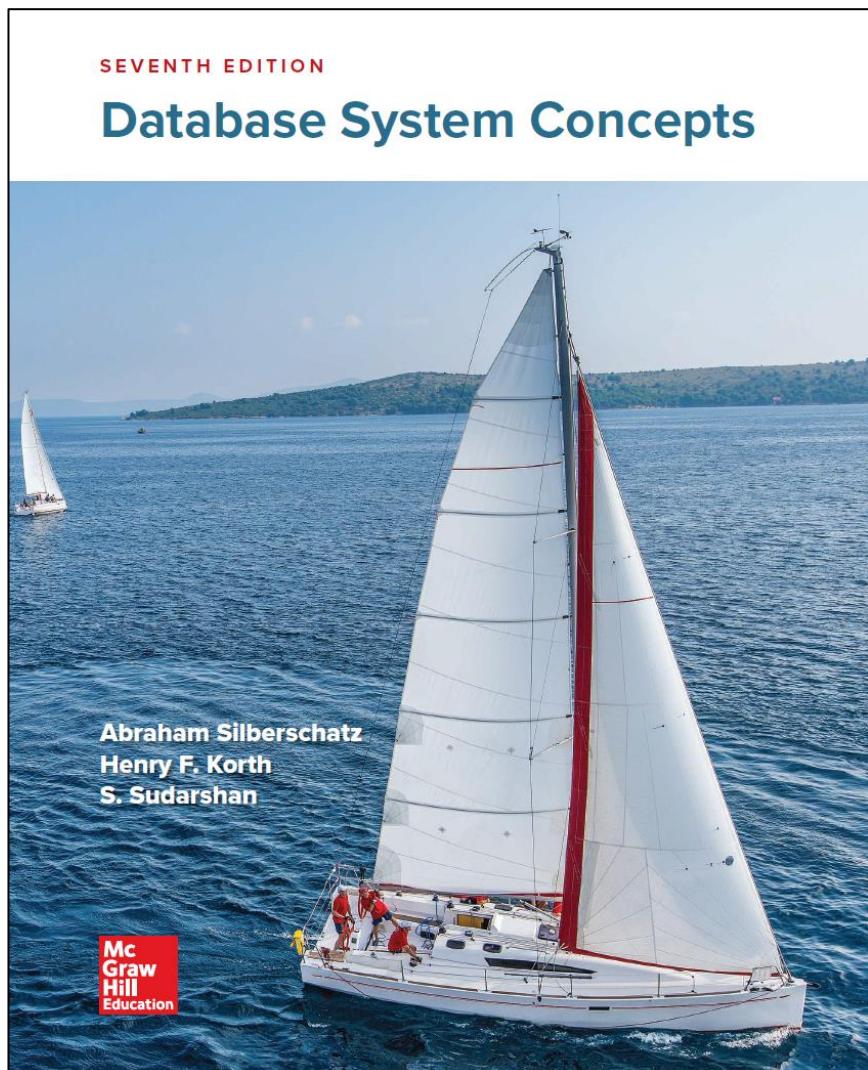


# Data Administration in Information Systems

---

Storage and file organization

# Storage and file organization



### x Contents

#### PART FIVE ■ STORAGE MANAGEMENT AND INDEXING

##### Chapter 12 Physical Storage Systems

12.1 Overview of Physical Storage Media	559	12.6 Disk-Block Access	577
12.2 Storage Interfaces	562	12.7 Summary	580
12.3 Magnetic Disks	563	Exercises	582
12.4 Flash Memory	567	Further Reading	584
12.5 RAID	570		

##### Chapter 13 Data Storage Structures

13.1 Database Storage Architecture	587	13.7 Storage Organization in Main-Memory Databases	615
13.2 File Organization	588	13.8 Summary	617
13.3 Organization of Records in Files	595	Exercises	619
13.4 Data-Dictionary Storage	602	Further Reading	621
13.5 Database Buffer	604		
13.6 Column-Oriented Storage	611		

##### Chapter 14 Indexing

14.1 Basic Concepts	623	14.8 Write-Optimized Index Structures	665
14.2 Ordered Indices	625	14.9 Bitmap Indices	670
14.3 B <sup>+</sup> -Tree Index Files	634	14.10 Indexing of Spatial and Temporal Data	672
14.4 B <sup>+</sup> -Tree Extensions	650	14.11 Summary	677
14.5 Hash Indices	658	Exercises	679
14.6 Multiple-Key Access	661	Further Reading	683
14.7 Creation of Indices	664		

