Contents

2.2 Node-level

2.2.1 Server (computation, HW accelerators)

A **Server** is a computing system designed to manage, process, and deliver data or services to other computers (clients) over a network. In the context of datacenters and Warehouse-Scale Computers (WSCs), servers are the **atomic units of computation**, the fundamental building blocks of the entire system architecture.

Though **conceptually similar to a desktop PC**, servers **differ** in critical ways:

- They are **significantly more powerful**, scalable, and modular.
- They are designed for **continuous operation**, high availability, and **dense physical packaging** within a rack structure.

Servers in modern datacenters must balance **performance**, **density**, **power efficiency**, and **maintainability**.

Server Types

Server form factors **define how** servers are **physically organized** and **deployed** in datacenter environments. There are three principal types:

1. **Tower Server**s (section 2.2.1.1, page 27). Resemble traditional desktop PCs. They are ideal for small-scale deployments or low-density use cases.
   - ✔ **Pros**: Easy to upgrade, good cooling, low cost.
   - ✖ **Cons**: Large footprint, not optimized for rack deployment.
2. **Rack Server**s (section 2.2.1.2, page 28). Designed to slide **into a rack** (standardized shelves) in units (U); e.g., 1U = 1.75 inches. It is the most common server format in datacenters.
   - ✔ **Pros**: High compute density, easy to cable and scale.
   - ✖ **Cons**: Requires dedicated infrastructure (rack, cooling, power).
3. **Blade Server**s (section 2.2.1.3, page 30). Extremely compact and multiple blades share power, cooling, and networking through a **blade enclosure**. It is excellent for environments where space and energy are at a premium.
   - ✔ **Pros**: Highest density and modularity, centralized management.
   - ✖ **Cons**: Higher initial cost, vendor lock-in, increased heat density.

Server Architecture

Servers are typically **integrated into a tray or blade enclosure**, which contains:

- **Motherboard**: The central PCB that **interconnects all components**.
- **Chipset**: **Manages data flow** between CPU, RAM, storage, and peripherals.
- **Expansion slots**: For GPUs, network cards, and other **accelerators**.

Servers in WSCs tend to **use homogeneous hardware/software platforms** to simplify large-scale orchestration and maintenance.

Server Architecture

The **Motherboard** acts as a **central nervous system for the server**, it hosts:

- **CPU sockets** (e.g., up to 2 for dual Xeon systems)
- **DIMM slots** for RAM
- **Storage connectors** (e.g., SATA, NVMe)
- **NIC slots** (Network Interface Cards)

This level of configurability allows tailoring servers for compute-heavy, memory-bound, or I/O-intensive applications.

2.2.1.1 Tower Server

A **Tower Server** is a type of server designed in a vertical, standalone chassis that closely resembles a **standard tower desktop computer**. Unlike blade or rack servers, which are designed for high-density environments, tower servers prioritize **simplicity and accessibility**, often at the cost of physical footprint.

- **Structure**: Independent, **vertical** case (not meant for rack mounting).
- **Deployment**: Common in small businesses, branch offices, or settings where only a few servers are needed.
- **Internal layout**: Lots of **space for expansion components** like disks or PCIe cards.

Advantages

- ✔ **Scalability & Ease of Upgrade**. **Easy** to open and **upgrade**, users can add storage, memory, or cards as needed.
- ✔ **Cost-Effective**. Usually the **cheapest server type**, suitable for budget-constrained environments.
- ✔ **Easy Cooling**. Due to **low component density**, natural airflow is often sufficient. Less need for specialized cooling systems.

Limitations

- ✖ **Space Consumption** Tower servers consume **significant physical space** and don't scale well in quantity.
- ✖ **Basic Performance** They usually **offer lower performance and redundancy** compared to enterprise-grade rack or blade servers.
- ✖ **Cable Management** Not ideal for structured environments, cables can become messy and hard to manage.

2.2.1.2 Rack Servers

A **Rack Server** is a server built specifically to be **mounted vertically in standardized racks**, which are metallic shelves designed to hold multiple servers and IT components. Rack servers are the **default choice** in medium to large-scale datacenters, balancing compute density, modularity, and serviceability.

Physical Standardization

- **Servers are stored in racks** which follow a global standard:
  - **1U (Rack Unit)** = 1.75 inches (44.45 mm) in height.
  - Servers may come in 1U, 2U, 4U, up to 10U formats depending on power and component density.
- Racks also house other components: networking switches, storage arrays, power distribution units (PDUs), and cooling units.

This **standardization allows for efficient vertical stacking** of servers, optimizing physical space and simplifying cabling.

Racks as More Than Just Shelves

A rack is not just a mechanical holder, it is **part of the power, networking, and management infrastructure of the datacenter**:

- **Power Infrastructure**:
  - Shared power distribution units.
  - Battery backup (UPS).
  - Power conversion units.
- **Networking**:
  - Top of Rack (ToR) switches connect all servers in the rack to the datacenter network fabric.
  - Simplifies cabling and reduces latency.
- **Cooling**: designed for front-to-back airflow, aligned with datacenter cooling strategy (e.g., cold aisle containment).
- **Dimensions** can vary, but the classic rack is 19 inches wide and up to 48 inches deep.

Advantages

- ✔ **Modularity**: Individual servers can be **hot-swapped**, upgraded, or replaced **without disrupting others**.
- ✔ **Failure Containment**: Easy to **isolate and service a failed node** without bringing down the system.
- ✔ **Cable Management**: Organized by rear/backplanes or Top-of-Rack (ToR) switches.
- ✔ **Cost-Efficient Scaling**: **Scales vertically** at relatively lower incremental cost compared to other formats.

Challenges

- ✖ **High Power Demand**: **Higher component density** requires more energy and advanced cooling systems.
- ✖ **Thermal Hotspots**: Tight stacking can cause **hot zones**, especially with accelerator-heavy nodes.
- ✖ **Maintenance Overhead**: Large racks with tens of servers can become **complex to manage** physically as systems scale.

2.2.1.3 Blade Servers

Blade Servers represent the **most advanced evolution** in server form factors. They are designed to **maximize space efficiency** and **centralized manageability**, making them ideal for **large-scale enterprise datacenters** and **high-performance computing environments**.

A blade server is essentially a **stripped-down**, **ultra-thin server board** (the "blade") that fits into a blade enclosure, a shared chassis providing:

- Power
- Networking
- Cooling
- Centralized management

The enclosure conforms to the same **rack unit standard** (U), allowing it to integrate seamlessly with existing rack infrastructure. We can think of a blade system as a server equivalent of a modular bookshelf, where each "book" is a full server, and the "bookshelf" provides shared power, ventilation, and data connectivity.

Advantages

- ✔ **Compactness & Density**: The **smallest physical form factor** among all servers, allowing high-density deployments within a minimal footprint.
- ✔ **Minimal Cabling**: The **shared backplane** removes the need for complex cabling; power and network connections are centralized.
- ✔ **Centralized Management**: Blade systems typically include **unified management interfaces** (e.g., iLO, iDRAC) to monitor and configure blades collectively.
- ✔ **Scalability & Reliability**: New **blades can be added** with minimal disruption; enclosures support **load balancing** and **failover mechanisms**.
- ✔ **Uniform Infrastructure**: Simplifies deployment with **shared cooling**, **network fabrics**, and **power redundancy**.

Disadvantages

- ✘ **High Initial Cost**: Blade enclosures and vendor-specific blades often demand **significant upfront investment**.
- ✘ **Vendor Lock-In**: Typically, only blades from the **same manufacturer** (e.g., HPE, Dell, Cisco) **are compatible** with a given enclosure.
- ✘ **Thermal Density**: The compact form causes **higher heat output per rack unit**, requiring advanced HVAC design and monitoring.
- ✘ **Limited Flexibility**: While modular, blade systems trade off flexibility for density, upgrades and replacements may be **constrained by the enclosure's architecture**.

2.2.1.4 Machine Learning

While Moore's Law historically predicted that transistor density would double every 18-24 months, the **growth in ML model complexity** has surpassed this pace. Since 2013, compute demand for AI training has doubled approximately **every 3.5 months**. This exponential curve far exceeds the capabilities of general-purpose CPUs, triggering a renaissance in specialized hardware.

What is Machine Learning?

At its core, **Machine Learning (ML)** refers to **computational methods** that enable systems to **learn from data** without being explicitly programmed. Rather than defining rules manually, ML allows a system to build a model from **patterns observed in examples**.

In supervised learning:

- A system learns a **target function** $y = f(x)$ that **maps inputs** (features) **to outputs** (labels).
- This is **done using a training dataset** $(x_1, y_1), \ldots, (x_N, y_N)$, and the model is later tested on unseen inputs.

Applications include: classification (e.g., cat vs. dog), regression (e.g., predicting flight delays), image recognition, speech synthesis, fraud detection, etc.

Artificial Neural Networks (ANNs)

Artificial Neural Networks (ANNs) are a **subset of ML models** inspired by the human brain. They consist of **layers of interconnected neurons**, including:

- **Input layer**: receives the data
- **Hidden layers**: transform data using weighted functions and nonlinear activations
- **Output layer**: produces the prediction

The key learning mechanisms are:

- **Backpropagation**: adjusts weights based on the error between prediction and actual target.
- **Gradient descent**: optimizes the model parameters iteratively.

Hardware Acceleration: Why ML Needs More Than CPUs

Modern ML, particularly **deep learning**, is computationally expensive. Training models like GPT or ResNet involves processing **billions of parameters** across massive datasets. To meet these demands, **Warehouse-Scale Computers (WSCs) integrate specialized accelerators such as**:

- **Graphics Processing Units (GPUs)**. GPUs are highly parallel processors originally designed for graphics rendering but are now extensively used for ML because they:
  - Execute the **same operation across many data elements in parallel** (SIMD).
  - Accelerate matrix operations central to deep learning.
  - Support ML frameworks via CUDA, OpenCL, OpenMP, SYCL, etc.

  GPUs are often housed in **PCIe-attached trays**, interconnected via NVLink or NVSwitch for ultra-fast data exchange.

  Distributed training across multiple GPUs requires **low-latency**, **high-bandwidth interconnects**. Performance can also be bottlenecked by slowest learner or network synchronization delays.

