UST 과학기술연합대학원대학교
UNIVERSITY OF SCIENCE & TECHNOLOGY

Lecture 7: Storage Systems & Storage Structure

Dr. Kyong-Ha Lee
(kyongha@kisti.re.kr)

KiSTi www.kisti.re.kr 한국과학기술정보연구원
Korea Institute of Science and Technology Information

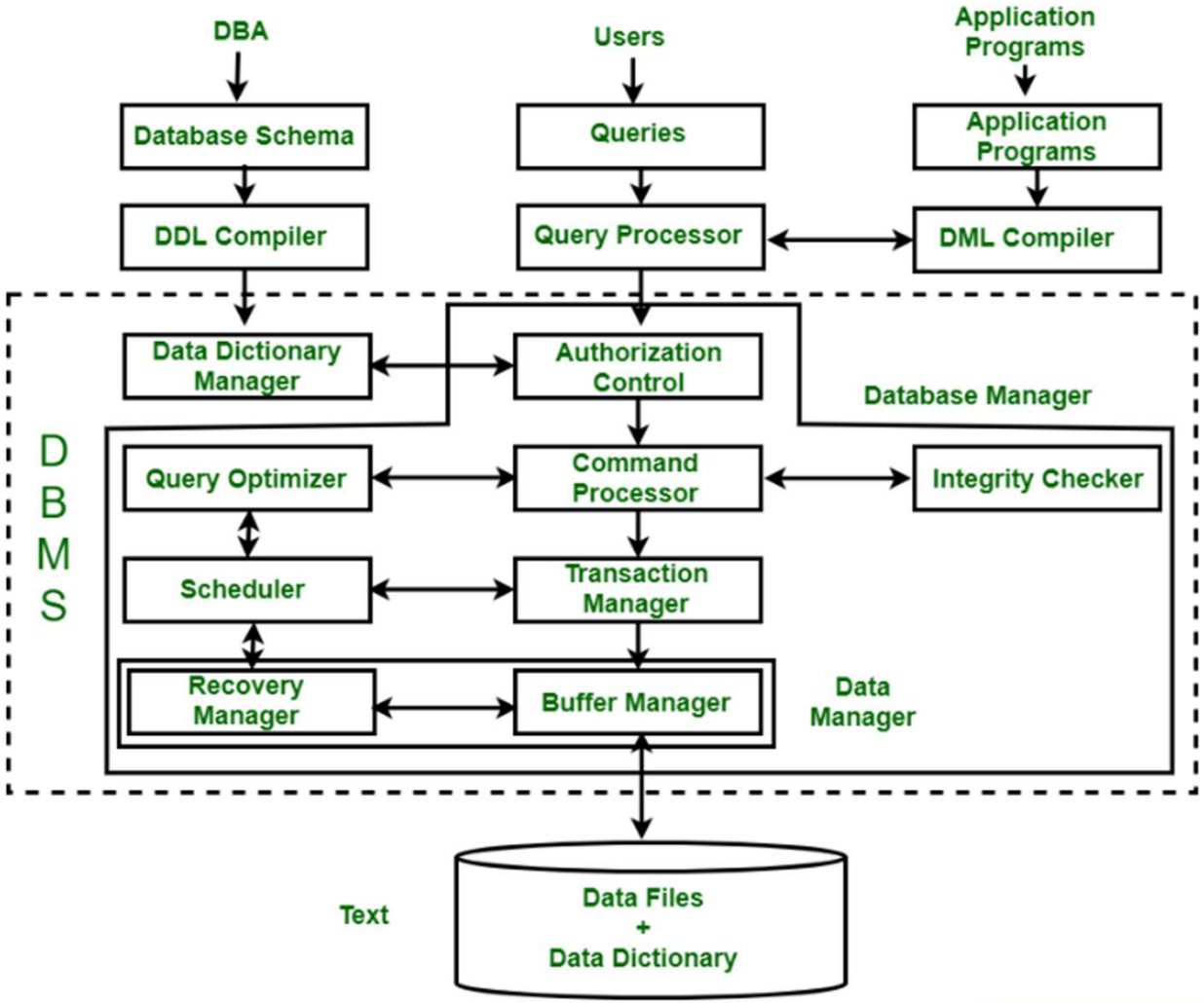
Contents

Brief overview of this lecture

- Basic theories and principles about storage systems and data storage structures used in a database
- Not much discussion on implementation or tools, but will be happy to discuss them if there are any questions

*Disclaimer: these slides are based on the slides created by the authors of Database System Concepts 7th ed. and modified by K.H. Lee.