#### PART SIX ■ QUERY PROCESSING AND OPTIMIZATION

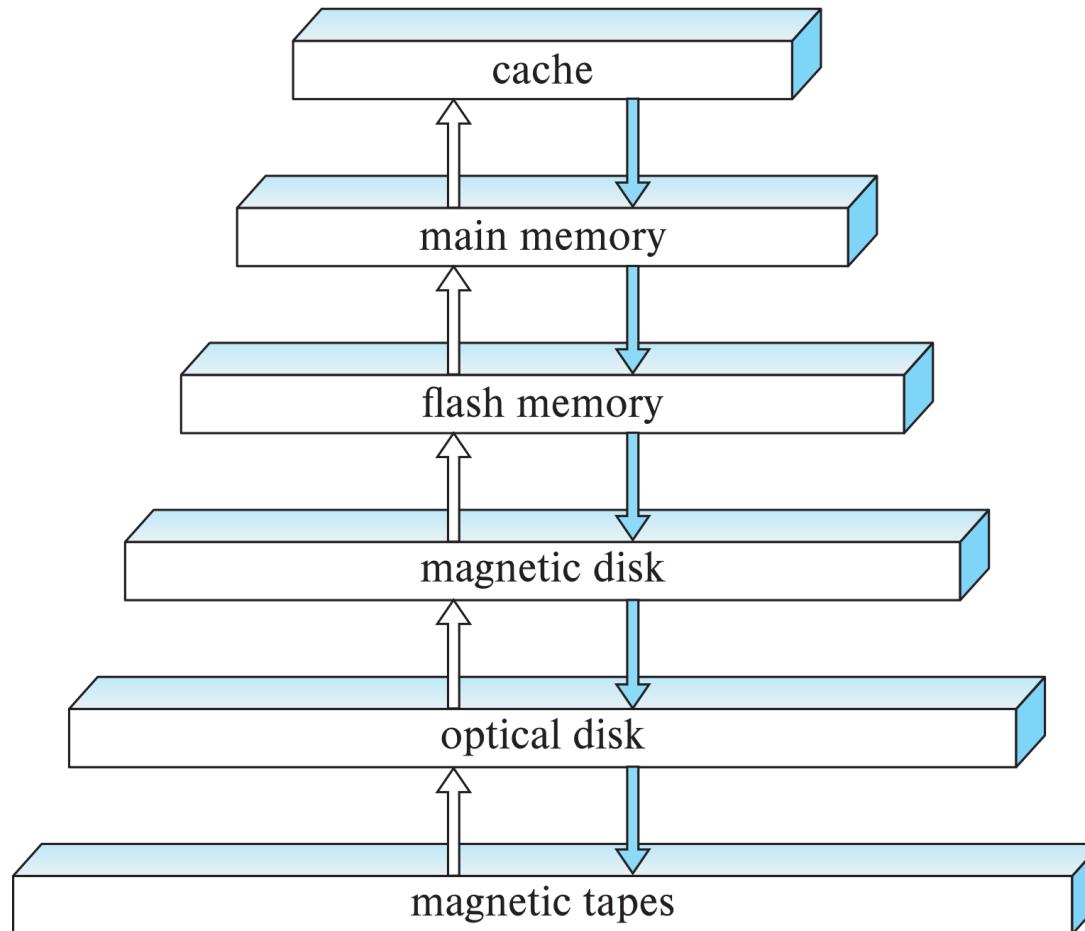
##### Chapter 15 Query Processing

15.1 Overview	689	15.7 Evaluation of Expressions	724
15.2 Measures of Query Cost	692	15.8 Query Processing in Memory	731
15.3 Selection Operation	695	15.9 Summary	734
15.4 Sorting	701	Exercises	736
15.5 Join Operation	704	Further Reading	740
15.6 Other Operations	719		