- **Tensor Processing Units (TPUs)**. Developed by Google, TPUs are **domain-specific architectures** designed **specifically for ML workloads**. TPU generations:
  - **TPUv1**: Inference-only, connected via PCIe.
  - **TPUv2**: Supports both training and inference; includes MXUs (matrix units) and high-bandwidth memory (HBM).
  - **TPUv3**: Liquid-cooled, supercomputing-class performance. Up to 100 PFLOPS per pod.
  - **TPUv4/TPUv5**:
    - v4 pod: 4096 devices
    - v5e: cost-efficient variant
    - v5p: high-performance variant scalable to 8000+ devices
    - Used in global data centers since 2023

  A **TPU Pod** aggregates **hundreds of TPU cores with shared memory** and custom high-speed networks for massive parallelism.

- **Field-Programmable Gate Arrays (FPGAs)**. FPGAs offer **customizable digital logic** that can be reprogrammed after manufacturing.
  - Flexible hardware, can be reconfigured for different algorithms.
  - Suitable for:
    - Network acceleration
    - Security tasks (e.g., encryption)
    - Data analytics
    - Specialized ML inference

  For example, Microsoft Azure integrates FPGAs for infrastructure efficiency, lowering carbon footprint and improving hardware reuse.

Feature	GPU	TPU	FPGA
Purpose	General-purpose ML compute	ML-specific acceleration (esp. DL)	Flexible, reconfigurable hardware
Programmability	CUDA, OpenCL, etc.	TensorFlow, PyTorch (high-level)	VHDL, Verilog (low-level, HDL)
Flexibility	High	Medium (optimized for tensors)	Very high (reprogrammable)
Efficiency	Good	Excellent (for tensor ops)	Excellent (for specific pipelines)
Use Case	Training + Inference	Training + Inference	Offloading, network, analytics

Table 3: Summary: GPU vs TPU vs FPGA.

	Advantages	Disadvantages
CPU	✔ Easy to be programmed and support any programming framework. ✔ Fast design space exploration and run your applications.	✖ Suited only for simple AI models that do not take long to train and for small models with small training set.
GPU	✔ Ideal for applications in which data need to be processed in parallel like the pixels of images or videos.	✖ Programmed in languages like CUDA and OpenCL and therefore provide limited flexibility compared to CPUs.
TPU	✔ Very fast at performing dense vector and matrix computations and are specialized on running very fast programming based on Tensorflow.	✖ For applications and models based on the TensorFlow. ✖ Lower flexibility compared to CPUs and GPUs.
FPGA	✔ Higher performance, lower cost and lower power consumption compared to other options like CPUs and GPU.	✖ Programmed using OpenCL and High-Level Synthesis (HLS). ✖ Limited flexibility compared to other platforms.

Table 4: Comparison of CPU, GPU, TPU and FPGA.

2.2.2 Storage (type, technology)

Data has significantly grown in the last few years due to sensors, industry 4.0, AI, etc. The growth favours the **centralized storage strategy** that is focused on the following:

- **Limiting redundant data**
- **Automatizing replication and backup**
- **Reducing management costs**

The *storage technologies* are many. One of the oldest but still used is the **Hard Disk Drive (HDD)**, a magnetic disk with mechanical interactions. However, the mechanical nature of HDDs imposes physical limits on access speed and reliability. In contrast, **Solid-State Drive (SSD)**, which lack moving parts and are built using NAND flash memory, offer significantly faster access times and better durability. The **Non-Volatile Memory express (NVMe)** also exists, which is the **latest industry standard** for running PCIe[2] SSDs.

In terms of cost per terabyte, NVMe drives are currently the most expensive (typically €100-200 for 1 TB), followed by SSDs (€70-100), while HDDs remain the most economical option (€40-60). This price-performance hierarchy makes hybrid storage architectures (HDD + SSD) increasingly appealing:

- A speed storage technology (**SSD or NVMe**) as **cache** and **several HDDs for storage**. It is a combination used by some servers: a small SSD with a large HDD to have a faster disk.
- Some HDD manufacturers produce Solid State Hybrid Disks (SSHD) that combine a small SDD with a large HDD in a single unit.

[2]**PCIe (peripheral component interconnect express)**. is an interface standard for connecting high-speed components

2.2.2.1 Files

The operating system views the disk as a **flat collection of independently addressable data blocks**. Each **block is assigned a unique LBA (Logical Block Address)**, enabling efficient data referencing and organization. To streamline access and reduce management overhead, the **OS typically groups these blocks into clusters**, larger units that serve as the minimum granularity for disk I/O operations.

Clusters typically range in size from a single disk sector (512 bytes or 4 KB) up to 128 sectors (64 KB), depending on the file system configuration. Each cluster may store either:

- **File data**. The actual contents of user files.
- **Metadata**. System-level information necessary to support the file system,:
  - File and directory names
  - Folder hierarchies and symbolic links
  - Timestamps (creation, modification, access)
  - Ownership and access control data
  - **Links to the LBA where the file content can be located on the disk**

Consequently, the **disk space is divided into different cluster types** based on their purpose:

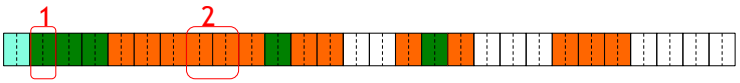
- Metadata:
  - Fixed-position metadata clusters, used to initialize and mount the file system
  - Variable-position metadata clusters, which manage directories and symbolic links
- File data clusters, containing the actual contents of files
- Unused clusters, which are free and available for future allocations

Figure 4: A cluster can be seen visually as an array. In this image, for example, we've shown three types of cluster: metadata fixed position (azure), metadata variable position (green), file data (orange), unused space (white).

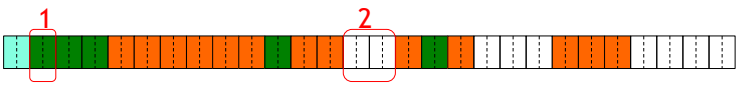
The following explanation introduces some basic operations on the files to see what happens inside the disks.

- **File Reading**
  1. **Access the Metadata**. Before the system knows **where** the file is stored on disk, it must find information about the file, called metadata. This metadata is stored in **variable-position metadata clusters**. These clusters are not always in the same place on disk, they move and grow as the system evolves.
  2. **Locate the File's Data Blocks (clusters)** and **Read the Actual Content**. Once the OS has the metadata, it now knows:
     - Which **Logical Block Addresses (LBAs)** contain the data.
     - **How many clusters need to be read** to get the full content.

     It uses this information to issue **read commands** to the disk controller. Finally, the **disk accesses the physical sectors or clusters** indicated by the LBAs and transfers that data into main memory (RAM).



- **File Writing**
  1. **Access Metadata to Find Free Space**. The operating system **first checks the file system's metadata** to **find a free area of disk space** where it can store our new data.
  2. Once free space is identified, the OS:
     - **Allocates** one or more clusters, depending on the file size
     - **Writes our data** into these clusters on the physical disk



Since the *file system can only access clusters*, the **actual space taken up by a file on a disk is always a multiple of the cluster size**. Given:

- s, the *file size*
- c, the *cluster size*

Then the **actual size on the disk** a can be calculated as:

$$a = \left\lceil \frac{s}{c} \right\rceil \times c \tag{1}$$

Where ceil rounds a number up to the nearest integer. It's also possible to calculate the **amount of disk space wasted by organising the file into clusters** (**wasted disk space** w):

$$w = a - s \tag{2}$$

A formal way to refer to wasted disk space is **internal fragmentation** of files.

Example 4: internal fragmentation

- File size: 27 byte
- Cluster size: 8 byte

The *actual size* on the disk is:

$$a = \left\lceil \frac{27}{8} \right\rceil \cdot 8 = \lceil 3.375 \rceil \cdot 8 = 4 \cdot 8 = 32 \text{ byte}$$

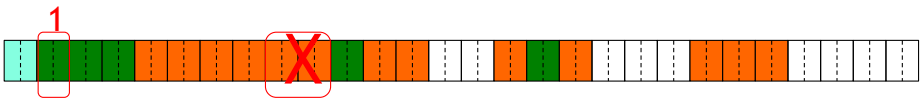
And the internal fragmentation w is:

$$w = 32 - 27 = 5 \text{ byte}$$

- **Deleting**

  1. The file system updates its metadata structures to:
     - **Remove the file name** from the directory
     - **Mark the clusters** where the file was stored as **free or available**
     - Optionally, update timestamps or record deletion events

     Importantly: **The data itself is not erased at this stage.** The actual bytes remain on disk until they are overwritten by another file.



- **External fragmentation**. It happens when there are **enough free clusters on the disk** to store a file, but **not all together (not contiguous)**. So, when the system tries to write a large file, it must **split it** into smaller parts and place them in **different, scattered locations** on the disk.

  Why does this happen? Over time, as files are created, deleted, resized, or moved, the disk becomes **less organized**. Clusters are freed in different spots, and the available space is no longer one big continuous area. So:

  - A new file cannot fit in one continuous chunk.
  - The OS stores it in **multiple non-adjacent clusters**.

  This is called **external fragmentation** because: the **fragmentation is not inside the file itself, but in the way its data is laid out externally across the disk**.

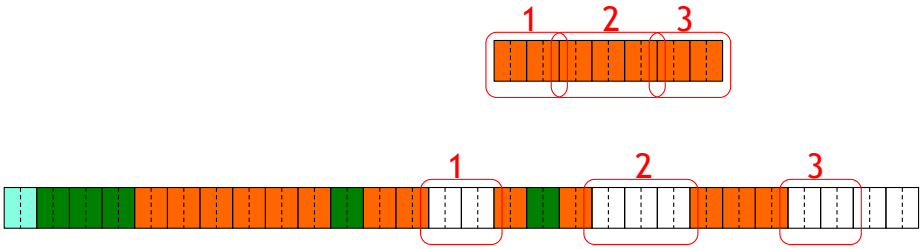


Figure 5: Each number (1, 2, 3) corresponds to a portion (chunk) of a file. A file that was meant to be stored as `[1][2][3]` (contiguously) has instead been broken into parts and stored in a scattered layout across different disk clusters. This situation represents **external fragmentation**, where the file system could not find a large enough continuous block of free space to store the file all together.

2.2.2.2 HDD

A **Hard Disk Drive (HDD)** is a **data storage device that uses rotating disks (platters) coated with magnetic material**.

Data is read randomly, meaning individual data blocks can be stored or retrieved in any order rather than sequentially.

An HDD consists of one or more rigid (*hard*) rotating disks (platters) with magnetic heads arranged on a moving actuator arm to read and write data to the surfaces.

