page addresses avoids waiting for erase
      - **translation table** tracks mapping
        - » also stored in a label field of flash page
      - remapping carried out by **flash translation layer**
    - 4 after 100,000 to 1,000,000 erases, erase block becomes unreliable and cannot be used
      - **wear leveling**



# Optical Disks

- Compact disk-read only memory (CD-ROM)
  - Removable disks, 640 MB per disk
  - Seek time about 100 msec (optical read head is heavier and slower)
  - Higher latency (3000 RPM) and lower data-transfer rates (3-6 MB/s) compared to magnetic disks
- Digital Video Disk (DVD)
  - DVD-5 holds 4.7 GB , and DVD-9 holds 8.5 GB
  - DVD-10 and DVD-18 are double sided formats with capacities of 9.4 GB and 17 GB
  - Blu-ray DVD: 27 GB (54 GB for double sided disk)
  - Slow seek time, for same reasons as CD-ROM
- Record once versions (CD-R and DVD-R) are popular
  - data can only be written once, and cannot be erased.
  - high capacity and long lifetime; used for archival storage
  - Multi-write versions (CD-RW, DVD-RW, DVD+RW and DVD-RAM) also available





# Magnetic Tapes

- Hold large volumes of data and provide high transfer rates
  - Few GB for DAT (Digital Audio Tape) format, 10-40 GB with DLT (Digital Linear Tape) format, 100 GB+ with Ultrium format, and 330 GB with Ampex helical scan format
  - Transfer rates from few to 10s of MB/s
- Tapes are cheap, but cost of drives is very high
- Very slow access time in comparison to magnetic and optical disks
  - limited to sequential access.
  - Some formats (Accelis) provide faster seek (10s of seconds) at cost of lower capacity
- Used mainly for backup, for storage of infrequently used information, and as an off-line medium for transferring information from one system to another.
- Tape jukeboxes used for very large capacity storage
  - Multiple petabytes ( $10^{15}$  bytes)



# **File Organization, Record Organization and Storage Access**



# File Organization

- A database is mapped into a number of different files that are maintained by the underlying operating system. These files reside permanently on disks.
- A **file** is organized logically as a sequence of records. These records are mapped onto disk blocks.
- Each file is logically partitioned into fixed-length storage units called **blocks**, which are the units of both storage allocation and data transfer.
- Most databases use block sizes of 4 to 8 kilobytes by default, but many databases allow the block size to be specified when a database instance is created.
- A block may contain several records. We shall assume that
  - *no record is larger than a block*
    - 4 realistic for most data-processing applications, such as our university database (exception images, movies)
  - *each record is entirely contained in a single block*
    - 4 this restriction simplifies and speeds up access to data items



# File Organization

- In a relational database, tuples of distinct relations are generally of different sizes.
- One approach (**fixed-length records**): easiest to implement
  - assume record size is fixed
  - each file has records of one particular type only
  - different files are used for different relations
- An alternative approach (**variable-length record**)
  - structure the files so that they can accommodate multiple lengths for records
- However, files of fixed-length records are easier to implement than are files of variable-length records.



# Fixed-Length Records

- Each record of *instructor* file in *university* database is defined (in pseudocode) as:

```
type instructor = record
    ID varchar (5);
    name varchar(20);
    dept_name varchar (20);
    salary numeric (8,2);
end
```

- Assume that each character occupies 1 byte and that numeric (8,2) occupies 8 bytes.
- Suppose that instead of allocating a variable amount of bytes for the attributes *ID*, *name*, and *dept\_name*, we allocate the maximum number of bytes that each attribute can hold. Then, the *instructor* record is 53 bytes long.
- A simple approach is to use the first 53 bytes for the first record, the next 53 bytes for the second record, and so on.



# Fixed-Length Records

- However, there are **two problems** with this simple approach:
- 1. Unless the block size happens to be a multiple of 53 (which is unlikely), some records will cross block boundaries. It would thus require two block accesses to read or write such a record.
- 2. It is difficult to delete a record from this structure. The space occupied by the record to be deleted must be filled with some other record of the file, or we must have a way of marking deleted records so that they can be ignored.
- To avoid the first problem, we allocate only as many records to a block as would fit entirely in the block (can be computed by  $\text{block size} / \text{record size}$ ), and discarding the fractional part.