# 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



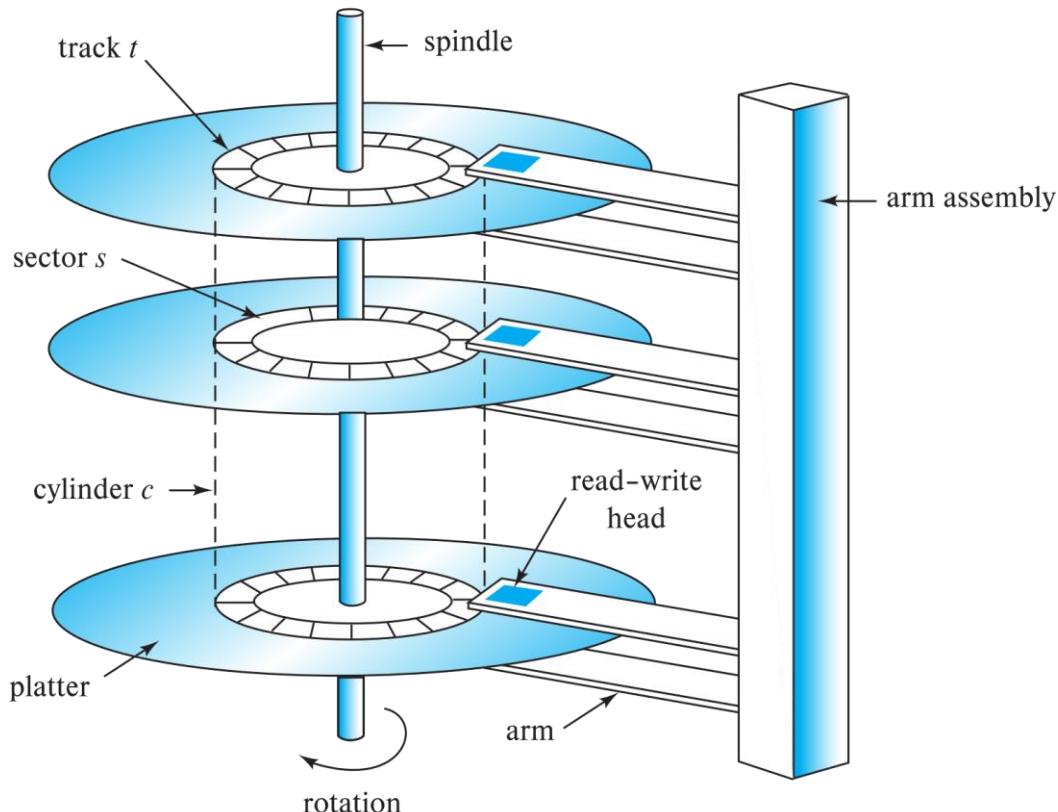
# Storage Hierarchy (Cont.)

- **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

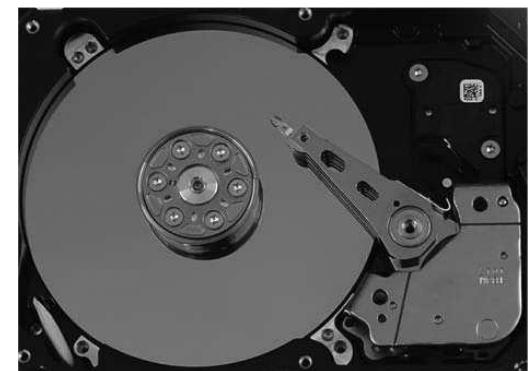
# Storage Interfaces

- 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**

# 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  $i^{\text{th}}$  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
    - 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.)

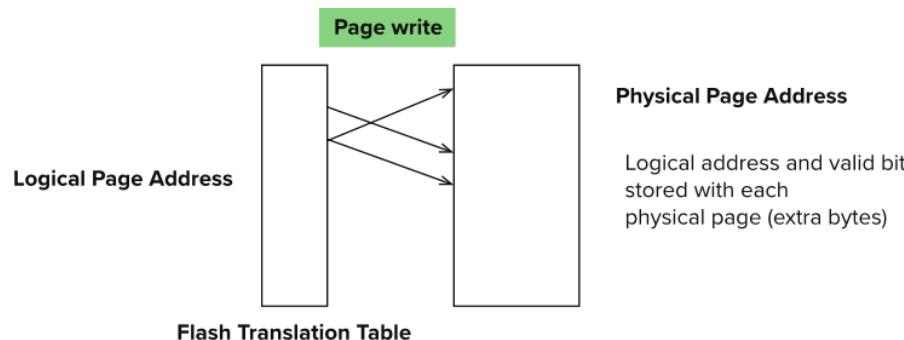
- **Mean time to failure (MTTF)** – the average time the disk is expected to run continuously without any failure.
  - Typically 3 to 5 years
  - Probability of failure of new disks is quite low, corresponding to a “theoretical MTTF” of 500,000 to 1,200,000 hours for a new disk
    - E.g., an MTTF of 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