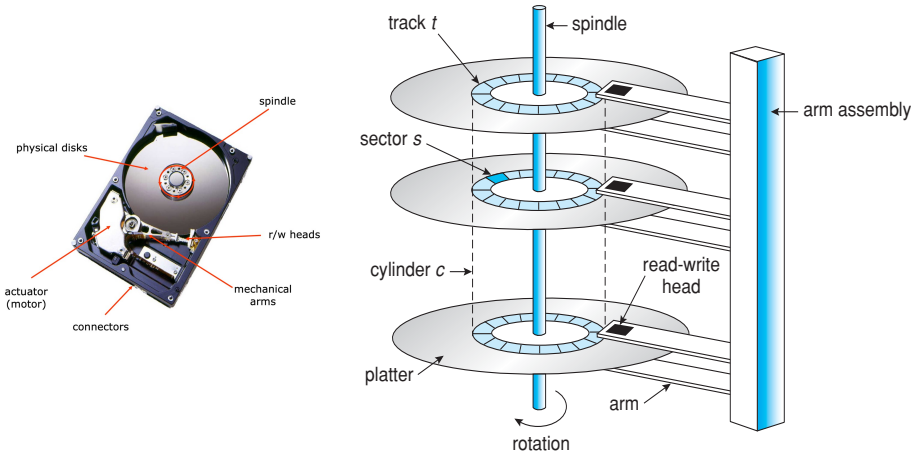


Figure 6: Hard Drive Disk anatomy.

Externally, hard drives expose a large number of **sectors** (blocks):

- Typically, 512 or 4096 bytes.
- Individual **sector writes** are **atomic**.
- Multiple sectors write it may be interrupted (**torn write**[3]).

The geometry of the drive:

- The sectors are arranged into **tracks**.
- A **cylinder** is a particular track on multiple platters.
- Tracks are arranged in concentric circles on **platters**.
- A disk may have multiple double-sided platters.

The **driver motor spins the platters at a constant rate**, measured in **Revolutions Per Minute (RPM)**.

[3]Torn writes happen when only part of a multi-sector update is written successfully to disk.

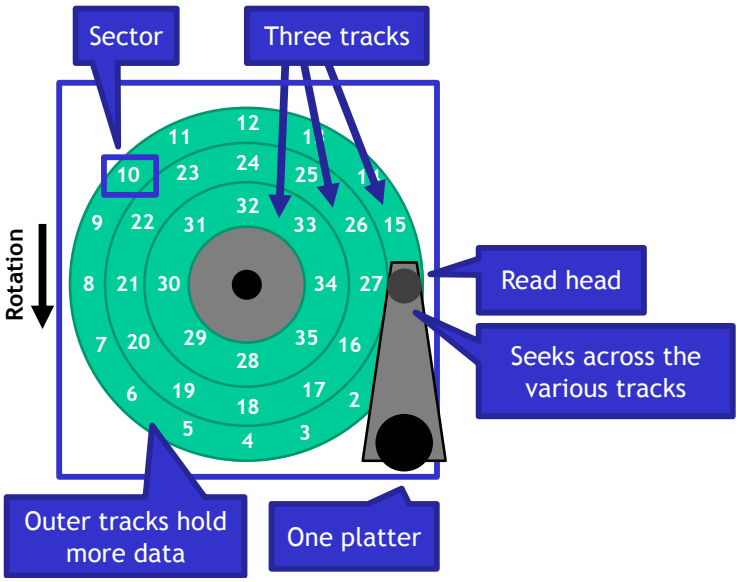


Figure 7: Example of HDD geometry.

The geometry of an HDD refers to the **physical layout** of data on the surface of a spinning disk. Figure 7 shows this:

- **One Platter**. The circular surface shown is a platter, which is a rigid, magnetic-coated disk where data is stored. Most HDDs have multiple platters stacked vertically, but here we focus on one for simplicity.
- **Tracks**. The platter is divided into concentric circles called tracks. Each track is a circular path that the read/write head can follow. In the figure 7, we see three tracks, each larger than the last, moving outward from the center.
- **Sectors**. Each track is divided into pie-like slices called sectors. A sector is the smallest physical unit that can be read or written on a disk (typically 512 bytes or 4 KB). The numbers (0 to 35) represent sector identifiers along the circular tracks. Note how the outer tracks contain more sectors because they have a larger circumference, so they can physically hold more data; this is known as Zone Bit Recording (ZBR), not mentioned in the course.
- **Read/Write head**. The read head is shown floating above the platter. It moves radially across the surface to switch from one track to another (this is called seek). Once on the correct track, the head waits for the desired sector to rotate beneath it (rotational latency), and then it reads/writes data.
- **Rotation and seek behavior**. The platter spins at high speed (e.g., 7200 RPM). As it rotates, the sectors pass under the stationary head. Data is accessed when the correct sector aligns with the head.

⚠ Types of Delay in disk Access

When a file is read from or written to a disk, especially an HDD, the total time taken is influenced by several delays. These are due to the mechanical and electronic processes involved in locating and transferring the data. Exists **four types of delay**:

- **Rotational Delay** (a.k.a. **Rotational Latency**) is the **time needed for the disk to rotate so that the desired sector aligns with the read/write head**. It depends on the RPM (Revolutions Per Minute) of the disk.
- **Seek Delay** is the **delay caused by the mechanical movement of the read/write head as it travels from one track to another** on a spinning disk. It is the dominant mechanical delay in many HDD operations, especially for random access patterns.

  How it works. Moving the head involves a sequence of physical phases:

  1. **Acceleration**: the actuator moves the head out of its resting position.
  2. **Coasting**: the head glides at a constant speed (if the distance is large).
  3. **Deceleration**: the actuator slows down to avoid overshooting.
  4. **Settling**: a short pause to stabilize the head at the desired track.

- **Transfer time** is the **time required to actually move the data**, once the head is correctly positioned over the desired sector. It's the final step in the I/O pipeline, where data is **read from or written to** the magnetic surface.

  What affects transfer time?

  1. **Rotational Speed (RPM)**, determines how quickly sectors pass under the head.
  2. **Data density**, more tightly packed data, more data per second read.
  3. **Size of the request**, reading more data takes more time.

  Even though it's much shorter than seek or rotation delays, **transfer time scales with data size**. For large sequential reads, transfer time becomes more relevant, especially when seek and rotation delays are minimized.

- **Controller Overhead** is the **non-mechanical delay introduced by the disk controller**, which is the hardware interface between the OS and the physical disk. It involves:
  - **Buffer management**: transferring data between disk and system memory (often using DMA).
  - **Interrupt processing**: informing the OS when the I/O operation is complete.
  - **Command translation**: converting OS-level I/O requests into device specific actions (e.g., SATA/NVMe commands).

– **Scheduling and queueing**: organize I/O operations for efficiency.

To see how these delays are calculated, we suggest you refer to the performance section 4.2, about HDD 4.2.1, page 130.

Cache

Hard Disk Drives (HDDs) are mechanical devices, and their performance is often limited by seek times and rotational latency. To partially overcome these limits, **many HDDs integrate a small** (8, 16, 32 MB) **amount of fast memory** (RAM) on the controller board, this is the **cache** or **track buffer**. The key functions of HDD cache are:

1. **Read Caching**. When the disk reads a block from the platter, it may also load **adjacent blocks** into the cache, expecting that they might be requested next (a technique known as **read-ahead**).

   Pros

   - ✔ If the next read request is for cached data, the disk can respond **instantly** from fast RAM.
   - ✔ This avoids mechanical movement and **cuts seek and rotational delays**.
   - ✔ Very effective for **sequential reads**.

2. **Write Caching**. There are two strategies here:

   (a) **Write-Back Cache** (Faster, Riskier). The HDD **reports the completion of the write** operation as soon as the data is in the cache, **before it is actually written to the platter**. Actual writing to disk happens later, in the background.

   Pros

   - ✔ Faster perceived performance
   - ✔ Useful in workloads with many small writes

   Cons

   - ✖ If power is lost before the data is flushed to disk, the **data is lost**. It is a **file system corruption risk**!
   - ✖ This is why it's considered **dangerous in critical systems** without battery backup or UPS.

   (b) **Write-Through Cache** (Safer, Slower). The HDD only reports the completion of the write operation **after the data has been fully written to disk**. Safer, but **slower**, since the OS must wait for the mechanical operation to finish.

3. Hybrid Cache: **Flash-Based Caching**. Some modern HDDs integrate **small flash memory** used for **persistent caching**:

   - Data stays even if the power goes out.
   - Combines the speed of SSD with the capacity of HDD.
   - Great for frequently accessed blocks (e.g., boot files, apps).

Disk Scheduling

While **caching** helps improve disk performance, it **doesn't eliminate the delays caused by seek and rotational latency**, especially during random access patterns. In systems with many I/O requests (like in a database or OS kernel), it's crucial to **choose the right order** to process requests to **minimize head movement**. This is where disk scheduling algorithms come in.

Instead of serving I/O requests in the order they arrive, the **disk controller (or OS)** can render them to improve efficiency. It is possible:

- Because every **disk request** includes the **target position** (i.e., the cylinder/track).
- So we can **estimate the cost (seek time)** of each request and choose the most efficient order.

Common disk scheduling algorithms are:

1. **First Come, First Serve (FCFC)** (the worst)

   How it works? Requests are handled in the **order they arrive**. No optimization, simple queue processing.

   Pros

   ✔ (Naive) Easy to implement

   Cons

   ✘ Can lead to **long seek distances** (i.e., inefficient head movement)

2. **Shortest Seek Time First (SSTF)** (great performance, but watch out for starvation)

   How it works? Always serve the request **closest to the current head position**.

   Pros

   ✔ Minimizes **total head movement**.

   ✔ Efficient in practice.

   Cons

   ✘ Can lead to **starvation**: distant requests may never be served if new close requests keep arriving.

3. **SCAN (Elevator Algorithm)** (good performance as SSTF, but fairer)

   How it works? The head **moves in one direction** (like an elevator), serving all requests in that direction. When it reaches the end, it **reverses direction**.

   Pros

   ✔ Good worst-case behavior.

   ✔ **No starvation**, every request will eventually be served.

Cons

- ✗ Requests at **edges** of the disk may have **longer wait times**.

4. **Circular SCAN (C-SCAN)** (fair, but less efficient than SCAN in some cases)

   How it works? Like SCAN, but head **only moves in one direction**. When it reaches the end, it **jumps back** to the beginning (like a circular elevator).

   Pros

   - ✓ More **uniform wait time** for all requests.

   Cons

   - ✗ Longer total movement (because of the jump).