# 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
  - 4 Modification: do not allow records to cross block boundaries

- 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 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



## Deleting record 3 and compacting

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





## Deleting record 3 and moving last record

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



# Free Lists

- We need to introduce an additional structure.
- At the beginning of the file, we allocate a certain number of bytes as a **file header**. The header will contain a variety of information about the file.
- For now, all we need to store there is the address of the first deleted record. We use this first record to store the address of the second available record, and so on.
- Can think of these stored addresses as **pointers** since they “point” to the location of a record.
- More space efficient representation: reuse space for normal attributes of free records to store pointers. (No pointers stored in in-use records.)

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





# Free Lists

- On insertion of a new record, we use the record pointed to by the header. We change the header pointer to point to the next available record. If no space is available, we add the new record to the end of the file.
- Insertion and deletion for files of fixed-length records are simple to implement, because the space made available by a deleted record is exactly the space needed to insert a record.
- If we allow records of variable length in a file, this match no longer holds. An inserted record may not fit in the space left free by a deleted record, or it may fill only part of that space.



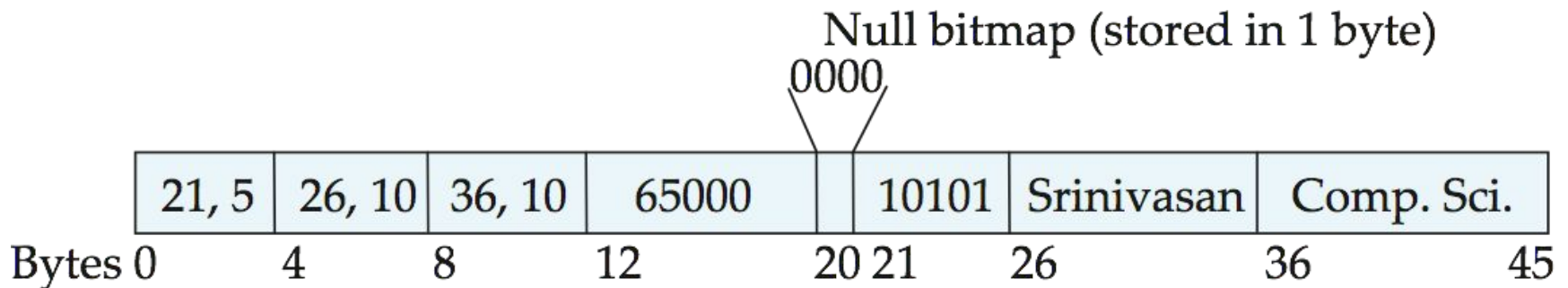
# Variable-Length Records

- Variable-length records arise in database systems in several ways:
  - Storage of multiple record types in a file (multitable clustering file)
  - Record types that allow variable lengths for one or more fields such as strings (**varchar**)
  - Record types that allow repeating fields, such as arrays or multisets.
- Different techniques for implementing variable-length records exist. Two different problems must be solved by any such technique:
  - 1. How to represent a single record in such a way that individual attributes can be extracted easily, even if they are of variable length
  - 2. How to store variable-length records within a block, such that records in a block can be extracted easily.
- The representation of a record with variable-length attributes typically has two parts: an initial part with fixed length attributes, followed by data for variable length attributes.