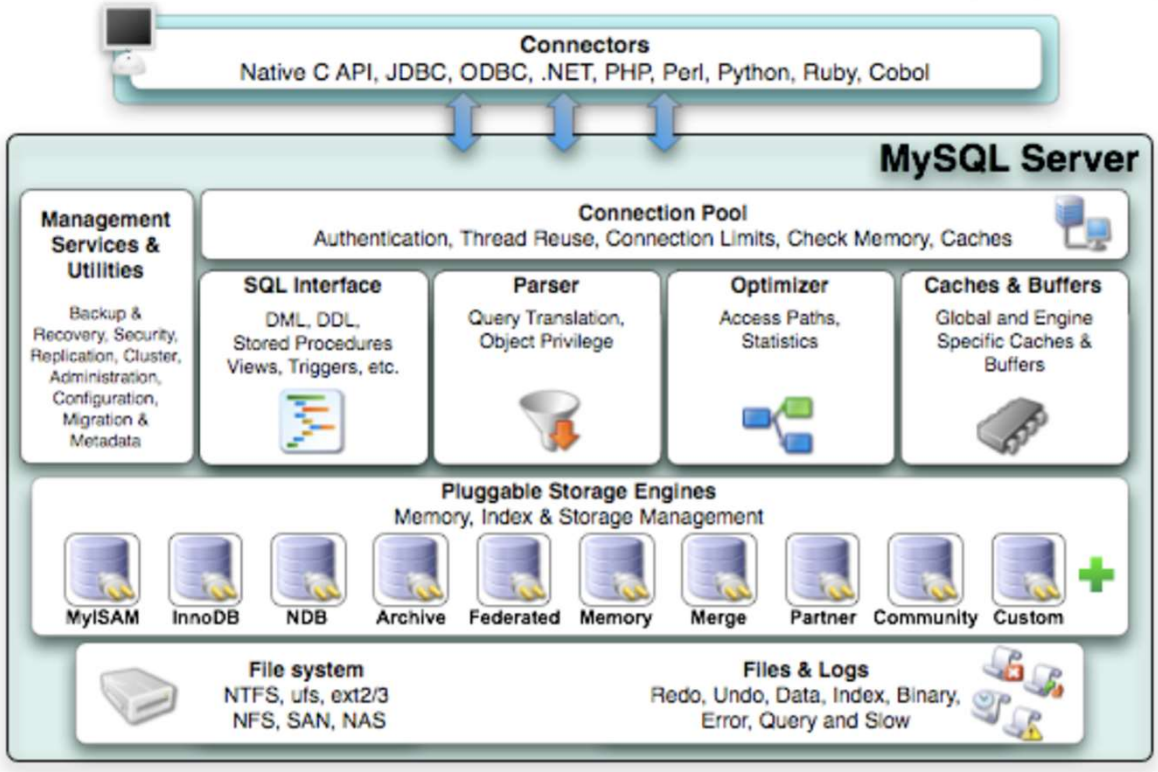
DBMS Architecture Overview



* Source: Structure of Database Management System - GeeksforGeeks

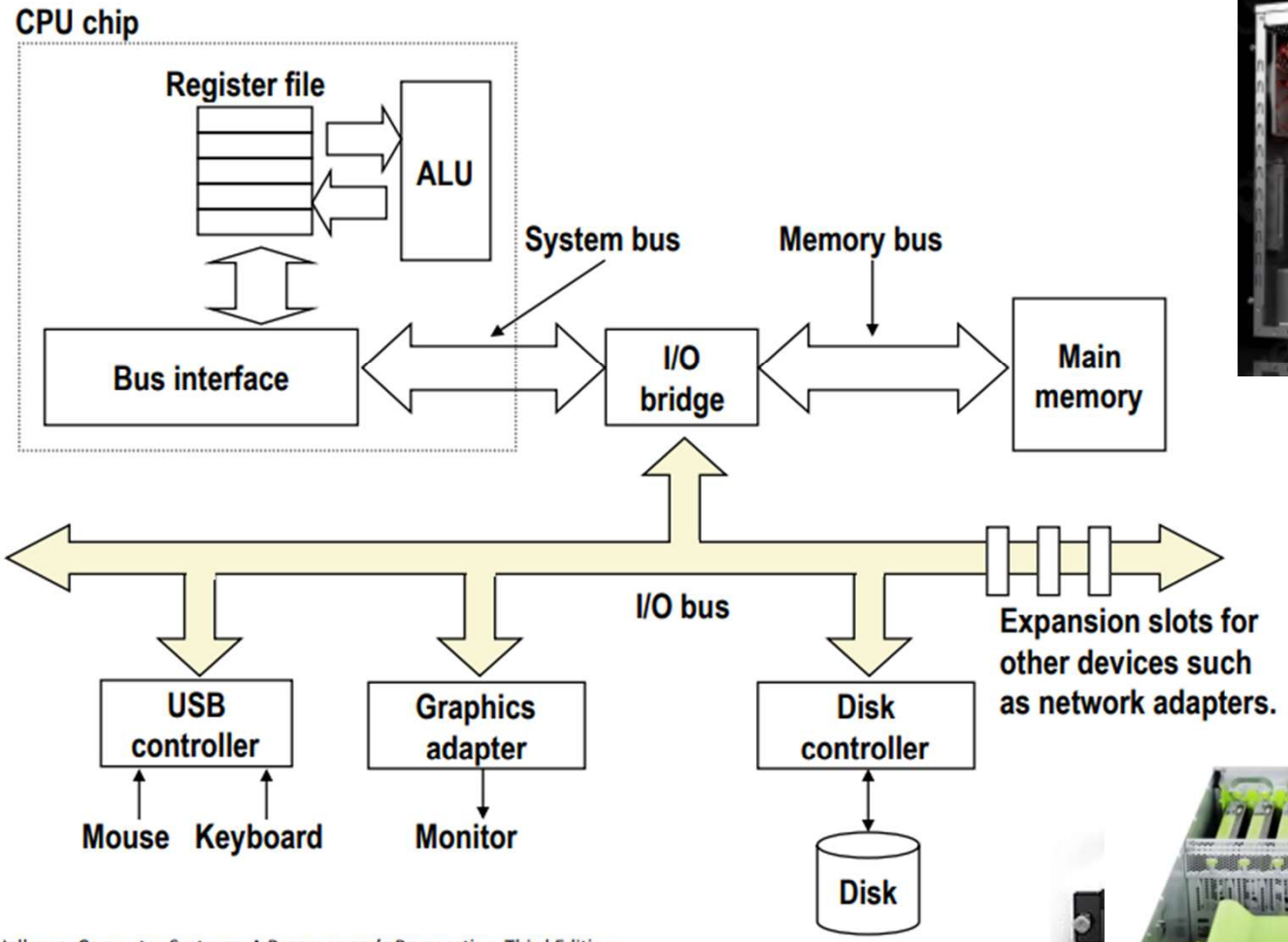
MariaDB Architecture

- Ideally the same as MySQL since it is a community-developed fork of the MYSQL
  - With the concerns of MySQL acquisition by Oracle Inc. in 2009



* Source:https://docs.oracle.com/cd/E19957-01/mysql-refman-5.5/storage-engines.html

I/O BUS in Computer Systems



Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Quanta Cloud Technology Rackgo X Big Sur

Disk and Files

- DBMS stores information on disks
- This has major implications for DBMS design
  - **READ**: transfer data from disk to main memory(RAM)
  - **WRITE**: transfer data from RAM to disk
  - Both are high-cost operations, relative to in-memory operations, so must be planned carefully!
- Why not store everything in main memory?
  - Costs are still too much
    - As of 2022, 1 GB for **4,479 won** ($22.5) with RAM, 18won(<¢1.5) with HDD
  - Main memory is volatile!

Disks

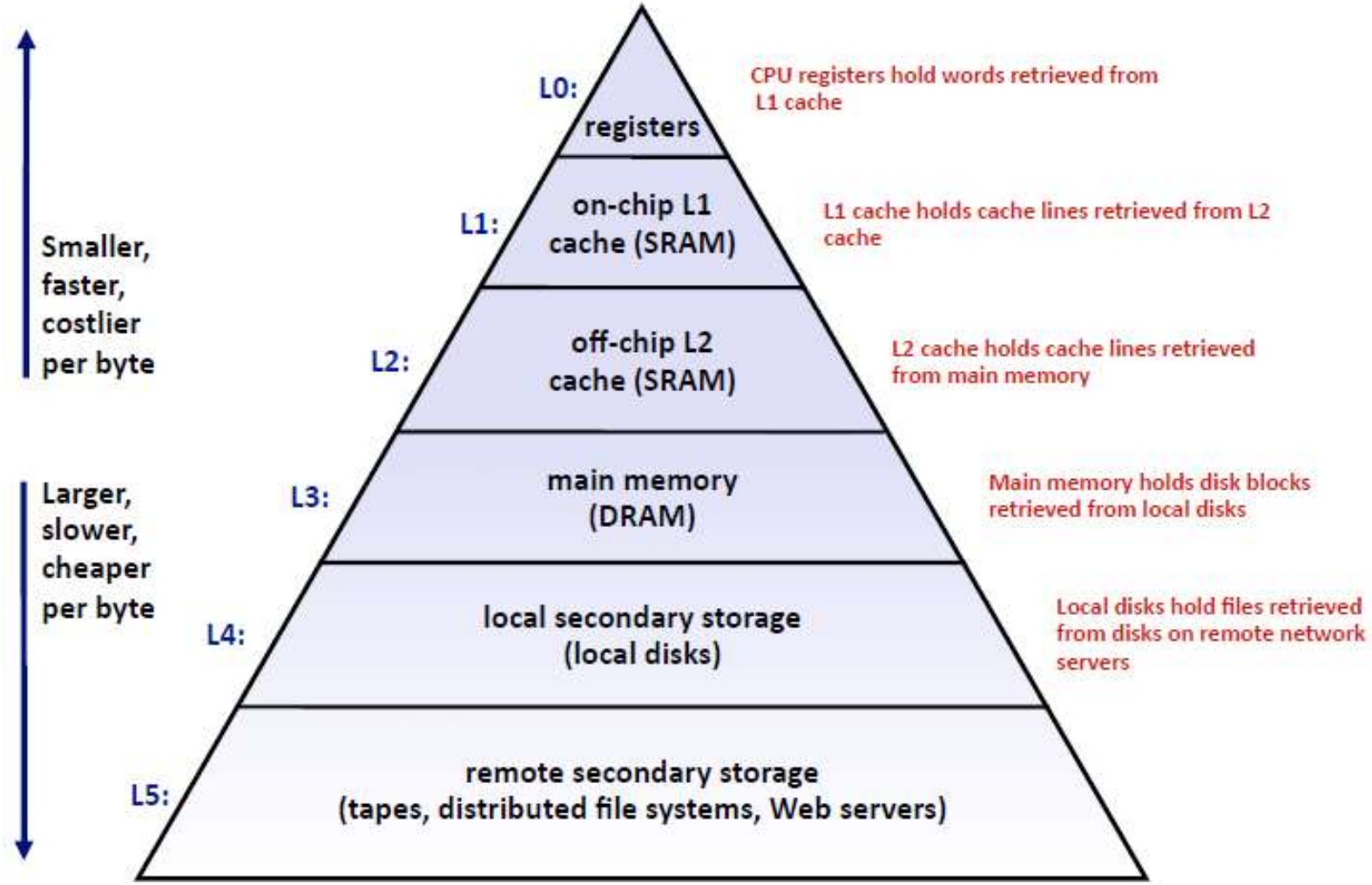
- Secondary storage device of choice
- Main advantage over tapes
  - **random access vs. sequential access**
- Data is stored and retrieved in units called **disk blocks** or **pages**
- Unlike RAM, time to retrieve a disk page varies depending upon location of disk.
  - Therefore, relative placement of pages on disk has major impact on DBMS performance

Classification of Physical Storages

- **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 batter-backed up main-memory.