5. **C-LOOK** (smart compromise between performance and fairness)

   How it works? Like C-SCAN, but instead of going to the physical end of the disk, th **head only goes as far as the last request in that direction**, then **jumps back to the smallest request**.

   Pros

   - ✓ **Saves movement** compared to full C-SCAN.
   - ✓ Still **avoids starvation**.

Algorithm	Fairness	Risk of Starvation	Strategy
FCFS	✓ Yes	✗ None	Serve in arrival order
SSTF	✗ No	⚠ Possible	Serve closest request
SCAN	✓ Yes	✗ None	Elevator (back & forth)
C-SCAN	✓ Yes	✗ None	One-direction sweep (circular)
C-LOOK	✓ Yes	✗ None	One-direction sweep (limited)

Table 5: Scheduling Algorithms.

2.2.2.3 SSD

A **Solid-State Drive (SSD)** is a **non-volatile storage device** that retains data without power. Unlike traditional Hard Disk Drives (HDDs), an SSD has **no mechanical parts**, *no spinning platters*, *no moving heads*. Internally, it's made of **transistors**, similar to those found in CPUs and RAM. It includes a **controller**, which manages read/write operations, wear leveling, garbage collection, and interface emulation. SSDs often adopt HDD-compatible interfaces (e.g., SATA, PCIe/NVMe) and form factors (2.5", M.2) for backward compatibility. Offers **higher performance** than HDDs, especially in **random access latency** and IOPS.

Flash Cell Technologies: Storing Bits in NAND

Modern SSDs use **NAND Flash Memory**, where data is stored in memory **cells**. Each cell can store one or more bits:

Cell Type	Bits per Cell	Characteristics
SLC	1	Fastest, most durable (up to 100k cycles), expensive
MLC	2	Slower than SLC, less durable, more dense
TLC	3	Used in most consumer SSDs; cheaper
QLC	4	High density, lower endurance
PLC	5	Experimental/extremely high density, low endurance

Increasing the number of bits per cell improves **capacity and cost-efficiency**, but **reduces endurance** and **performance**. Each cell type requires more precise voltage thresholds and incurs **more error correction and wear**.

Internal Organization

Solid-State Drives are built on NAND flash memory, which is **internally structured in a hierarchical manner** to optimize storage density and access efficiency.

- **Cell**: Basic storage unit (SLC, MLC, TLC, etc.) storing bits via trapped electrons.
- **Page**: The **smallest unit that can be read or written**. Typically 4-16 KB.

  Pages can be:

  - ✔ **Valid (In-use)**, contain active, readable data.
  - ✖ **Dirty (Invalid)**, hold obsolete or overwritten data.
  - – **Empty (Erased)**, ready to be programmed (written).
- **Block**: The **smallest unit that can be erased**. Usually contains 64-256 pages.

A block might have a capacity of 128-256 KB, composed of many smaller pages. Each **page** maps to a **Logical Block Address (LBA)** visible to the OS.

Deepening: Logical Block Address (LBA)

The **Logical Block Address (LBA)** is a key abstraction used by operating systems and file systems to **refer to storage locations** on a disk (HDD or SSD) in a simple, sequential manner.

LBA is an index number that uniquely identifies a fixed-size block of data (typically 512 bytes or 4 KB) on a storage device. It **hides the physical layout** of the drive (cylinders, heads, sectors) and presents a **flat address space** to the OS.

How does it work with SSDs? The OS issues read/write commands to specific LBA addresses. Internally, the SSD controller translates each LBA to a **physical location** in flash memory using a structured called the **Flash Translation Layer (FTL)**. This mapping is **dynamic** due to wear leveling, garbage collection, and bad block management.

In summary, the LBA is how the operating system sees the disk. Instead, the physical address is where the data actually resides inside the SSD.

Key operations are:

- `READ`: reads data from a **page**.
- `PROGRAM`: writes to a **page** (only if it's empty).
- `ERASE`: wipes an entire **block** (required before rewriting any page in that block).

The **important limitation** is that flash memory **cannot overwrite data in place**, we have to erase the whole block before reusing it. This lead to a **read-modify-erase-write cycle** even for simple updates.

Write Amplification Phenomenon

Write Amplification refers to the phenomenon where **the amount of actual data written to the NAND flash is greater than what the host system requested to write**.

Why does Write Amplification happen in SSDs? It stems from the **erase-before-write constraint** of NAND flash memory:

1. **(Flash) Pages can only be written once**, and to change them, we must erase the **entire block** (which may contain many pages).
2. If we modify even a **small piece of data**, the SSD:
   - Allocates a new page for the updated data.
   - Marks the old page as invalid (dirty).
   - Eventually, **copies all valid pages** from a block, **erases the block**, and **rewrites it** (this is called **garbage collection**).

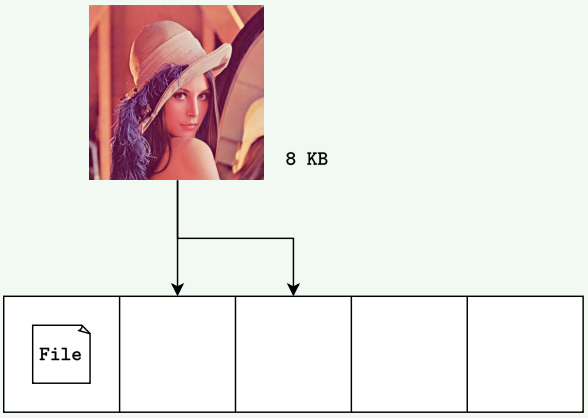
So a 4KB write might cause **hundreds of KBs** to be written internally!

Example 5: Write Amplification

Given a hypothetical SSD:

- Page Size: 4 KB
- Block Size: 5 Pages
- Drive Size: 1 Block
- Read Speed: 2 KB/s
- Write Speed: 1 KB/s

1. Write a 4 KB `.txt` file.
   - One page used (page size ÷ file dimension);
   - Time: 4 seconds (write speed × file dimension, 1 KB/s × 4 KB).
2. Write an 8 KB `.png`
   - Takes 2 pages;
   - There are 3 pages used in total;
   - Time: 8 seconds.



3. Delete the 4 KB `.txt` file
   - The first page is now **invalid** (dirty), but still **physically used**;
   - The SSD cannot reuse it until the whole block is erased.
4. Write a 12 KB image
   - OS sees "3 free pages", but **only 2 are truly empty**;
   - Can't fit 12 KB in 2 empty pages.

What happens internally?

(a) SSD reads the 8 KB of still-valid pages from NAND into a temporary cache.
This step takes 4 seconds (size to read ÷ read speed).

(b) Marks the 1st page (old `.txt`, step 1) as discarded.

(c) Places the new 12 KB into the cache.

(d) **Erases the entire 20 KB block** (this is mandatory before rewriting any page)

(e) **Rewrites**:
- 8 KB old data
- 12 KB new data

This step takes 20 seconds to write to the NAND.

The OS thought it was just a 12 KB write and expected only 12 seconds. But the SSD actually wrote 20 KB and read 8 KB (24 seconds total). This is a case of write amplification caused by limited empty pages, erase-before-write constraint, and the need to preserve valid data.

A direct mapping between Logical and Physical pages is not feasible inside the SSD. Therefore, each SSD has an FTL component that makes the SSD *look like an HDD*.

Flash Translation Layer (FTL)

The **Flash Translation Layer (FTL)** is the hidden "brain" of an SSD, it **makes NAND flash usable in the same way as a traditional hard disk**, despite its very different constraints.

Translates Logical Block Addresses (LBA) (see page 46) from the operating system into actual **physical locations** in the NAND flash. Also, it makes the SSD behave like an HDD to the OS (abstracts away erase-before-write and wear issues).

Why We Need Translation? Direct LBA to physical mapping isn't feasible because:

- ✗ Flash memory **can't overwrite in-place** (must erase first)
- ✗ Pages are grouped into blocks and must be programmed **in order**.
- ✗ Flash memory blocks **wear out over time**, requiring wear balancing.

The FTL responsibilities are:

1. **Address Mapping**. Maintains a **mapping table**, logical to physical page. Supports dynamic remapping when data is updated (e.g., a page becomes dirty and is relocated).

2. **Log-Structured Writes**. Uses log-structured techniques: **writes go to the next available page** in an erased block. It ensures writes are sequential (within a block), reducing write amplification and improving performance.

3. **Garbage Collection**. Identifies blocks full of **invalid/dirty pages**. Reads out remaining valid pages, erases the block, and rewrites valid data + new data elsewhere. Necessary to free up space for future writes.

4. **Wear Leveling**. Flash cells can only endure a limited number of erases. FTL spreads out writes across all blocks to **prevent early death** of heavily used regions.

Example 6: Log-Structured FTL

Setup:

- **Page size**: 4 KB
- **Block size**: 4 pages (total 16 KB per block)
- **Initial condition**: all pages are marked **INVALID**
- **Action**: perform a series of logical writes

Assume that a page size is 4 KB and a block consists of four pages. The write list is (`Write(pageNumber, value)`):

- `Write(100, a1)` → write value `a1` to logical page `100`
- `Write(101, a2)` → write value `a2` to logical page `101`
- `Write(2000, b1)` → write value `b1` to logical page `2000`
- `Write(2001, b2)` → write value `b2` to logical page `2001`
- `Write(100, c1)` → overwrite logical page `100` with value `c1`
- `Write(101, c2)` → overwrite logical page `101` with value `c2`

The steps are:

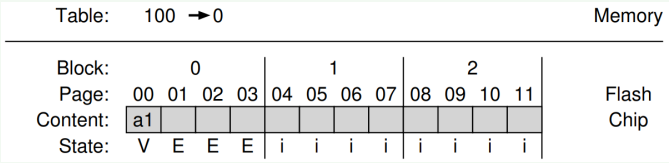
1. The initial state is with all pages marked as `INVALID(i)`:

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:												
State:	i	i	i	i	i	i	i	i	i	i	i	i

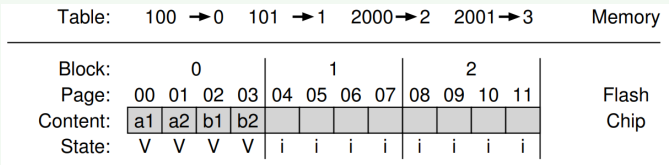
2. Erase block zero:

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:												
State:	E	E	E	E	i	i	i	i	i	i	i	i

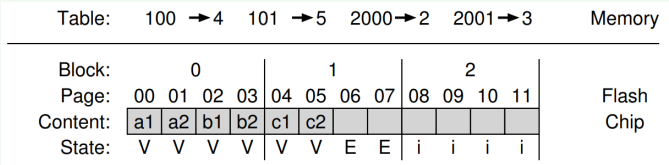
3. Program pages in order and update mapping information (first `Write(100, a1)`):



4. After performing four writes (`Write(100, a1)`, `Write(101, a2)`, `Write(2000, b1)`, `Write(2001, b2)`):



5. After updating 100 and 101:



Why Garbage Collection Exists

In SSDs, data cannot be updated in place, when a page is modified:

- The **old version becomes obsolete** (marked *invalid*).
- The **new version** is written to a **fresh page**.
- Over time, blocks fill with **invalid pages**, this is called *garbage*.