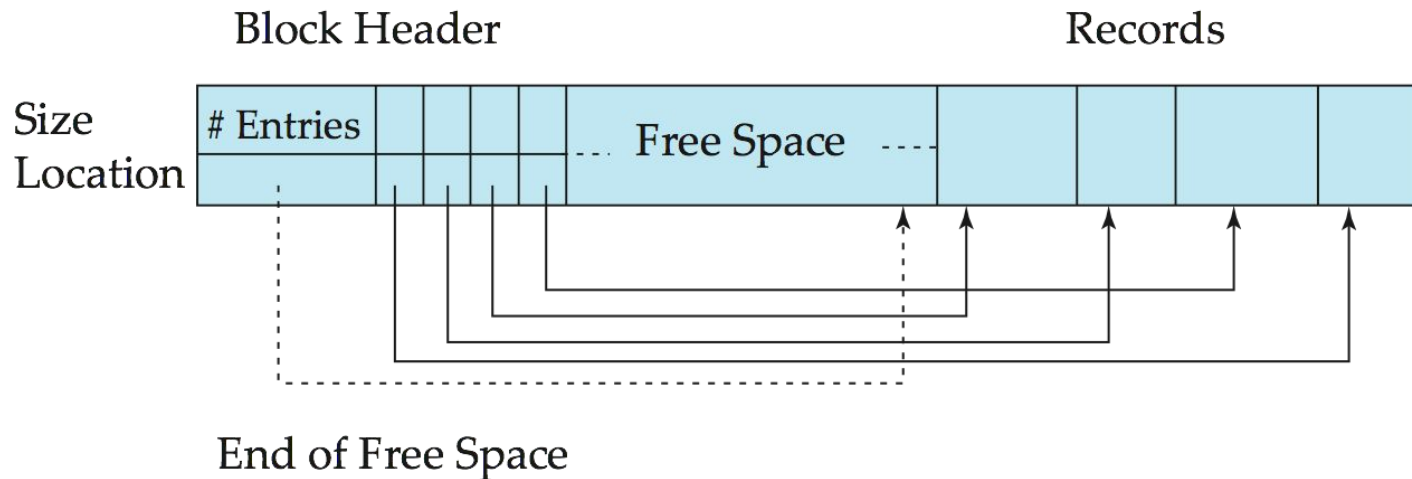
# Variable-Length Records

- The values for variable-sized attributes are stored consecutively, after the initial fixed-length (numeric values, dates, or fixed length character strings) part of the record.
- Variable length attributes represented by fixed size (**offset, length**), with actual data where *offset* denotes where the data for that attribute begins within the record, and *length* is the length in bytes of the variable-sized attribute.
- Null values represented by null-value bitmap. If the *salary* were null, the fourth bit of the bit map would be set to 1, and the *salary* value stored in bytes 12 through 19 would be ignored.
- Since the record has four attributes, the null bitmap for this record fits in 1 byte, although more bytes may be required with more attributes.





# Variable-Length Records: Slotted Page Structure



- Storing variable-length records in a block. The **slotted-page** structure is commonly used for organizing records within a block.
- A header at the beginning of each block, containing the following information:
  - The number of record entries in the header
  - The end of free space in the block
  - An array whose entries contain the location and size of each record
- The actual records are allocated *contiguously* in the block, starting from the end of the block. The free space in the block is contiguous, between the final entry in the header array, and the first record.





# Variable-Length Records: Slotted Page Structure

- If a record is inserted, space is allocated for it at the end of free space, and an entry containing its size and location is added to the header.
- If a record is deleted, the space that it occupies is freed, and its entry is set to *deleted* (its size is set to  $-1$ , for example). Further, the records in the block before the deleted record are moved, so that the free space created by the deletion gets occupied, and all free space is again between the final entry in the header array and the first record.
- The end-of-free-space pointer in the header is appropriately updated as well.
- Records can be grown or shrunk by similar techniques, as long as there is space in the block. The cost of moving the records is not too high, since the size of a block is limited: around 4 to 8 kilobytes.
- The slotted-page structure requires that there be no pointers that point directly to records. Instead, pointers must point to the entry in the header that contains the actual location of the record.
- This level of indirection allows records to be moved to prevent fragmentation of space inside a block, while supporting indirect pointers to the record.