- Factors affecting choice of storage media include
  - Speed with which data can be accessed
    - Bandwidth vs. latency
  - Cost per unit of data
  - Reliability

Memory Hierarchy



Memory hierarchy

- **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(SSD), magnetic disks(HDD)
- **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(ODD)
  - 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

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
- **Storage Area Networks (SAN)**
  - a large number of disks are connected by a high-speed network to a number of servers
- **Network Attached Storage (NAS)**
  - networked storage provides a file system interface using networked file system protocol, instead of providing a disk system interface
  - NAS vs SAN - Network Attached Storage vs Storage Area Network - YouTube

Now in Memory Hierarchy

- Very cheap HDD with very very high capacity
  - Seagate 8TB Barracuda 3.5' HDD (7200rpm, 256MB) for 142,800 won($115.52) in April 2022
    - 1GB for **18 won**
- Write-once storage
  - Tape drive is dead, ODD is waning
  - Due to the poor latency and seek time
    - Seek time >= 100ms
  - although 22X DVD writer can sequentially write 4.7GBytes within 3 minutes (29.7MB in theory)
    - 1GB for **53.83 won**
- Price of RAM has fallen enough to keep much more data in memory than before.
  - A 8GB DDR4 Memory(3,200MHz) for 35,830 won
    - 1 GB for **4,479 won** ($22.5)
  - but, still the cost is much higher than the cost of HDD

The Five-Minute Rule

- Cache randomly accessed disk pages that are reused every 5 minutes

Break-even Reference Interval(seconds) =

$$\frac{PagesPerMBofRAM}{AccessPerSecondPerDisk}$$ *Technology ratio*

x

$$\frac{PricePerDiskDrive}{PricePerMBofDRAM}$$ *Economic ratio*

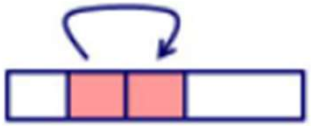
- In 1987, breakeven interval was **~2** minutes
- After that, **~5** minutes in 1997, **~88** minutes in 2007, **~5 hours** in 2017
- **"Memory becomes HDD, HDD becomes Tape, and Tape is dead", by Jim Gray**
- Today's memory is ~102,400 times faster than HDD (2010)
  - Memory : 83 ns(250 cycles)
  - HDD : 8.5ms (25.7 million cycles)
    - (256/116) x (61.94/0.0225) = ~**101 minutes**.

=> **Cache your data in memory as always as possible.**

Locality

- **Locality** of reference
  - Tendency of a processor to access to the same set of memory locations repetitively over a short period of time
  - Predictable behivior that occurs in computer systems and it is a great candidate for performance optimization

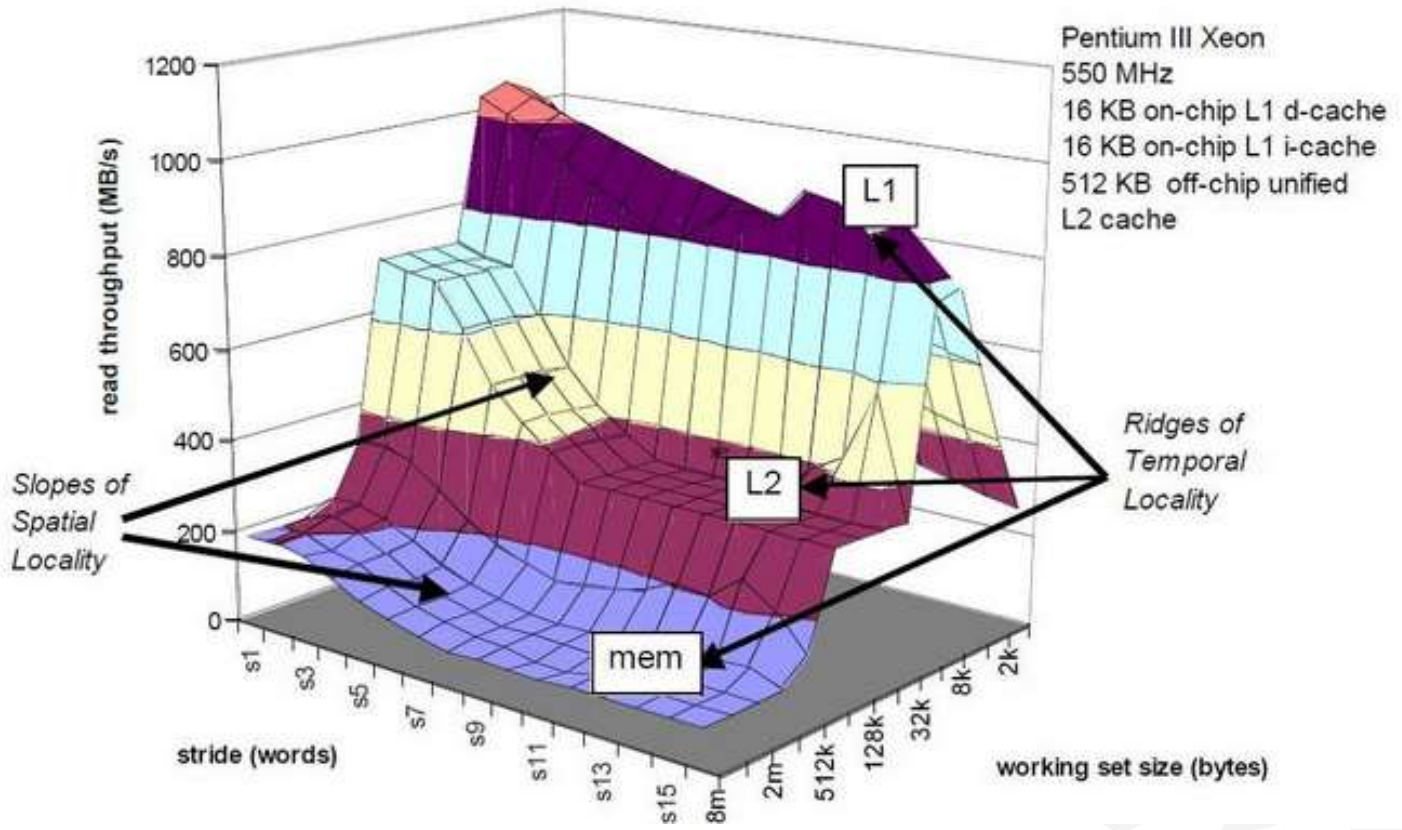
- **Temporal locality**
  - Recently referenced items are likely to be referenced in the near future
  - Capacity limits the # of items to be kept at a time
- **Spatial locality**



  - Also kwnoaas data locality
  - Items with nearby addressed tend to be referencced closed together in time