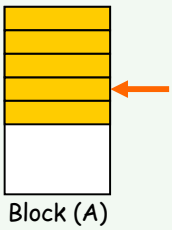
Garbage Collection (GC) is the process of **reclaiming space** by **erasing blocks that contain invalid pages**. It works like this:

1. **Identify a block** that has invalid (dirty) pages.
2. **Copy** any **still-valid pages** into a new block.
3. **Erase** the **original block** completely (erasing works only at block granularity).
4. **Update** the **mapping table** to reflect new page locations.

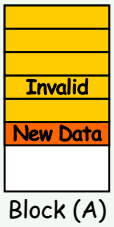
Example 7: how garbage collection works

The steps are:

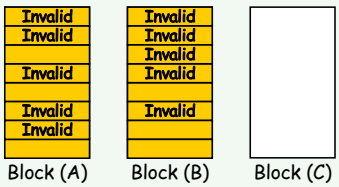
1. Update request for existing data:



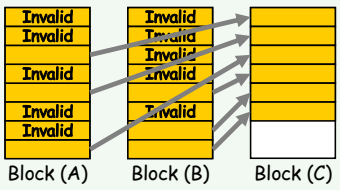
2. Find a free page, and save the new data:



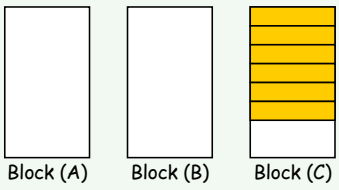
3. This scenario may continue until there are not enough free blocks:



4. Collect valid pages into a free block:



5. Update the map table and erase invalid (obsolete) blocks:



⚠ Three Major Problems of SSD Architecture

1. **Garbage Collection Is Expensive**. Garbage collection (GC) in SSDs is **unavoidable** due to the nature of NAND flash memory: flash **cannot overwrite pages**, only erase entire blocks.

   It is so expensive because it requires reading valid data (read cost), rewriting that data (write cost), and erasing entire blocks. If blocks are **partially valid**, even small updates can trigger **large internal data movement**. This leads to write amplification, effective write speed and wear on NAND cells.

   The **more valid data** in a block, the **higher the cost** of GC.

   - **Ideal case**: reclaim blocks with only invalid pages (no migration needed).
   - **Realistic case**: migrate live data, high overhead.

   ✅ Mitigation Techniques

   - **Overprovisioning**. Reserve extra flash capacity that the user can't see. More space, then delayed GC.
   - **Background GC**. Perform GC during **idle periods** to avoid disrupting foreground writes.
   - **Write buffering**. Use DRAM/SRAM to accumulate small writes and reduce fragmentation.
   - **Hot/Cold Separation**. Store frequently updated data (hot) separately from rarely updated data (cold).

2. **The Ambiguity of Delete**. In traditional file systems (especially on HDDs), **deleting a file** simply:

   - Removes the file's metadata (e.g., from the file system's directory tree).
   - Does **not erase or inform the disk that the data blocks are invalid**.

   This behavior is fine for HDDs, which can overwrite sectors anytime. But for SSDs, this **creates a serious mismatch**.

   SSDs rely on Garbage Collection (GC) to free space. GC assumes that **only invalid pages** can be discarded. However:

   - The SSD sees no distinction between "old" and "deleted" data unless explicitly told.
   - So even **deleted files look valid** to the SSD.
   - When GC runs, it **copies all pages** it believes to be valid, including junk!

   This causes SSDs to waste time and NAND endurance **preserving deleted data**.

   ✅ How to Fix it: TRIM / UNMAP. Modern OSs and SSD interfaces support special commands: TRIM (SATA) and UNMAP (SCSI/NVMe).

These commands allow the **OS to explicitly tell the SSD**: these Logical Block Addresses (LBAs) are no longer valid, feel free to erase them.

3. **Mapping Table Size and FTL Scalability**. In an SSD, the **Flash Translation Layer (FTL)** maps: **Logical Block Addresses (LBAs) from the OS to physical flash pages**. This mapping is essential because:

   - Flash memory can't overwrite in place
   - Pages must be written sequentially
   - Blocks must be erased before reuse

   So the SSD keeps an **internal mapping table** to know **where every logical page actually resides**.

   The mapping table **grows** proportionally with: **drive capacity**, and **granularity of mapping**.

   FTL Strategies to Cope

   - **Block-Level Mapping**. The FTL maps **each logical block number (LBN)** to a **physical block number (PBN)**. All pages within that block are **assumed to map 1:1**.

     Pros

     * **Very small mapping table**
     * Efficient in **sequential-write** workloads (e.g. logging, archiving).

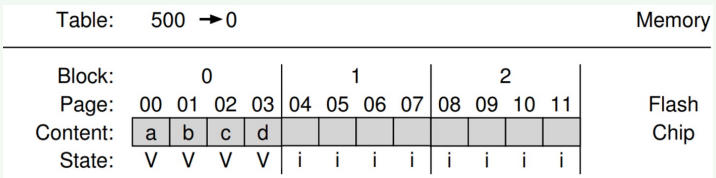
     Cons

     * **Terrible for random writes**: to update just 1 page, the entire block must be read, modified, erased, rewritten.
     * Results in very **high write amplification**.

Example 8: Block Mapping

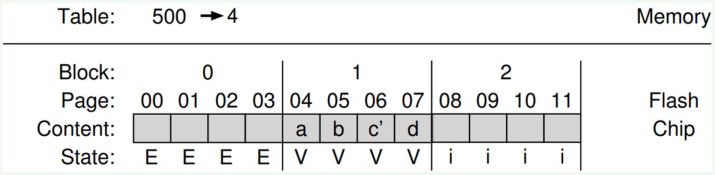
The first four writes:

- `Write(2000, a)`
- `Write(2001, b)`
- `Write(2002, c)`
- `Write(2003, d)`



And finally the last one:

- `Write(2002, c')`



- **Hybrid FTL**. Combine the best of Block-Level Mapping for most pages and use **Page-Level Mapping** (FTL maps each logical page number to a physical page number) for small updates. Often implemented using a **log block buffer**.

  ✔ **Pros**

  * Lower memory usage than pure page-level
  * Lower write amplification than pure block-level

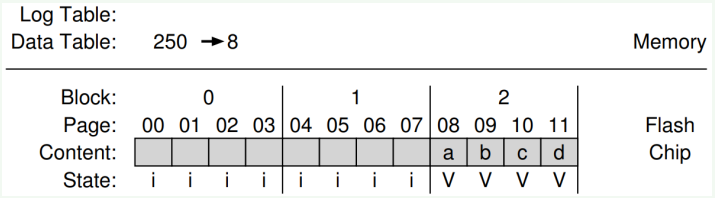
  ✖ **Cons**

  * More complex logic (copy-back handling, log merging)
  * Still suffers from some write amplification during **log cleaning**

Example 9: Hybrid Mapping

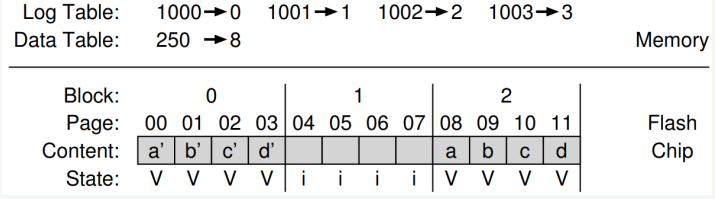
Let's suppose the following sequence:

- `Write(1000, a)`
- `Write(1001, b)`
- `Write(1002, c)`
- `Write(1003, d)`

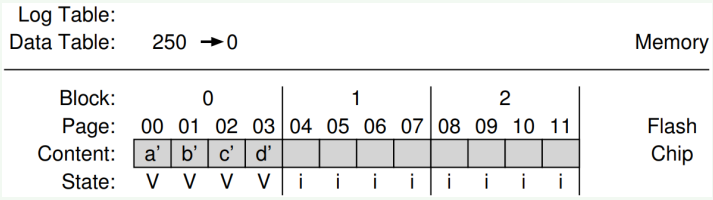


Let's update some pages:

- `Write(1000, a')`
- `Write(1001, b')`
- `Write(1002, c')`
- FTL updates only the page mapping information



When needed, FTL can perform MERGE operations:



- **Page mapping plus caching** (a.k.a. **Demand-based FTL (DFTL)**). Keep **only a portion** of the page-level mapping table in DRAM (a cache). **Store** the full mapping table in **flash itself**. On cache miss, **read the mapping** from flash into RAM (like a page table swap-in).

  ✔ **Pros**
  - **Scales to very large SSDs** with **low DRAM footprint**
  - Maintains page-level flexibility without storing entire table in RAM

  ✖ **Cons**
  - Mapping lookup incurs **read latency** on cache misses
  - More complex metadata management (must protect flash-stored tables)

✔ The importance of Wear Leveling

Flash memory has a **limited number of erase/write (EW) cycles**:

- Each block can only sustain ≈3,000 to 100,000 cycles (depending on type: SLC, MLC, TLC, QLC).
- If some blocks are used more than others (write **skew**), they **wear out faster**.

Without intervention, the SSD's lifespan would be determined by its **most heavily used block**. Ensure that **all blocks** in the SSD **wear out evenly** to: maximize **overall drive lifetime**, avoid **early failure** due to localized hot spots.

How Wear Leveling works? The **Flash Translation Layer (FTL)**:

1. Monitors **erase/write cycles** per block.
2. Identifies **cold blocks** (rarely updated, long-lived data).
3. Periodically:
   - **Reads** valid data from cold blocks
   - **Moves** it to fresher blocks
   - **Erases** and **reuses** the original blocks

   Even clod blocks are "rotated" to ensure wear balance.