# Organization of Records in Files

- So far, we have studied how records are represented in a file structure. A relation is a set of records. Given a set of records, the next question is how to organize them in a file.
- Several of the possible ways of organizing records in files are:
- **Heap** – Any record can be placed anywhere in the file where there is space for the record.
  - There is no ordering of records.
  - Typically, there is a single file for each relation.
- **Sequential** – store records in sequential order, based on the value of the “*search key*” of each record.
- Records of each relation are generally stored in a separate file. In a **multitable clustering file organization** records of several different relations can be stored in the same file and in fact in the same block within a file
  - Motivation: store related records on the same block to reduce the cost of certain join operations (to minimize I/O)





# Organization of Records in Files

- **B<sup>+</sup>-tree file organization**
  - The traditional sequential file organization does support ordered access even if there are insert, delete, and update operations, which may change the ordering of records.
  - However, in the face of a large number of such operations, efficiency of ordered access suffers
  - The B<sup>+</sup>-tree file organization is related to the B<sup>+</sup>-tree index structure and can provide efficient ordered access to records even if there are a large number of insert, delete, or update operations.
  - Further, it supports very efficient access to specific records, based on the search key.
- **Hashing** – a hash function computed on some attribute of each record; 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

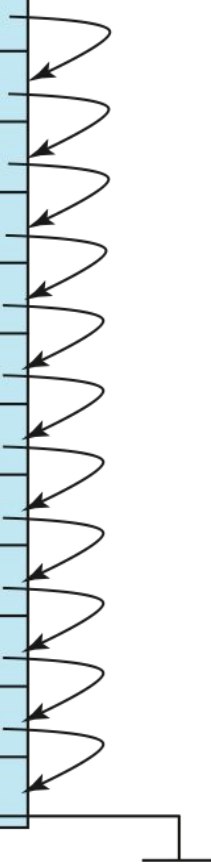
- A **sequential file** is designed for efficient processing of records in sorted order based on some search key.
- A **search key** is any attribute or set of attributes; it need not be the primary key, or even a superkey.
- To permit fast retrieval of records in search-key order, we chain together records by pointers. The pointer in each record points to the next record in search-key order.
- Furthermore, to minimize the number of block accesses in sequential file processing, we store records physically in search-key order, or as close to search-key order as possible.
- The sequential file organization allows records to be read in sorted order; that can be useful for display purposes, as well as for certain query-processing algorithms.



# Sequential File Organization

- Example: *Instructor* relation
- The records in the file are ordered by a **search-key**, *ID*