- **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

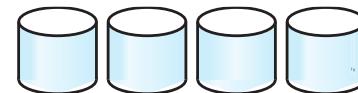
- **Redundancy** – store extra information that can be used to rebuild information lost in a disk failure
- E.g., **Mirroring (or shadowing)**
  - Duplicate every disk. Logical disk consists of two physical disks.
  - Every write is carried out on both disks
    - Reads can take place from either disk
  - 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  $500 \times 10^6$  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



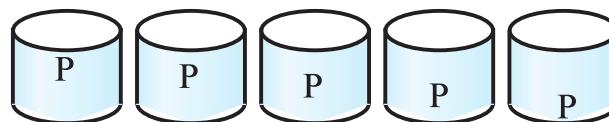
(b) RAID 1: mirrored disks

# 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.

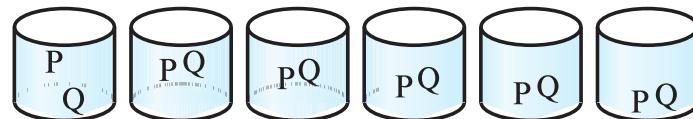


(c) RAID 5: block-interleaved distributed parity

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

---

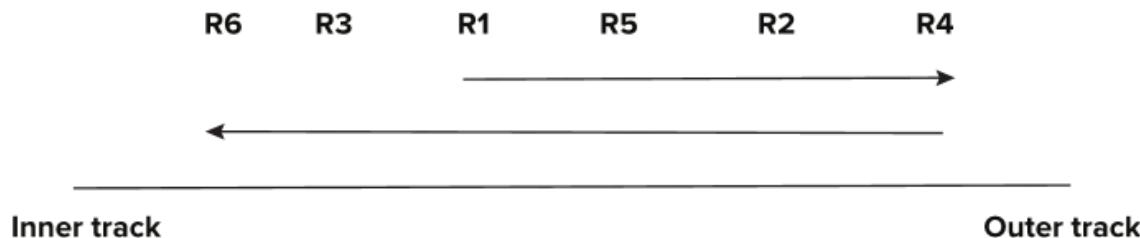
- **Software RAID:** RAID implementations done entirely in software, with no special hardware support
- **Hardware RAID:** RAID implementations with special hardware
  - Use non-volatile RAM to record writes that are being executed
  - Beware: power failure during write can result in corrupted disk
    - 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 (Cont.)

- **File organization**
  - Allocate blocks of a file in as contiguous a manner as possible
  - Allocation in units of **extents**
  - Files may get **fragmented**
    - E.g., if free blocks on disk are scattered, and newly created file has its blocks scattered over the disk
    - Sequential access to a fragmented file results in increased disk arm movement
    - Some systems have utilities to **defragment** the file system, in order to speed up file access