There are three types of Wear Leveling:

- **Dynamic**: Spread writes among blocks not currently holding static data
- **Static**: Occasionally move long-lived (cold) data to give its block a rest
- **Hybrid**: Combine both (used in most SSDs)

Wear Leveling Disadvantages

- **Increases Write Amplification**: moving cold data incurs extra writes.
- **Consumes bandwidth**: periodic copying reduces performance.
- **Complexity**: requires tracking per-block wear counts and scheduling background operations.

However, to **partially fix** this, a simple policy to apply is that each flash block has an **Erase/Write Cycle Counter** and maintains the value of:

$$|\text{Max (EW cycle)} - \text{Min (EW cycle)}| < \varepsilon \tag{3}$$

Where ε is a system-defined Wear Leveling threshold.

HDD vs SSD

UBER and TBW are two metrics help quantify how **reliable and durable a storage device is over time and under heavy use**.

- **Unrecoverable Bit Error Ratio (UBER)**. The probability that a bit cannot be recovered correctly by the device, even after error correction.

$$\text{UBER} = \frac{\text{Number of unrecoverable bit errors}}{\text{Total bits read}} \tag{4}$$

  A **lower** UBER means **better data reliability**.

- **Endurance rating: Terabytes Written (TBW)**. The total amount of data we can write to the SSD over its warrantied lifetime before cells are expected to wear out.

$$\text{TBW} = \text{Endurance rating of the SSD (from manufacturer)} \tag{5}$$

  For example, a 250 GB SSD with TBW = 70 TB, we can write: $70 \div 365 = 190$ GB/day.

2.2.2.4 RAID

How is RAID born?

Before 1980, **JBOD (Just a Bunch Of Disks)** was a common setup before the advent of disk arrays and RAID. JBOD simply **connects multiple physical disks to a system**, but each disk operates **independently**. There's **no striping** and **no redundancy**. Each disk has its **own mount point**, and the operating system or user decides where to store data. If one disk fails, **only the data on that disk is lost**, no protection or recovery is provided. **Performance bottlenecks** also occurred as CPUs became faster than disk I/O.

Researchers at UC Berkeley (notably Patterson, Gibson and Katz) proposed using **multiple cheaper disks** together: treat **many small, inexpensive disks** as **one large logical disk**. These were called **Disk Arrays**. The main idea was to achieve higher performance and more memory through parallelism.

- Disk Arrays appear as a **single logical high-performance disk**.
- Use **data striping** and **parallel access**, but these **weren't formalized or optimized** in the early implementation. This is because the core idea was to spread data across multiple disks to improve throughput (multiple disks serving different I/Os) and latency (multiple I/Os in parallel reduce queue wait time). But the **parallelism was often implicit**, not exposed to the software or the user. Some arrays allowed **concurrent I/O by accident**, not design. **I/O scheduling** was hardware-specific, **not standardized**.

Key improvements:

- ✔ **Parallel Access**: Multiple disks could serve I/O requests simultaneously.
- ✔ **Performance Boost**: Higher bandwidth and lower latency via concurrent disk activity.
- ✔ **Scalability**: Easy to increase capacity by adding disks.
- ✔ **Lower Cost per GB**: Used **multiple small, cheap disks** instead of one large expensive disk.
- ✔ **Modular Architecture**: Allowed easier maintenance, replacement, and upgrades.

Limitations:

- ✖ **No Standard Striping**: Striping (if present) was non-standard and not guaranteed.
- ✖ **No Redundancy**: If a disk failed, **data was lost**, arrays were vulnerable.
- ✖ **No RAID Levels**: No formal trade-offs between performance/reliability (e.g. RAID 0-6).
- ✖ **No Fault Tolerance Mechanism**: No automatic reconstruction, hot spares, or parity.

✖ **Limited Write Optimization**: RAID improved performance (esp. for writes) using smart parity layouts.

In 1988, the same researchers published the seminal paper: "A Case for Redundant Arrays of Inexpensive Disks (RAID)" [7]. They made two key contributions:

- They **formally introduced RAID**: not just disk arrays, but structured ones that also **handle reliability** via redundancy.
- They **defined RAID levels** (RAID 0 to RAID 5 in the original paper), each with different trade-offs between: *performance*, *reliability* and *storage efficiency*.

RAID was important because it **solved the reliability problem introduced by disk arrays**. Disk Arrays solved performance and capacity, but RAID solved reliability too, combining **Data Striping** (performance) and **Redundancy** (reliability).

Definition 2: RAID

RAID (Redundant Array of Independent Disks) is a **storage technology** that **combines multiple physical disks into a single logical unit** to improve performance, reliability, or both, by using data striping, redundancy, or both.

RAID is based on two core techniques: **Data Striping** (for performance, splits data across disks), and **Redundancy** (for reliability, adds fault tolerance via mirroring or parity).

Exists different RAID levels. Each level defines a strategy for:

- *How data is striped*
- *How redundancy is added*
- *What failure modes are tolerated*
- *How performance and capacity are affected*

Data Striping

Data Striping is the technique of **splitting data into small chunks** (called **stripe units**) and distributing them **across multiple disks in a round-robin manner**. Used primarily to **improve performance** by enabling parallel disk access. It does **not add redundancy** by itself.

- **Stripe Unit**: The amount of **data written to a single disk** before moving to the next one. Can be in **bits**, **bytes**, **blocks** or **KBs**. Small stripe unit, better parallelism for large files. Large stripe unit, fewer disk operations for small reads.
- **Stripe Width**: The **number of disks** over which the data is stripped. If we have 4 disks participating in stripping, stripe width is 4. **Determines how many disks work in parallel** for a given I/O.
- How are **Multiple Independent I/O Requests** processed? **Small**, random I/O **requests** from **different applications or users**. RAID can **process them in parallel** on different disks. As result: **lower disk queue lengths and faster response times**.
- How is a **Single Multiple-Block I/O Request** processed? A **large read or write operation** on a big file (e.g., a video or a database query). Striping lets **multiple disks work together** to serve this one big request. As result, **much faster data transfer** compared to using one disk.

Redundancy

Redundancy is the method of **adding extra information to protect data from disk failures**. If a disk fails, the system can reconstruct the missing data using the redundant information. Redundancy is *essential* because:

- Disk arrays use many disk, and **more disk**, **higher risk of failure**.
- RAID introduces redundancy to **tolerate and recover from failures**.

There are two types of redundancy:

1. **Data Duplication (Mirroring)**. Every block of data is **copied exactly** to another disk. Used in RAID 1 and RAID 10.
   - ✔ Can survive **complete disk failures**.
   - ✔ Fast reads (can load-balance between mirrors).
   - ✖ **Wastes 50%** of storage (one disk stores only a copy).
2. **Parity-Based Redundancy**. Adds a **calculated parity block** (usually XOR) that allows **rebuilding lost data**. Used in RAID 5 and RAID 6.
   - ✔ More **storage-efficient** than mirroring.
   - ✔ Can tolerate 1 (RAID 5) or 2 (RAID 6) disk failures.
   - ✖ **Writes are slower** (need to read/update parity on each write).

Why is it called "Redundant Array"? Because RAID **stores more than just our data**, it stores the **redundant data necessary to protect it**.

RAID levels

A **RAID level** defines a specific way to organize **striping** and **redundancy** across multiple disks. Each level balances **three key trade-offs**:

- **Performance**: *How fast are reads/writes?*
- **Fault Tolerance**: *Can it survive disk failure?*
- **Storage Efficiency**: *How much usable space is left?*

RAID	Striping	Redundancy	Tolerates Failure?	Main Benefit
0	✔	✖ None	✖ No	Max speed, no protection
1	✖	✔ Mirroring	✔ 1 disk	Reliability via copies
5	✔	✔ Parity (rotated)	✔ 1 disk	Balance of perf + space
6	✔	✔ Dual Parity	✔ 2 disks	Extra protection

Table 6: Standard RAID Levels: RAID 4 is not shown because it is rarely used; RAID 5 has the same concept but eliminates RAID 4's limitations. RAID 2 and 3 are also not shown because they are not covered in this course.

Each RAID level is a **design pattern** that answer: "*How can I spread and protect data across multiple disks to meet my performance, capacity, and reliability goals?*".

Table 7: RAID - Table of Contents.

RAID 0 - Striping Without Redundancy

RAID 0 splits (stripes) data across multiple disks to increase performance. It **does not provide redundancy**: if one disk fails, **all data is lost**.

Structure

- **Data Striping**: ✔ Yes (block-level)
- **Redundancy**: ✖ None
- **Minimum disks**: 2
- **Fault Tolerance**: ✖ 0 disks, a single failure is fatal
- **Storage Efficiency**: ✔ 100% (all capacity is usable)

When is it used?

High performance environments where **data loss is acceptable**, such as temporary data storage, video editing, gaming, scratch disks or caches.

Performance

Operation	Result
Reads	✔ Much faster (parallel access)
Writes	✔ Much faster (split writes)
Failure	✖ One disk dies, **all lost**

✔ Advantages

- **Lower cost** because it does not employ redundancy (no error-correcting codes are computed and stored).
- **Best read/write performance** (it does not need to update redundant data and is parallelized).

Disadvantages

- **Zero fault tolerance**.
- High risk: the failure rate is equal to the sum of all disk failure probabilities.

How does it work?

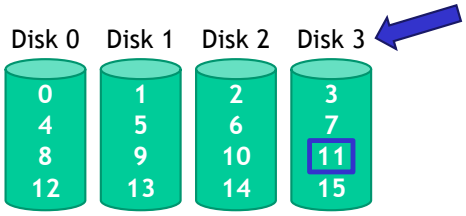
The main idea of RAID 0 is to use several physical disks as **a single logical disk** by **striping** data across them (splitting data into small blocks and writing them in **round-robin fashion** across all N disks). This approach improves performance:

- **Sequential access** is distributed across all disks, increased transfer rate.
- **Random access** naturally spreads I/O load across disks, lower latency and better throughput.

The system uses the following formulas to **determine which disk and offset holds a given logical block**:

$$\begin{aligned} \texttt{Disk index} &= \texttt{logical\_block\_number} \,\%\, \texttt{number\_of\_disks} \\ \texttt{Block Offset} &= \frac{\texttt{logical\_block\_number}}{\texttt{number\_of\_disks}} \end{aligned} \tag{6}$$

For example, the command `read block 11` in a 4-disk array gives disk 3 as the index and block 2 as the offset on that disk.



This formula **allows the RAID controller to locate any block without metadata**, just with arithmetic.

The choice of chunk size is critical because it affects performance at this level. The **Chunk Size** (also called **Stripe Unit** size) is the amount of **data written to each disk before moving to the next one**.