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	





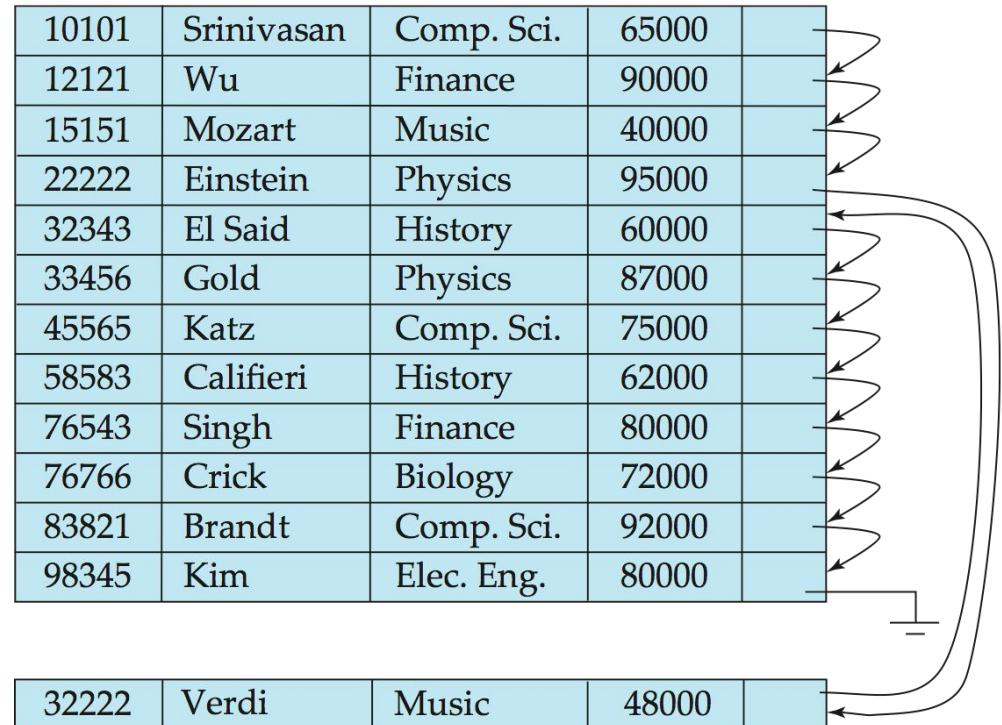
# Sequential File Organization

- It is difficult, however, to maintain physical sequential order as records are inserted and deleted, since it is costly to move many records as a result of a single insertion or deletion.
- We can manage **deletion** by using pointer chains (free list) , as used previously.
- For **insertion**, we apply the following rules:
  - 1. Locate the record in the file that comes before the record to be inserted in search-key order.
  - 2. If there is a free record (that is, space left after a deletion) within the same block as this record, insert the new record there. Otherwise, insert the new record in an *overflow* block. In either case, adjust the pointers so as to chain together the records in search-key order.



# 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





# Sequential File Organization

- **Reorganization**
- If relatively few records need to be stored in overflow blocks, this approach works well.
- Eventually, however, the correspondence between search-key order and physical order may be totally lost over a period of time, in which case sequential processing will become much less efficient.
- At this point, the file should be **reorganized** so that it is once again physically in sequential order. Such reorganizations are costly, and must be done during times when the system load is low.
- The frequency with which reorganizations are needed depends on the frequency of insertion of new records. In the extreme case in which insertions rarely occur, it is possible always to keep the file in physically sorted order.
- The B+-tree file organization, which we describe later, provides efficient ordered access even if there are many inserts, deletes, and updates, without requiring expensive reorganizations.



# Multitable Clustering File Organization

- Most relational database systems store each relation in a separate file, or a separate set of files. Thus, each file, and as a result, each block, stores records of only one relation, in such a design.
- Usually, tuples of a relation can be represented as fixed-length records. Thus, relations can be mapped to a simple file structure. This simple implementation of a relational database system is well suited to low-cost database implementations as in, for example, embedded systems or portable devices.
- This simple approach to relational database implementation becomes less satisfactory as the size of the database increases.
- There are performance advantages to be gained from careful assignment of records to blocks, and from careful organization of the blocks themselves. Clearly, a more complicated file structure may be beneficial, even if we retain the strategy of storing each relation in a separate file.





# Multitable Clustering File Organization

- However, many large-scale database systems do not rely directly on the underlying operating system for file management.
- Instead, one large operating system file is allocated to the database system. The database system stores all relations in this one file, and manages the file itself.
- Even if multiple relations are stored in a single file, by default most databases store records of only one relation in a given block. This simplifies data management.
- However, in some cases it can be useful to store records of more than one relation in a single block. To see the advantage of storing records of multiple relations in one block, consider the following SQL query for the university database:

```
select dept_name, building, budget, ID, name, salary  
from department natural join instructor;
```



# 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

- This structure mixes together tuples of two relations, but allows for efficient processing of the join.
- When a tuple of the *department* relation is read, the entire block containing that tuple is copied from disk into main memory. Since the corresponding *instructor* tuples are stored on the disk near the *department* tuple, the block containing the *department* tuple contains tuples of the *instructor* relation needed to process the query. If a department has so many instructors that the *instructor* records do not fit in one block, the remaining records appear on nearby blocks.
- A **multitable clustering file organization** is a file organization, that stores related records of two or more relations in each block. Such a file organization allows us to read records that would satisfy the join condition by using one block read. Thus, we are able to process this particular query more efficiently.



# Multitable Clustering File Organization (cont.)

- Good and Bad side
- Good for queries involving *department* ~~ins~~ *instructor*, and for queries involving one single department and its instructors
- Bad for queries involving only *department*
- *select \* from deparment* requires more block accesses than it did in the scheme under which we stored each relation in a separate file, since each block now contains significantly fewer *department* records.
- Results in variable size records
- When multitable clustering is to be used depends on the types of queries that the database designer believes to be most frequent. Careful use of multitable clustering can produce significant performance gains in query processing



# Multitable Clustering File Organization (cont.)

- Add pointer chains to link records of a particular relation
- To locate efficiently all tuples of the *department* relation within a particular block, we can chain together all the records of that relation using pointers; however, the number of blocks read does not get affected by using such chains.