# 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.

# 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*

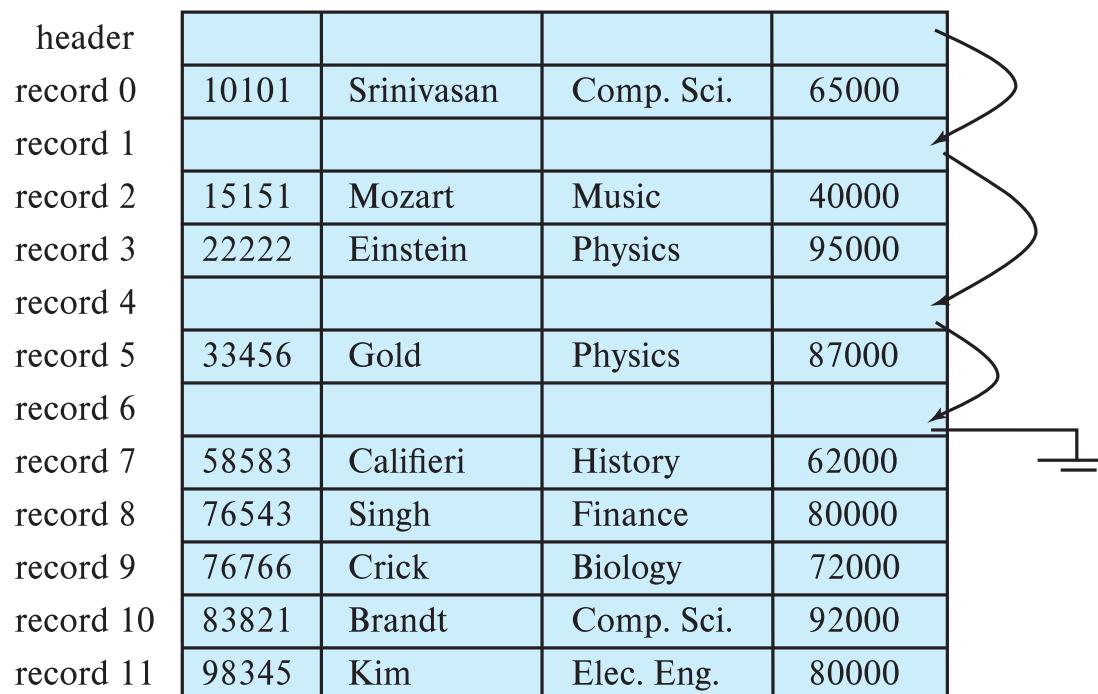
**Record 3 deleted and replaced by record 11**

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***

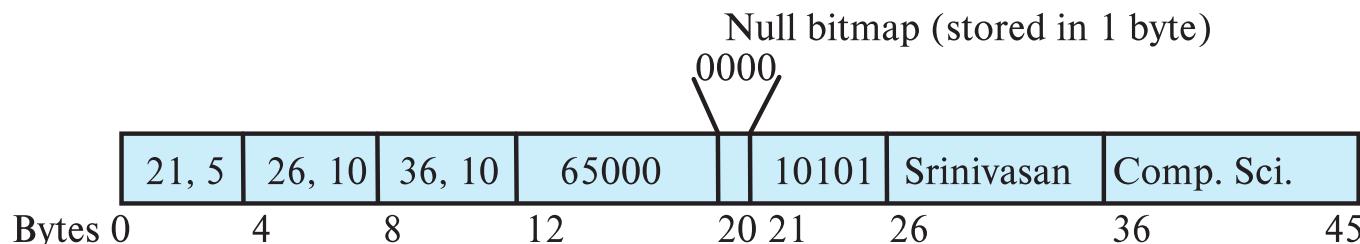
header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology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



The diagram illustrates the deletion of record 5 from a fixed-length record file. The file is represented as a table with 12 records. Record 5, which contains the data for 'Gold' with a physics grade of 87000, is being deleted. Arrows point from both the header cell and the data cells of record 5 to a pointer at the end of the file, indicating that the freed space is linked into a free 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



# 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

# 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<sup>+</sup>-tree file organization**
  - Ordered storage even with inserts/deletes
  - (More on this in the next classes)
- **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 the next classes)