The impact of chunk size is evident:

- **Small** chunks → better **parallelism** (each access touches more disks).
- **Large** chunks → lower **seek overhead** (each file uses fewer disks).

In practice, typical RAID arrays use 64 KB chunks as a **balanced** (tradeoff) default.

Measuring Performance

- **Measuring *Sequential* Transfer Rate**. For **large sequential I/O**, the transfer time includes:

  1. **Seek time** (page 130)
  2. **Rotational latency**
  3. **Actual Data Transfer Time**:

$$S = \frac{\texttt{transfer\_size}}{\texttt{time\_to\_access}} = \frac{\texttt{data\_size}}{\texttt{disk\_rate}} \tag{7}$$

  For example, suppose there is 7 ms of *seek time*, 3 ms of *rotational latency*, and 10 MB of *data to transfer* at a *disk rate* of 50 MB/s. The total time is given by the seek time, rotational latency, and the time taken to transfer the data:

$$\text{Time} = 7\,\text{ms} + 3\,\text{ms} + (10\,\text{MB} \div 50\,\text{MB/s}) = 210\,\text{ms}$$

  Thus, the sequential throughput is:

$$S = 10\,\text{MB} \div 0.21\,\text{s} = 47.62\,\text{MB/s}$$

  In RAID 0 with N disks, the **Total Sequential Throughput** is:

$$\text{Total Sequential Throughput} = N \times S \tag{8}$$

  S is the sequential throughput for a single disk in the system.

- **Measuring *Random* Transfer Rate**. For **small random accesses**, latency dominates over transfer. Using the same example as before, but changing the file size to 10 KB, the time is as follows:

$$\text{Time} = 7\,\text{ms} + 3\,\text{ms} + (10\,\text{KB} \div 50\,\text{MB/s}) = 0.98\,\text{MB/s}$$

  Resulting random throughput:

$$R = 10\,\text{KB} \div 10.2\,\text{ms} = 0.98\,\text{MB/s}$$

  In RAID 0 with N disks, the **Total Random Throughput** is:

$$\text{Total Random Throughput} = N \times R \tag{9}$$

  R is the random throughput for a single disk in the system.

Feature	Description
Capacity	N, all disk space is usable (no redundancy)
Reliability	0, MTTDL (Mean Time To Data Loss) = MTTF (Mean Time To Failure)
Performance	It provides full parallelization for both random and sequential throughput

RAID 1 - Mirroring

RAID 0 offers performance, but **no protection** against failures. **RAID 1 solves this** by maintaining **two identical copies of all data** (mirroring).

- ✔ **High reliability**: if one disk fails, the second copy is used seamlessly.
- ✔ **Fast reads**: can read from either disk (or balance across them).
- ⚠ The **write speed** is slightly slower than that of a single disk because the **data must be written to both disks**.
- ✖ **High cost**: only **50% of total disk space** is usable.

In theory, RAID 1 can use **more than 2 disks** to mirror data, but rarely done due to **very high costs** (only $1 \div N$ capacity used), and too much overhead for typical storage needs. In other words, more than 2-way mirroring is possible but **not practical**.

Why combined RAID levels at all?

RAID 1 is great, but it has its **limits**. It mirrors data, meaning one disk has the data and the other disk has a copy of it. Therefore, we always need pairs of disks. With two disks, RAID 1 is simple and reliable. But what if we want to use more than two disks and want both high reliability and high performance? We want to **combine the reliability that RAID 1 provides with the performance and scalability that RAID 0 provides**. In other words, how do we structure the disks as their number increases to get the best of both RAID 0 and RAID 1? The answer lies in the RAID 10 and RAID 01 techniques.

The goal of combining RAID levels is to achieve both reliability and performance. The combination is **generally written as** RAID $x + y$ (sometimes called RAID xy), meaning we **first** apply RAID x to small groups of disks and **then** apply RAID y on top of those groups.

We divide $n \times m$ total disks into m groups, each containing n disks.

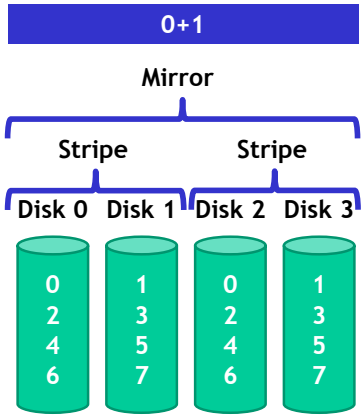
1. Apply RAID x to each group.
2. Apply RAID y to the m groups, treating each group as a "logical disk".

There are two possible combinations at this level:

- **RAID 0 + 1** is a **striped array of mirrored sets**. It combines the performance of RAID 0 and the fault tolerance of RAID 1, but with **some limitations**.

  The structure consists of a **minimum of 4 disks**:

  1. **Start with striping** (RAID 0): split data across 2 or more disks for performance.
  2. **Then mirror the whole striped group** (RAID 1).



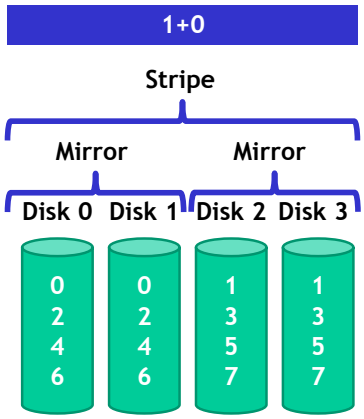
⚠ **RAID 0 + 1: Failure Behavior**. If **any one disk fails**, the system still works using the mirrored group. But if **a disk fails in each stripe**, **all data is lost**. After the **first failure**, the array **degrades into pure RAID 0**, losing redundancy.

Aspect	Value
Usable space	50% (half is mirror)
Read speed	✔ High (parallel reads)
Write speed	⚠ Slower (must write both copies)
Reliability	✔ Tolerates 1 disk failure (in same group)

- **RAID 1 + 0** combines **mirroring (RAID 1)** and **striping (RAID 0)**. First, data is **mirrored**, then it is **striped** across mirrored pairs. It offers **high performance (like RAID 0)**, **high fault tolerance (like RAID 1)**.

  The structure consists of a **minimum of 4 disks**:

  1. **Create mirrored pairs** (RAID 1)
  2. **Stripe data across pairs** (RAID 0)



RAID 1 + 0: Failure Tolerance

- ✔ Can survive **multiple disk failures**, as long as **only one per mirrored pair**.
- ✖ If both disks in a mirrored pair fail, data loss.

RAID 10 is **more fault-tolerance than RAID 01**, especially in cases of multiple failures. Because each mirrored pair in RAID 10 is self-contained:

- If one disk in a pair fails, only **that pair** needs to be rebuilt (not the entire array).
- The rest of the array **remains fully operational** and can keep serving requests.

In RAID 01:

- A single disk failure disables **half of the array**.
- A second failure in the other half = total data loss.

Aspect	Value
Usable space	50% of total (because of mirroring)
Read speed	✔ Very fast (parallel reads from mirrors)
Write speed	✔ Faster than RAID 5/6 (no parity calc.)
Reliability	✔ High (survives multiple failures safely)

Widely used in databases, transactional systems, and virtualized environments where both performance and fault tolerance are critical.

In conclusion, RAID 10 is created by **first mirroring disks** and **then striping** across the mirrored pairs. This configuration provides **faster read and write** speeds and **greater fault tolerance** than RAID 01, making it ideal for high-load systems.

Measuring Performance

- **Capacity**. Total usable space is $N \div 2$, since each block is stored twice, only **50% of disk space** is usable.
- **Reliability**. Can **survive 1 disk failure** guaranteed. In bets-case scenarios, can survive **up to** $N \div 2$ **disk failures**, if no two are mirrors of each other. However, if **both disks in a mirrored pair fail**, data is lost.
- ***Sequential* Performance**

  Sequential Write. Writes must go to both copies, then halved throughput:

$$\text{Sequential Write} = \left(\frac{N}{2}\right) \times S \tag{10}$$

Sequential Read. In theory, reads could be optimized, but usually reads are directed to **only one disk per mirror**. So, **half of the disks are idle** during reads:

$$\text{Sequential Read} = \left(\frac{N}{2}\right) \times S \tag{11}$$

- ***Random* Access Performance**

  Random Read. **Best case** for RAID 1. Reads can be **load-balanced** across all N disks. System can choose which mirror is less busy, then fully parallel.

  $$\text{Random Read} = N \times R \tag{12}$$

  Random Write. Writes must go to both disks in each pair. Only $N \div 2$ mirrors, then only $N \div 2$ writes happen in parallel.

  $$\text{Random Write} = \left(\frac{N}{2}\right) \times R \tag{13}$$

RAID 1 provides **excellent reliability** and **great random read performance**. Its main limitations are only 50% of storage is usable, and sequential throughput is limited to half the potential.

The Consistent Update Problem

In RAID 1, every write must be done **twice** (once per mirror). This raises a **consistency issue**: *what if the system crashes or loses power after writing to one disk, but before writing to the other?*

- One mirror has the **updated data**
- The other mirror has **state (old) data**
- The two copies are now **out of sync**

This violates the **atomicity** principle (a write must be **all or nothing**, either all mirrors are updated or none).

How it's handled. Many RAID controllers implement a **Write-Ahead Log (WAL)**:

- A special **non-volatile memory area** (battery-backed cache).
- Writes are first saved into the log **before** writing to disks.
- If power fails, the controller can use the log to:
  - Complete the write
  - Or roll it back (to restore consistency)

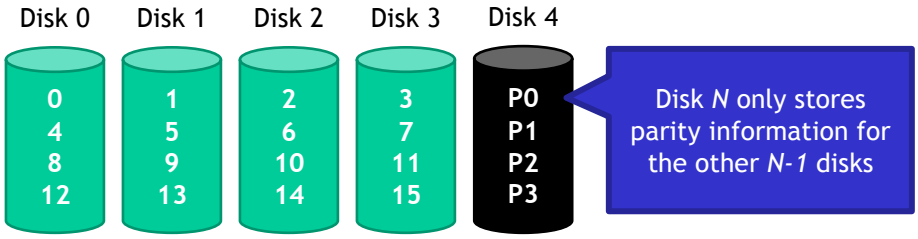
This ensures that mirrored copies stay **synchronized**, even after failure.

RAID 4 - Parity Drive

RAID 4 uses **block-level striping** (like RAID 0), but adds a **dedicated parity disk** to provide **fault tolerance**.