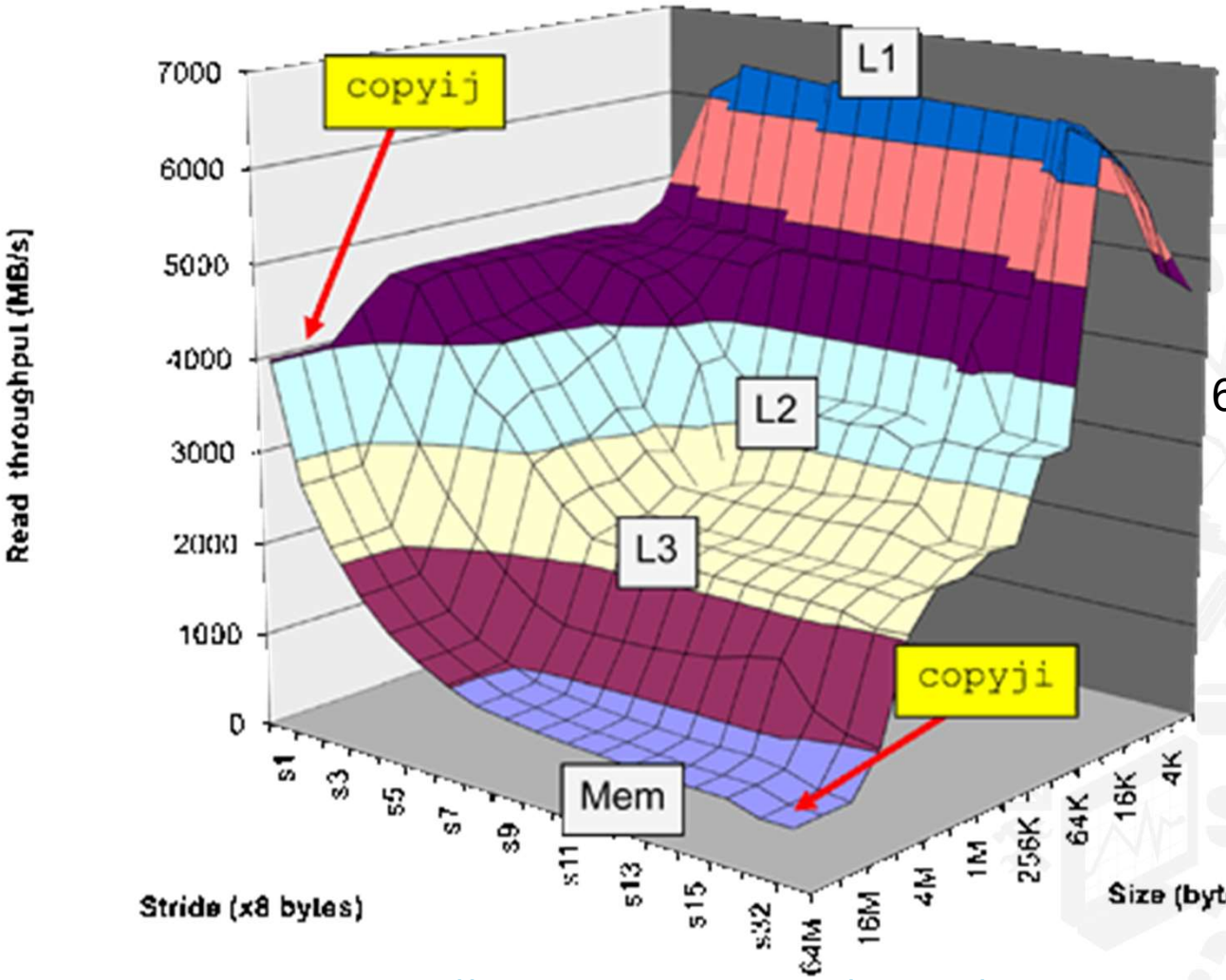
Memory Mountain(2000)

- Spatial and temporal localities affect system performance



*Source: Computer Systems, A Programmer's Perspective, 2003

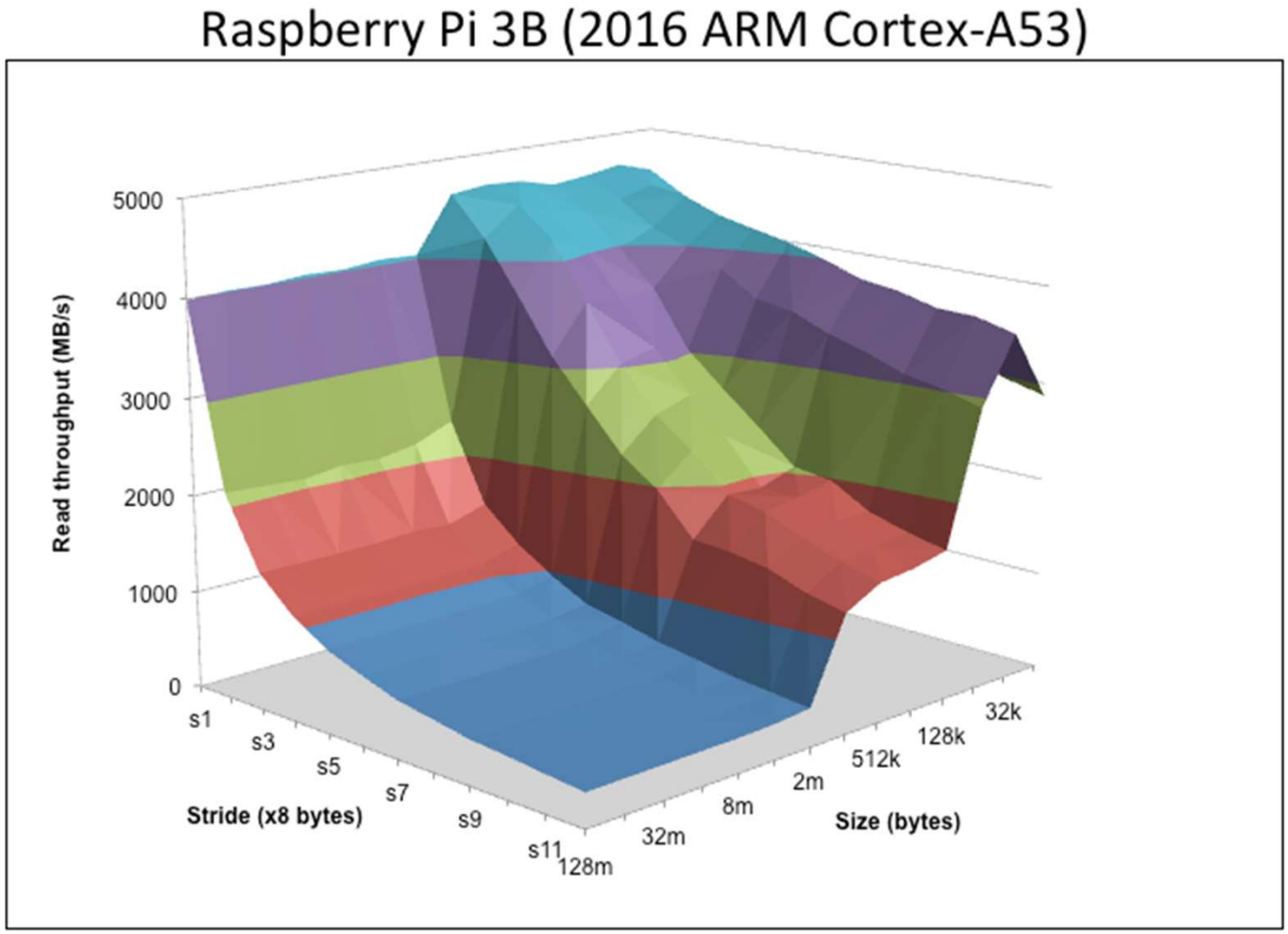
Memory Mountain(2010)



Intel Core i7
2.67GHz
32KB L1 d-cache
256KB L2 cache
8MB L3 cache
64B cache line size

source: http://csapp.cs.cmu.edu/public/perspective.html

Memory Mountain(2016, Mobile)



*source : [CS:APP: A Gallery of Memory Mountains (csappbook.blogspot.com)]