Comp. Sci.	Taylor	100000	
45564	Katz	75000	
10101	Srinivasan	65000	
83821	Brandt	92000	
Physics	Watson	70000	
33456	Gold	87000	



# 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
    - 4 E.g., *transaction* partition for current year on SSD, for older years on magnetic disk



# Data Dictionary Storage

- A relational database system needs to maintain data *about* the relations, such as the schema of the relations. In general, such “data about data” is referred to as **metadata**.
- Relational schemas and other metadata about relations are stored in a structure called the **data dictionary** or **system catalog**.
- Among the types of information that the system must store are these:
- Information about relations
  - names of relations
  - names, types (domain) and lengths of attributes of each relation
  - names and definitions of views
  - integrity constraints
- In addition, many systems keep the following data on users of the system:
  - Names of users, the default schemas of the users, and passwords or other information to authenticate users
  - Information about authorizations for each user



# Data Dictionary Storage

- Further, the database may store statistical and descriptive data about the relations, such as:
  - Number of tuples in each relation.
  - the number of distinct values for each attribute.
- The data dictionary may also note the storage organization (sequential, hash, or heap) of relations, and the location where each relation is stored:
  - If relations are stored in operating system files, the dictionary would note the names of the file (or files) containing each relation.
  - If the database stores all relations in a single file, the dictionary may note the blocks containing records of each relation in a data structure such as a linked list.
- Also store information about each index on each of the relations:
  - Name of the index
  - Name of the relation being indexed
  - Attributes on which the index is defined
  - Type of index formed