# 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

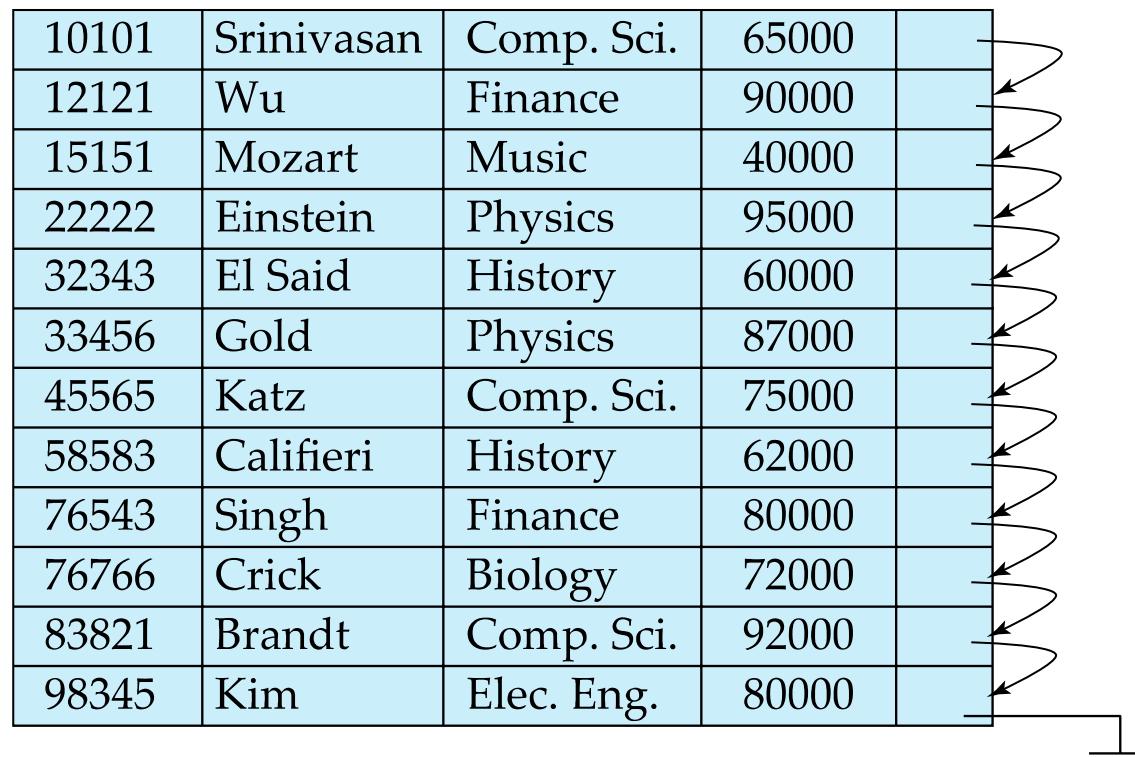
4	7	2	6
---	---	---	---

- 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**

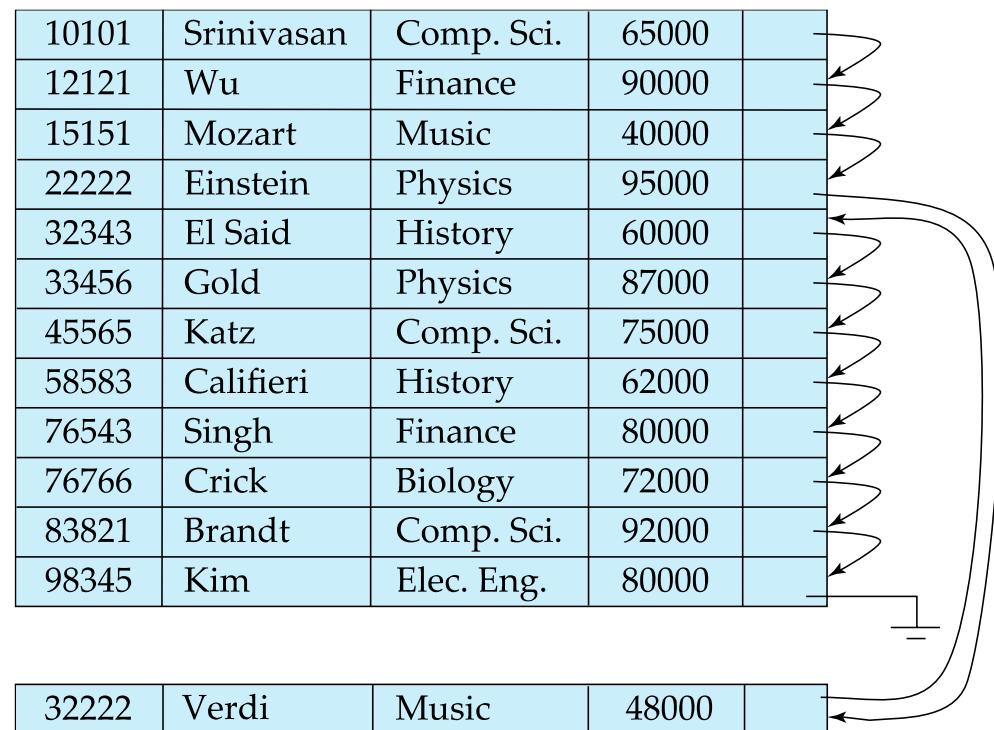
10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp.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



The diagram illustrates a sequential file organization. It consists of a table with 12 records, each representing a student's information: student ID, name, major, and salary. The records are arranged in ascending order of student ID. To the right of the table, a series of curved arrows points from the right edge of each record towards a single vertical line, representing the sequential access path through 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