Magnetic Disk Drive Mechanism

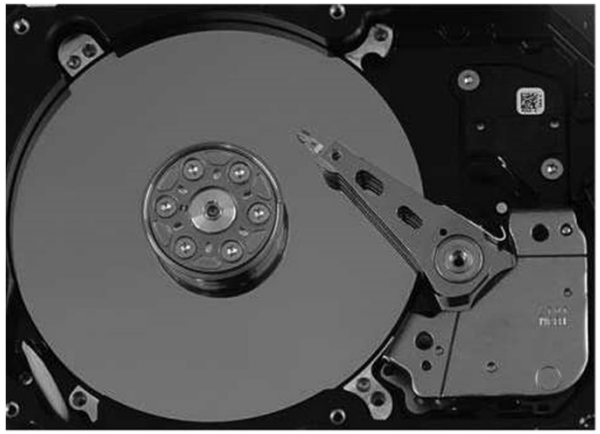
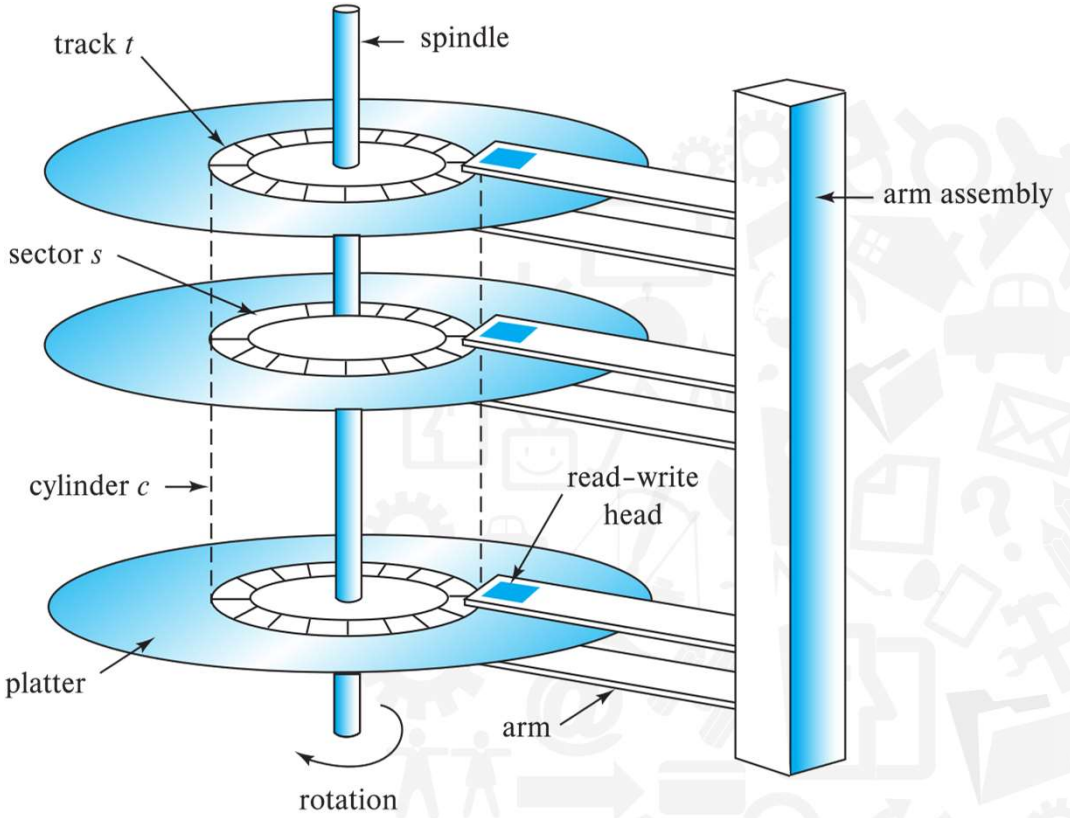


Photo of magnetic disk drive



Schematic diagram of magnetic disk drive

Magnetic Disk

- **Track**
  - Surface of platter is divided into circular tracks
- **Sector**
  - Each track is divided into sectors
  - Sector : the smallest unit of data that can be read or written
  - Sector size is typically 512 bytes
  - Typical sectors per track : 500 to 1000 on inner tracks to 1000 to 2000 on outer tracks
  - Block size is a multiple of sector size(which is fixed)
- Head-disk assembly
  - Multiple disk platters on a single spindle (1 to 5 usually)
  - One head per platter, mounted on a common arm
  - Cylinder *i* consists of *i*-th track of all the platters

Performance Measures

- Access time – time it takes from when a read or write request is issued to when data transfer begins
  - Seek time : time it takes to reposition the arm over the correct track
    - Avg. seek time is ½ the worst case seek time
    - 4 to 10 msec. on typical disks
  - Rotational latency : time it takes for the sector to be accessed to appear under the head
    - 4 to 11 msec. on typical disks (5400 to 15,000 rpm)
    - Avg. latency is ½ of the above latency
  - Data-transfer rate – the rate at which data can be retrieved from or stored to the disk
    - 25 to 200 MB/sec, lower for inner tracks

Performance Measures(cont'd)

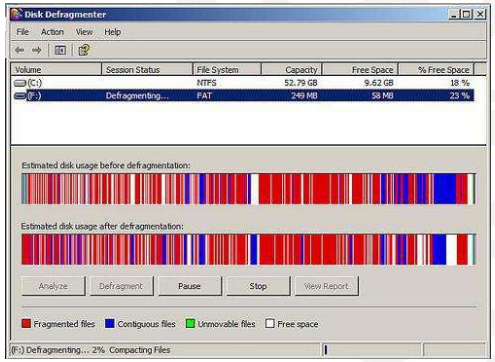
- **Disk block** is a logical unit for storage allocation and retrieval
  - 4 to 16 kilobytes typically
    - Smaller blocks : more transfers from disk
    - Larger blocks: more space wasted due to partially filled blocks
- **Sequential access pattern**
  - Disk seek requires only for the first block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

Performance Measures(cont'd)

- **Mean time to failure(MTTF)**
  - Avg. 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, corr. To a theoretical MTTF of 500,000 to 1,200,000 hours for a new disk
    - E.g., (literally) an MTTF of 1,200,000 hours for a new disk means that given 1,000 relatively new disks, on an average one will fail every 1,200 hours.
  - MTTF decreases as disk ages

Arranging pages on Disk

- Cost of accessing a disk block is not constant
- Different cost of access ‘next’ block:
  - Blocks on adjacent locations on the same track
  - Blocks on same track, or same cylinder
  - Blocks on adjacent cylinders, or distant cylinders
- If blocks in a file could be arranged sequentially on disk, seek and rotational delay would be minimized
  - E.g., defragmentation

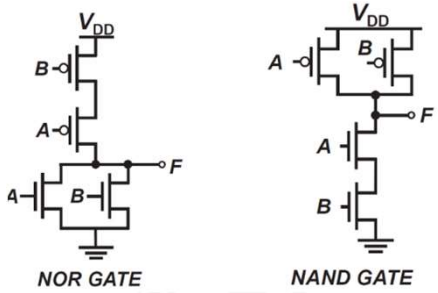


You must have seen this on your PC before

Latency lags bandwidth

- From 1983 to 2022
  - Capacity increases ~ 600,000 times (0.03GB → 18TB)
  - Bandwidth improved ~1,000 times(0.6 MB/s → 600MB/s)
  - Latency improved ~11.6 times (48.3 → 4.16ms)
- Why?
  - Moore's law helps bandwidth more than latency
  - Distance limits latency
  - Bandwidth is generally easier to sell
  - Latency helps bandwidth but not vice versa.(e.g., spinning disk faster)
  - Bandwidth hurts latency(e.g., buffer)
  - OS overhead hurts latency

Flash Storage



- **NAND flash** storage
  - Used widely for storage, cheaper than memory
    - 1 GB for 142 won(¢12) now, 3,619 won in 2010
  - Successfully occupied the position btw. memory and HDD

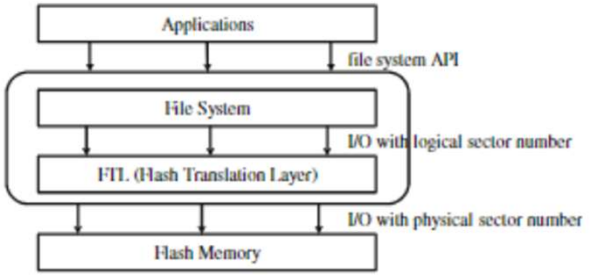
- **SSD(Solid State Disk)**
  - Use standard block-oriented disk interfaces, but store data on multiple flash storage devices internally
  - Transfer rate of up to 500MB/sec using SATA and up to 3GB/sec using NVMe PCIe