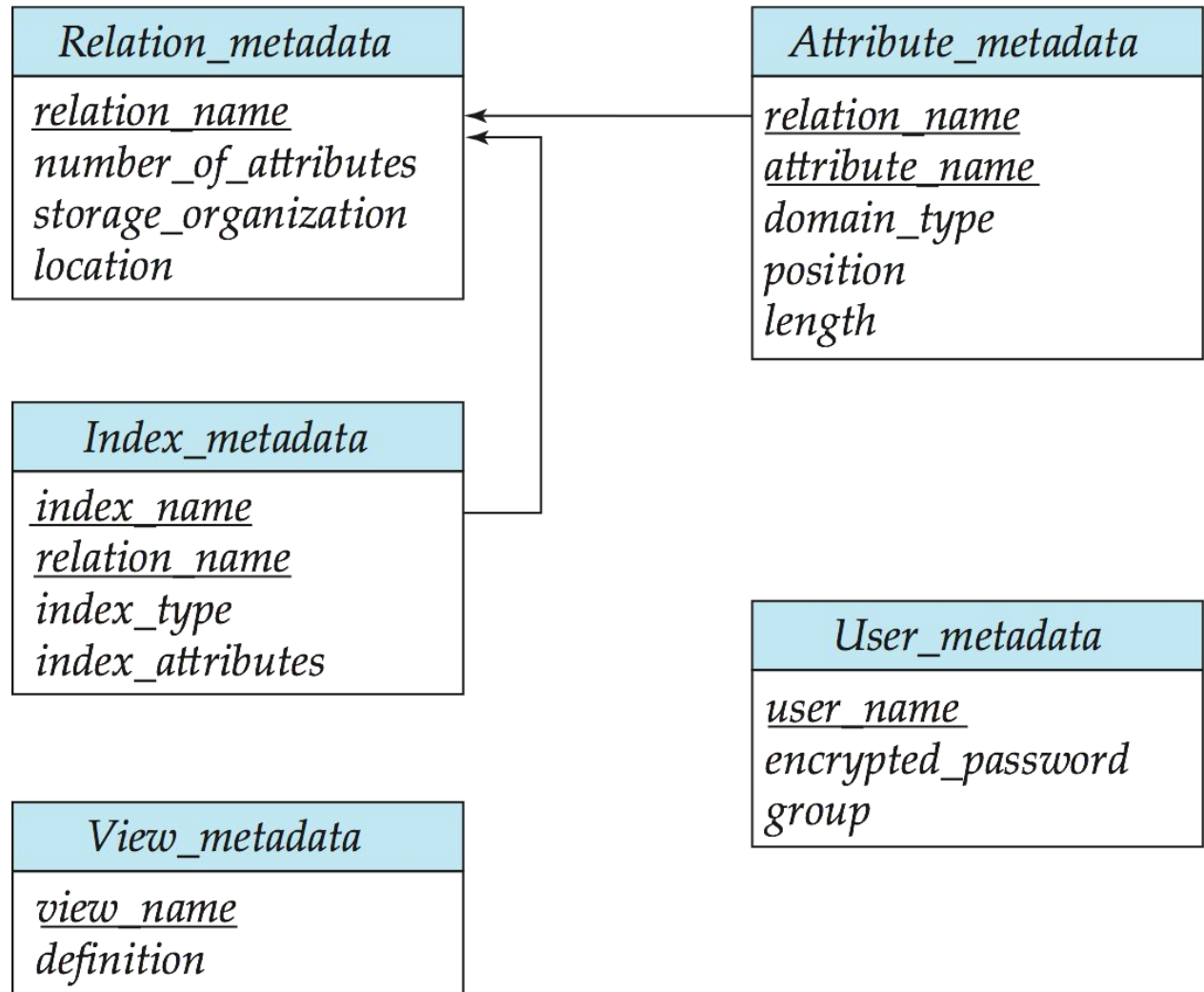
# Data Dictionary Storage

- All this metadata information constitutes, in effect, a miniature database.
- Some database systems store such metadata by using special-purpose data structures and code.
- It is generally preferable to store the data about the database as relations in the database itself. By using database relations to store system metadata, we simplify the overall structure of the system and harness the full power
- Whenever the database system needs to retrieve records from a relation, it must first consult the *Relation\_metadata* relation to find the location and storage organization of the relation, and then fetch records using this information.
- However, the storage organization and location of the *Relation\_metadata* relation itself must be recorded elsewhere (for example, in the database code itself, or in a fixed location in the database), since we need this information to find the contents of *Relation\_metadata*.



# Relational Representation of System Metadata

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





# End of Chapter 10

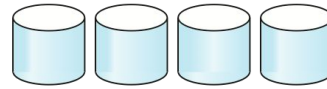
**Database System Concepts, 6<sup>th</sup> Ed.**

©Silberschatz, Korth and Sudarshan

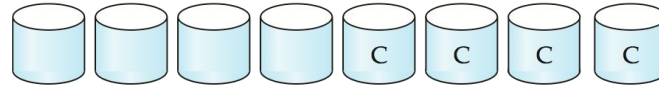
See [www.db-book.com](http://www.db-book.com) for conditions on re-use



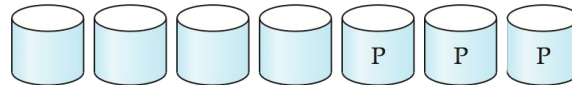
# Figure 10.03



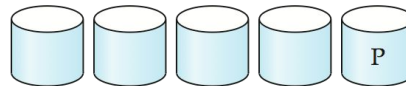
(a) RAID 0: nonredundant striping



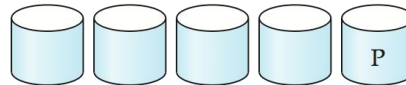
(b) RAID 1: mirrored disks



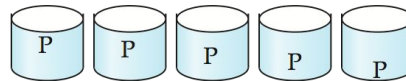
(c) RAID 2: memory-style error-correcting codes



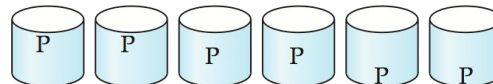
(d) RAID 3: bit-interleaved parity



(e) RAID 4: block-interleaved parity



(f) RAID 5: block-interleaved distributed parity



(g) RAID 6: P + Q redundancy



# Figure 10.18

Disk 1	Disk 2	Disk 3	Disk 4
$B_1$	$B_2$	$B_3$	$B_4$
$P_1$	$B_5$	$B_6$	$B_7$
$B_8$	$P_2$	$B_9$	$B_{10}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$



## Figure in-10.1

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