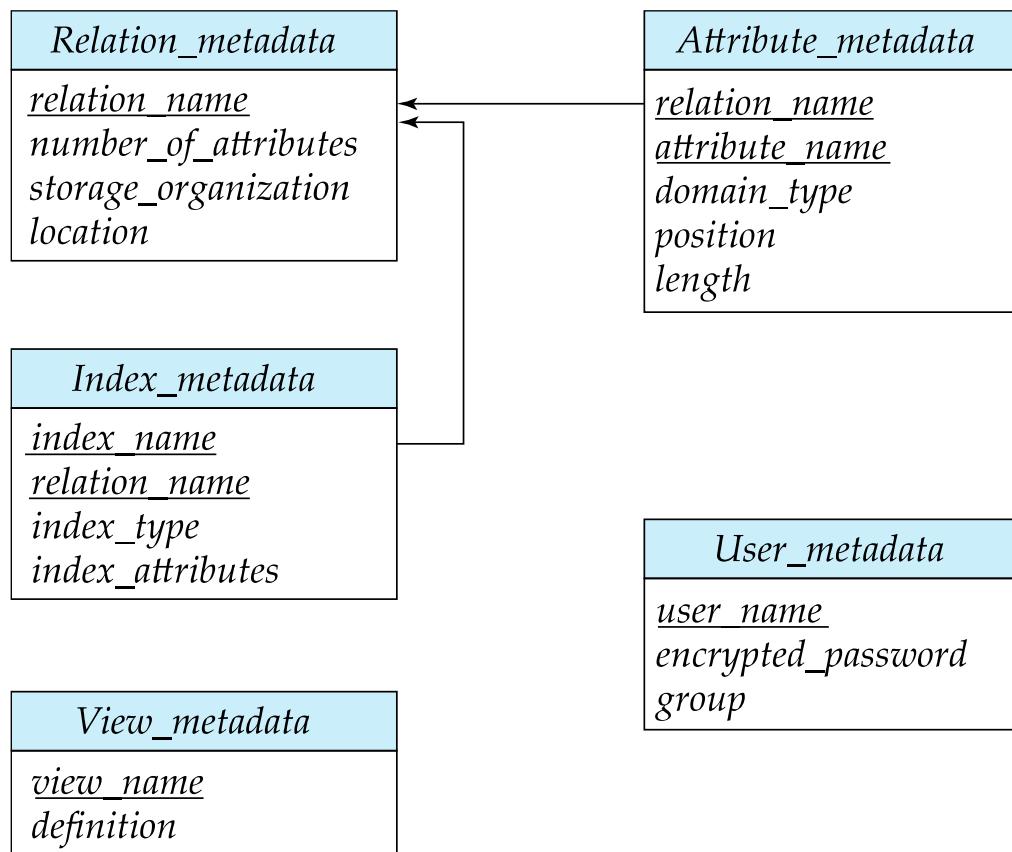
# Data Dictionary Storage

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 (next classes)

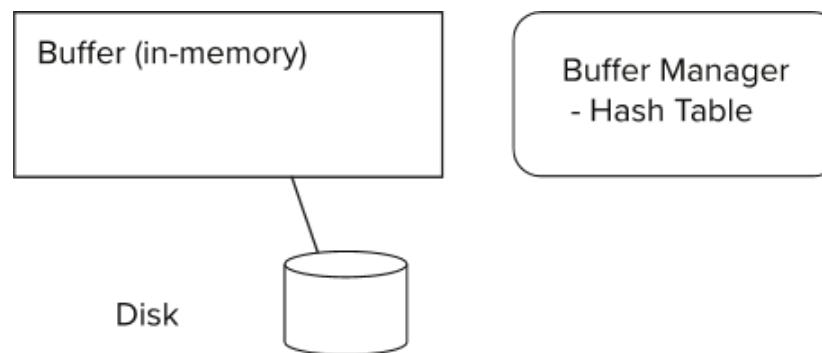
# Relational Representation of System Metadata

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



# 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.



# 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

- **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  $t_r$  of  $r$  do
  - for each tuple  $t_s$  of  $s$  do
  - if the tuples  $t_r$  and  $t_s$  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

# Optimization of Disk Block Access (Cont.)

- Buffer managers support **forced output** of blocks for the purpose of recovery (more in the next classes)
- **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

# 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**