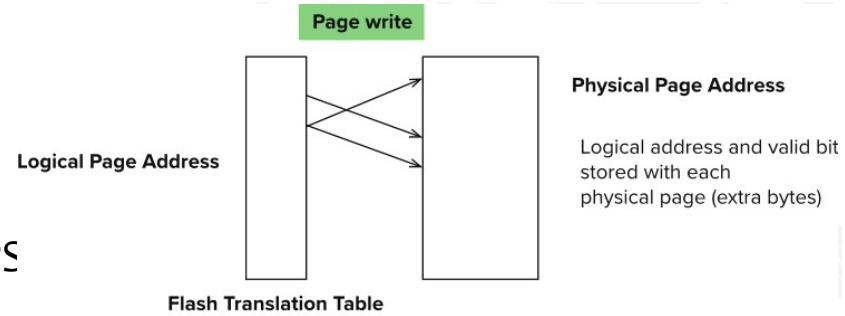
Features of NAND flash storage

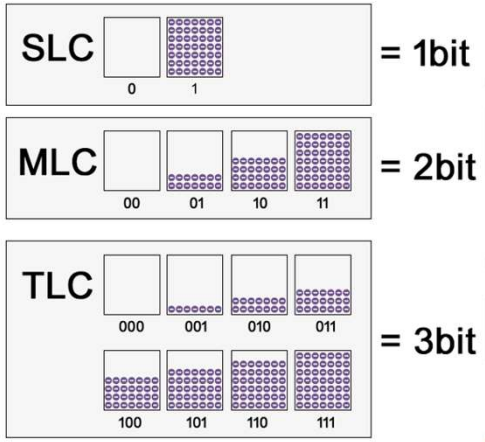
- No mechanical latency
  - Provides uniform random access speed without seek/rotational latency
- Page-at-a-time read(page: 512B to 4 KB)
- Page can only be written once
  - Must be erased to allow rewrite
- **No in-place update**
  - No data item or page can be updated in place without erasing it first
    - Erase unit (typically 256KB to 1MB , 128 to 256pages)is much larger than a page
- Limited life time
  - MLC : 0.1M times of writes, SLC: 1M times of writes guaranteed
  - After that, erase block becomes unreliable
  - **Wear-leveling** is required

Features of NAND flash storage(Cont'd)

- Asymmetric read & write speed
  - Read speed is typically at least 2X faster than write speed
  - Write (and erase) optimization is critical
- Asymmetric seq. vs. random I/O performance
  - E.g., Samsung 980 M.2 NVMe
    - Seq. read 2,900M/sec write 1,300M/sec
    - Random read 230K IOPS, write 320K IOPS
      - 920 M/sec, 1,280M/sec in total size
- "Disk" Abstraction
  - Remapping of logical page addresses to physical page addresses avoids waiting for erase
  - LPA→ (channel#, plane#)
- **FTL(Flash translation table)** tracks mapping
  - Also stored in a label field of flash page
  - Remmoing carried out by FTL







RAID

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

Improvement of Reliability via Redundancy

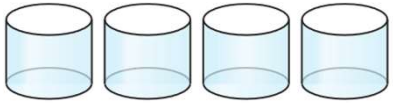
- **Redundancy** – store extra information that can be used to rebuild information lost in a disk failure
- e.g., **Mirroring** (or **shadowing**)
  - **Duplicate every disk**. Logical disk consists of two physical disks.
  - Every write is carried out on both disks
    - Reads can take place from either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

Improvement in Performance via Parallelism

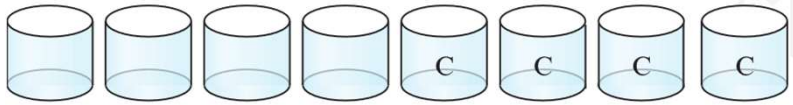
- Two main goals of parallelism in a disk system:
  1. Load balance multiple small accesses to increase throughput
  2. Parallelize large accesses to reduce response time.
- Improve transfer rate by striping data across multiple disks.
- **Bit-level striping** – split the bits of each byte across multiple disks
  - In an array of eight disks, write bit *i* of each byte to disk *i*.
  - Each access can read data at eight times the rate of a single disk.
  - But seek/access time worse than for a single disk
    - Bit level striping is not used much any more
- **Block-level striping** – with *n* disks, block *i* of a file goes to disk (*i* mod *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.



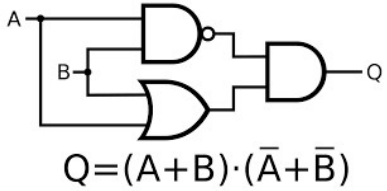
(a) RAID 0: nonredundant striping



(b) RAID 1: mirrored disks

RAID Levels (Cont'd)

- **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

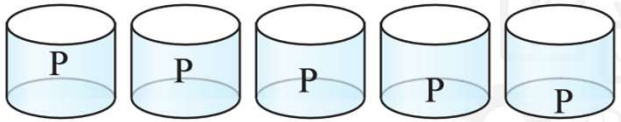


A	B	A ^ B
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

- To recover data for a block, compute XOR of bits from all other blocks in the set including the parity block
  - e.g., DISK 1 = 1111 , DISK 2 = 1110, DISK 3 = 1100 , DISK 4 = 1000
  - If we work on the first bit of disk 1-4, its parity bit1 is computed by xor 1 xor 1 xor 1 = 0
  - Assume disk 3 is damaged, then damaged bit can be recovered by computing 1 xor 1 xor 1 xor 0 = 1

RAID Levels (Cont'd)

- **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 mod* 5) + 1, with the data blocks stored on the other 4 disks.



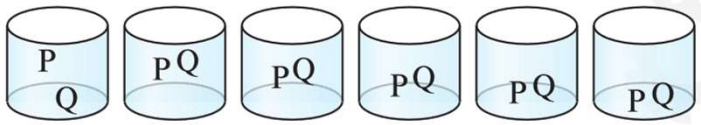
(c) RAID 5: block-interleaved distributed parity

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

  - Block writes occur in parallel if the blocks and their parity blocks are on different disks

RAID Levels (Cont'd)

- **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'd)

- 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

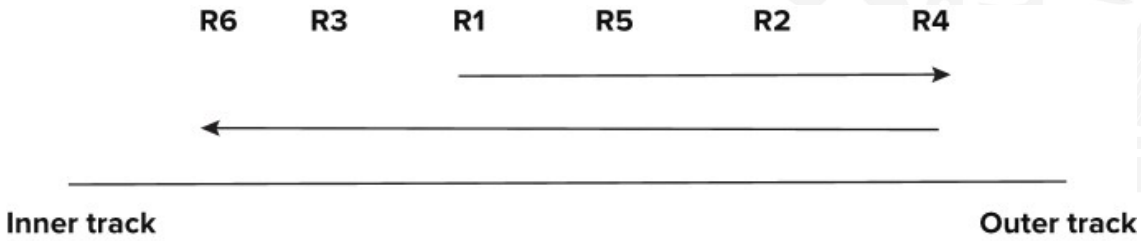
- RAID 0 is used only when data safety is not important
  - e.g., data can be recovered quickly from other sources

Choice of RAID Level(Con'td)

- Level 1 provides much better write performance than level 5
- 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.

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'd)

- **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
- **Non-volatile write buffers**

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, . . ., *n* to *i*, . . . , *n* – 1**
  - move record *n* to *i*
  - do not move records, but link all free records on a *free list*

e.g., if 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 $i + 1, \ldots, n$ to $i, \ldots, n - 1$
  - **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

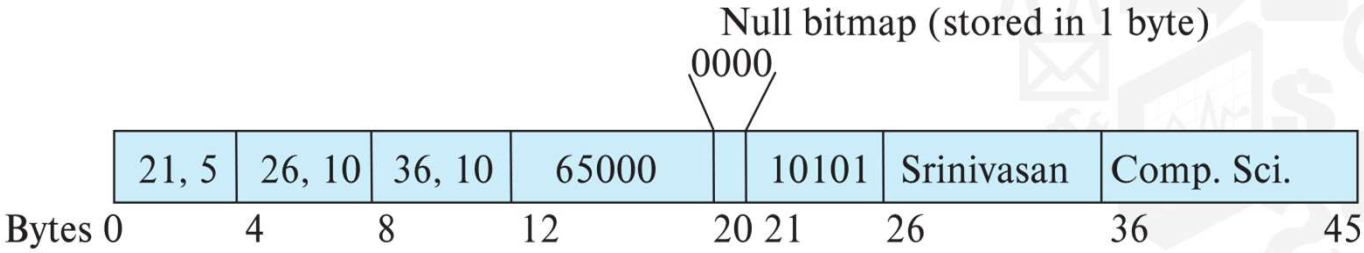
Fixed-Length Records

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

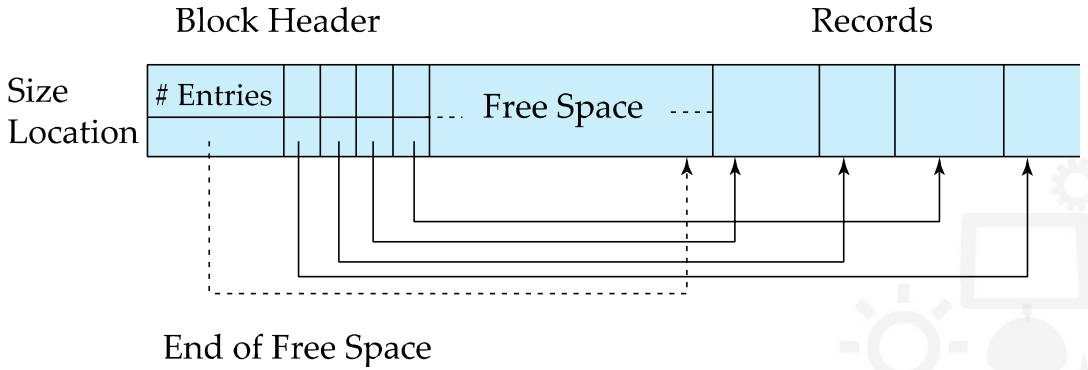
header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology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

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 (e.g., **varchar**)
  - Record types that allow repeating fields (used in some older data models).
- Attributes are stored in order
- Variable length attributes represented by fixed size (offset, length), with actual data stored after all fixed length attributes
- Null values represented by null-value bitmap



Variable-Length Records: Slotted Page Structure



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

Storing Large Objects

- e.g., blob/clob types
- Note: records must be smaller than pages. Then how can we handle the types?
- Alternatives:
  - Store as files in file systems
  - Store as files managed by database
  - Break into pieces and store in multiple tuples in separate relation
    - PostgreSQL TOAST(The Oversized-Attribute Storage Technique)

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
- **Multitable clustering file organization**
  - records of several different relations can be stored in the same file
  - Motivation: store related records on the same block to minimize I/O
- **B$^+$-tree file organization**
  - Ordered storage even with inserts/deletes
  - More on this in the Lecture 8
- **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 Lecture 8

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

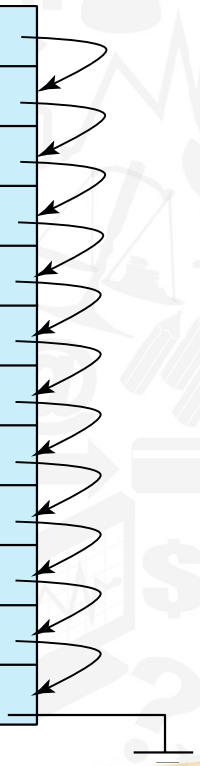
4	7	2	6

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

Sequential File Organization

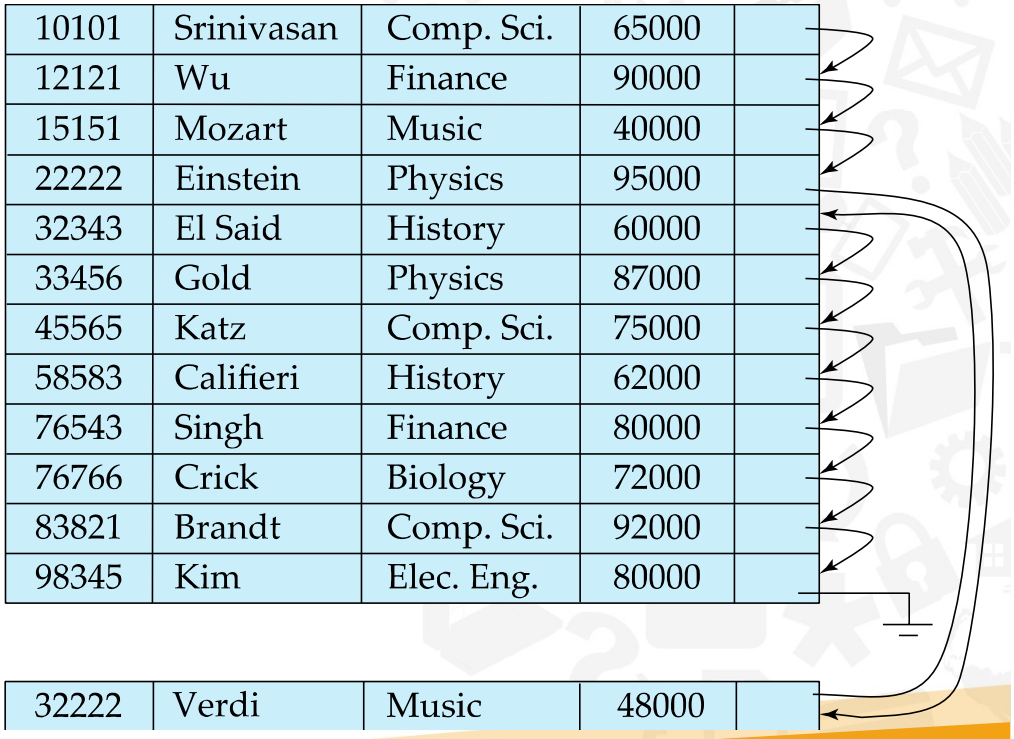
- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a search-key

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	



Sequential File Organization (Cont'd)

- 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



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	

32222	Verdi	Music	48000	

Multitable Clustering File Organization

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

department

dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Physics	Watson	70000

instructor

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

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 of *department* and *instructor*

- 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 in needed
- Partitioning
  - Reduces costs of some operations such as free space management
  - Allows different partitions to be stored on different storage devices
    - E.g., *transaction* partition for current year on SSD, for older years on magnetic disk

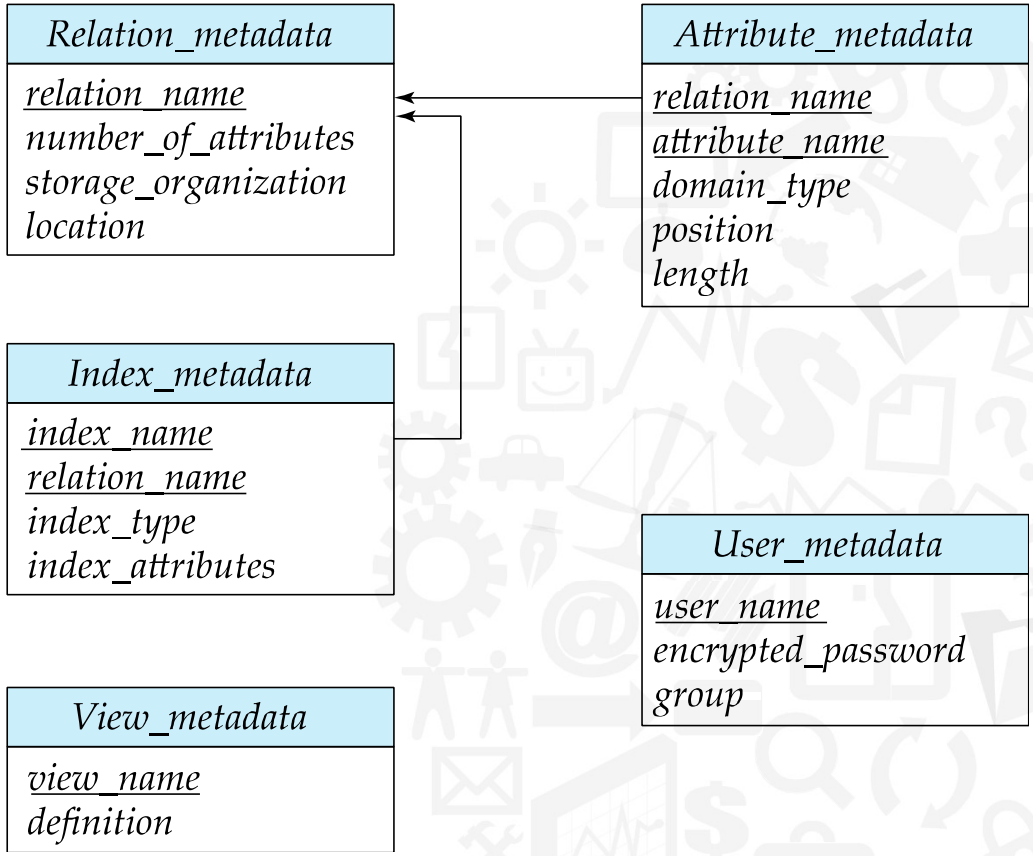
Data Dictionary Storage

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

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

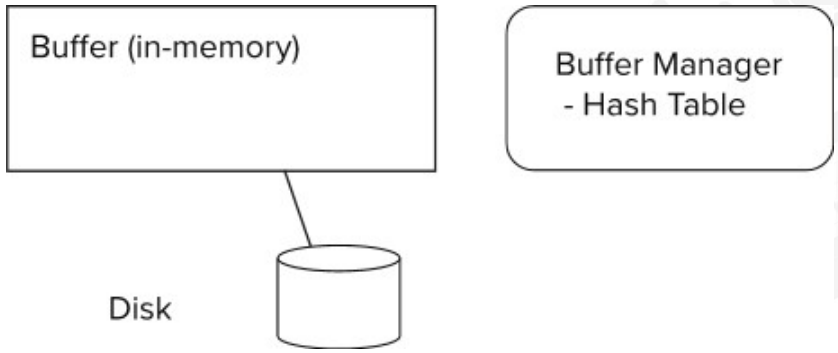
Relational Representation of System Metadata

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



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

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

Buffer-Replacement Policies

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

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

Buffer-Replacement Policies (Cont'd)

- **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

- Buffer managers support **forced output** of blocks for recovery (more in Lecture 11)
- **Nonvolatile write buffers** speed up disk writes by writing blocks to a non-volatile RAME 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 RAME
    - Write to log disk is very fast since no seeks are required
- **Journaling file systems** write data in-order to NVRAM or log disk
  - Reordering without journaling : risk of corruption of file system data

Column-Oriented Storage

- Row-Oriented : rows stored (usually) in sequential in a file

Key	Fname	Lname	State	Zip	Phone	Age	Sales
1	Bugs	Bunny	NY	11217	(123) 938-3235	34	100
2	Yosemite	Sam	CA	95389	(234) 375-6572	52	500
3	Daffy	Duck	NY	10013	(345) 227-1810	35	200
4	Elmer	Fudd	CA	04578	(456) 882-7323	43	10
5	Witch	Hazel	CA	01970	(567) 744-0991	57	250

- Column-oriented
  - Also known as colurmnar representation
  - Store each attribute of a relation separately
  - Example

Key
1
2
3
4
5

Fname
Bugs
Yosemite
Daffy
Elmer
Witch

Lname
Bunny
Sam
Duck
Fudd
Hazel

State
NY
CA
NY
CA
CA

Zip
11217
95389
10013
04578
01970

Phone
(123) 938-3235
(234) 375-6572
(345) 227-1810
(456) 882-7323
(567) 744-0991

Age
34
52
35
43
57

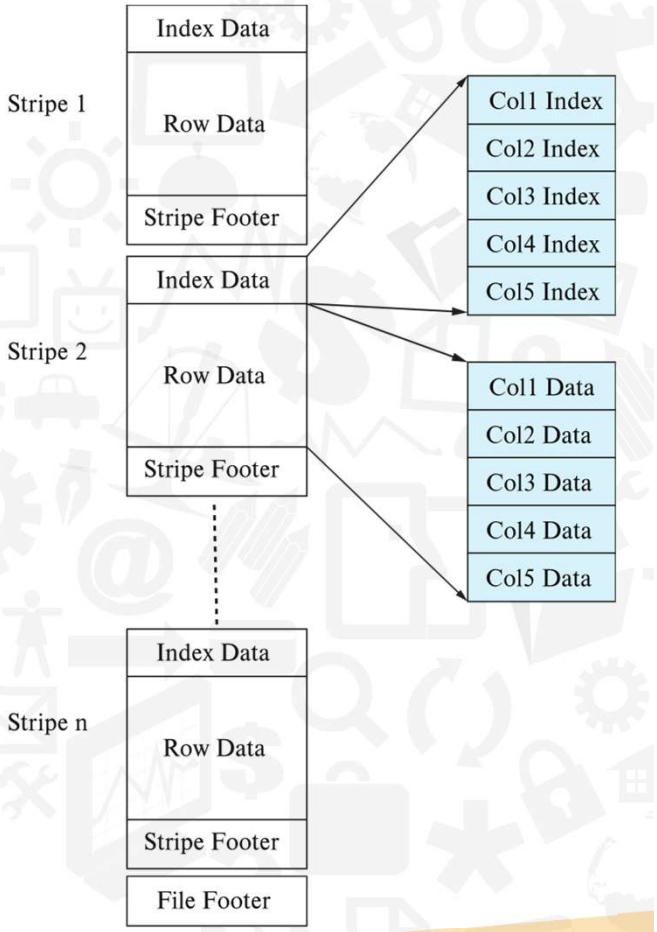
Sales
100
500
200
10
250

Columnar Representation

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

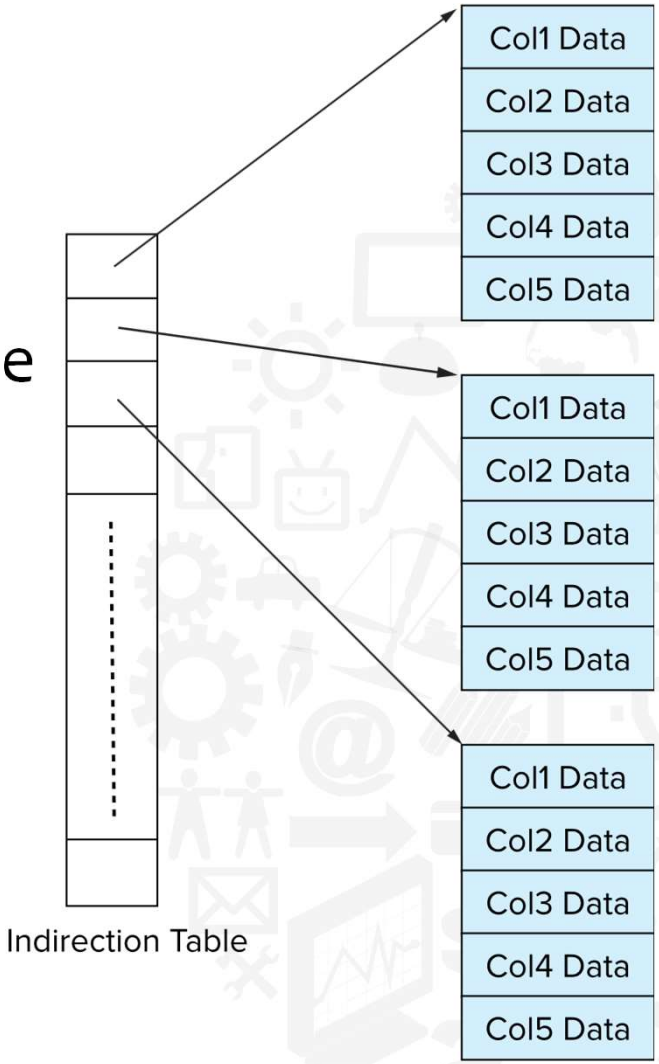
Columnar File Representation

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



Storage Organization in Main-Memory Databases

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



Question?

"Bart Simpson, will you stop raising your hand? You haven't gotten one right answer all day!"
–source: https://www.fox.com/the-simpsons

-