Parity Calculation

Parity is computed using the **XOR (exclusive OR)** operation. For example, 4 blocks per stripe:



Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	0	1	1	0 ^ 0 ^ 1 ^ 1 = 0
0	1	0	0	0 ^ 1 ^ 0 ^ 0 = 1
1	1	1	1	1 ^ 1 ^ 1 ^ 1 = 0
0	1	1	1	0 ^ 1 ^ 1 ^ 1 = 1

Parity calculated using XOR

Figure 8: RAID 4 - *How does it work?*

It is used XOR because it has a special property: if we know all but one value, we can **reconstruct the missing one**. For example, if Disk 2 is lost, it can be reconstructed using an XOR operation with Disks 0, 1, and 3 and the Parity unit.

Pros and Cons

- Fault tolerance: can survive **1 disk failure**.
- More **space-efficient** than RAID 1 (only 1 disk used for redundancy).
- **Parity disk is a bottleneck**. Every write must update parity, then a single disk becomes overloaded.
- Write performance is poor compared to RAID 0 or RAID 10.

How does it work?

- **Updating Parity on Write in RAID 4**. When we **modify a block** in RAID 4, we must also update the **parity block** in that stripe. There are two ways to do it:

  1. **Additive Parity**. Recalculate parity using all the blocks in the stripe. Steps:
     (a) **Read all other data blocks** in the stripe.
     (b) XOR them together to get the **new parity**.

     Simple and correct, but **expensive** because must **read all blocks** to recompute.
  2. **Subtractive Parity** (Efficient: modify incrementally). Update the parity by using a **mathematical trick**:

     $$\text{New Parity} = \text{Old Parity} \oplus \text{Old Data} \oplus \text{New Data} \tag{14}$$

     This works thanks to the **reversible** XOR property:

     $$A \oplus A = 0, \quad A \oplus 0 = A$$

     Very efficient because only need to read the block being written (**old value**) and the **old parity block**.

- **Reads in RAID 4**. Reads in RAID 4 are **fast and efficient**, especially compared to writes. Data is **striped across all data disks**, and the **parity disk is not involved** in normal reads. There are two possible read scenarios:

  1. **Sequential Read**. Large, continuous read (e.g., loading big file). Data is read in **parallel** from all data disks. It is **fast**, each disk reads part of the file simultaneously.
  2. **Random Read**. Small, scattered reads (e.g., database lookups). Still efficient because:
     – Each disk can handle **its own read** in parallel.
     – Parity disk is **not touched** during a normal read.

  As we saw at the beginning, if a data **disk fails**, the missing block is **reconstructed using XOR**:

  $$D_{\text{missing}} = P \oplus D_1 \oplus D_2 \oplus D_3 \oplus \dots$$

  Read recovery is still possible, it is just slower.

- **Serial Writes in RAID 4**. Serial writes mean writing **a sequence of blocks** that all belong to the **same stripe**, typically during large, contiguous writes (e.g., streaming, backups).

  In RAID 4, data is **striped** across all data disks. The **parity for each stripe** is stored in a **dedicated parity disk**.

  During serial writes, each data disk receives **one block** from the stripe. The **parity disk** is updated **once per stripe**. Since all writes hit the same stripe, the **parity disk becomes a bottleneck**.

Operation	Description
Data writes	✔ Parallel across data disks
Parity update	✘ Centralized
Bottleneck	Parity disk handles **every stripe's write**
Comparison	Same structure as read, then but **parity slows it**

Even though data writes are parallel, all serial writes in the same stripe **converge on the parity disk**, making it the **main point of contention**.

- **Random Writes in RAID 4**. A **random write** updates **one block** somewhere on disk, not the entire stripe. This is common in databases, file systems, and OS workloads.

  The writing process for each block is as follows:

  1. **Read** the **old data block** and the **old parity block**.
  2. **Compute** the parity block:

     $$P_{\text{new}} = D_{\text{old}} \oplus D_{\text{new}} \oplus P_{\text{old}}$$

     This efficiently updates parity using subtraction (XOR).
  3. **Write** the **new data block** and the **updated parity block**

⚠ The main problem here is the **bottleneck** caused by the parity disk. Even though the data block may change on **any disk**, the **parity block is always on the same disk**. Each write operation must **read from** and **write to** that disk. This creates **serialization**, only one parity update can occur at a time. Even with multiple writes to different data disks, they all **wait for the parity disk**.

Operation	Behavior
Data writes	✔ Fast (per disk)
Parity update	✘ Always goes to the **same disk**
Bottleneck	⚠ Parity disk handles **every write**
Overall	✖ **Terrible performance** on random writes

Measuring Performance

- **Capacity**: $N - 1$. One disk is used exclusively for parity, so it is **not usable for data**.
- **Reliability**: can tolerate **1 disk failure** (any one of the data or parity disks). But if a disk fails, the system can still serve data using parity reconstruction. However, write and read performance becomes **massively degraded** (especially if parity disk is affected).
- **Read/Write Performance**

Operation	Performance	Notes
Sequential Read	$(N-1) \times S$	✔ Full parallelism across all data disks
Sequential Write	$(N-1) \times S$	✔ Parallel as long as writes hit different stripes
Random Read	$(N-1) \times R$	✔ Reads avoid parity disk, then high parallel performance
Random Write	$R \div 2$	✖ Terrible, every write must access parity disk twice

> Parity disk becomes a **point of contention for every write**. Causes **serialization of random writes**, reducing overall throughput drastically.

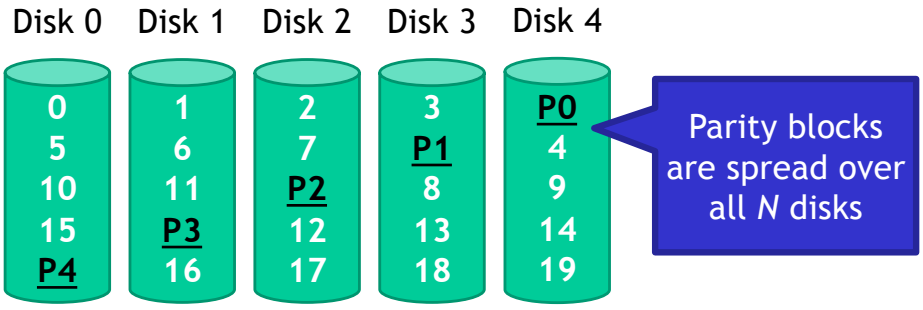
In conclusion, RAID 4 provides ample capacity and quick sequential input/output (I/O) thanks to block-level striping. It uses one dedicated parity disk for fault recovery. However, its **main weakness is its poor random write performance**, as all writes target the same parity disk. This is why RAID 4 is rarely used in practice and **led to the development of RAID 5, which solves this issue by distributing parity**.

RAID 5 - Rotating Parity

RAID 5 extends RAID 4 by using **distributed (rotating) parity**. This removes the **bottleneck of the single parity disk**.

How does it work?

In RAID 4, parity is always stored on a **dedicated parity disk**, causes a bottleneck. In RAID 5, parity is **spread (rotated)** across **all disks**, **one stripe at a time**.



Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	0	1	1	0 ^ 0 ^ 1 ^ 1 = 0
1	0	0	0 ^ 1 ^ 0 ^ 0 = 1	0
1	1	1 ^ 1 ^ 1 ^ 1 = 0	1	1
1	0 ^ 1 ^ 1 ^ 1 = 1	0	1	1

Parity position rotates from one disk to the next. So **each disk holds both data and some parity**; there's no dedicated parity disk. However, parity is still computed using the XOR operation, and the operand depends on the disks that calculate it. For example, Disk 4 will calculate the XOR using Disks 0, 1, 2, and 3. Disk 3 will use disks 0, 1, 2, and 4, and so on for the others.

What happens during a Random Write?

Like in RAID 4, RAID 5 must **preserve parity** whenever data changes. A **random write** modifies only one block in a stripe, so we use an efficient **subtractive parity update**.

1. **Read**
   (a) Read the *old data block*
   (b) Read the *old parity block*
2. **Compute**. Use XOR subtraction:

$$P_{\text{new}} = D_{\text{old}} \oplus D_{\text{new}} \oplus P_{\text{old}}$$

3. **Write**
   (a) Write the *new data block*
   (b) Write the *updated parity block*

In total, there are **4 disk operations**: 2 reads (old data and old parity) and 2 writes (new data and new parity).

RAID 5 improvement over RAID 4

In **RAID 4**, all parity writes go the **same disk**, and this creates a **bottleneck**. **RAID 5 solves** this slowdown by **rotating the parity**. Parity updates are spread across all disks, which distributes the load evenly. This allows **random writes to occur in parallel**, eliminating serialization due to a single disk. This is obviously much better than RAID 4.

Analysis

- **Capacity**: $N - 1$ disks. Same as RAID 4, one disk's worth of capacity is used for **distributed parity**.
- **Reliability**: tolerates 1 **disk failure**. However, during partial outage the degraded mode phenomena is highlighted, because all missing data must be reconstructed on-the-fly using parity, which causes massive performance degradation.
- **Read/Write Performance**

Operation	Performance	Notes
Sequential Read	$(N-1) \times S$	= Same as RAID 4, striping across data disks
Sequential Write	$(N-1) \times S$	✔ Fully parallel if stripes span across disks
Random Read	$N \times R$	Better than RAID 4, all disks can be read in parallel
Random Write	$\frac{N}{4} \times R$	Better than RAID 4, writes are distributed. However, each write requires an additional 2 reads and 2 writes for parity updates.

For each random write, we have 2 reads and 2 writes with N disks and parallelism. The **throughput** is:

$$\text{Throughput} \approx \frac{N}{4} \times R \tag{15}$$

In conclusion, RAID 5 provides a good **trade-off between capacity**, **reliability**, and **performance**. It avoids RAID 4's parity bottleneck by using **rotating parity**, enabling better parallelism for **random writes**. However, in case of a disk failure, **performance degrades** significantly due to **on-the-fly parity reconstruction**.

RAID 6 - Dual Parity

RAID 6 is an extension of RAID 5 that adds **a second independent parity block** per stripe. This allows it to **tolerate two simultaneous disk failures**, a major reliability upgrade over RAID 5.

How does it work?

It can tolerate **up to 2 concurrent disk failures**. Uses **two different types of parity**:

- **P parity**: is the same as in RAID 5, a "*simple*" XOR (exclusive-or) of the data blocks in a stripe.
- **Q parity**: is more complex. It is generated using **Reed-Solomon** coding, a mathematical technique over **Galois Fields (GF)**, which enables the correction of multiple simultaneous failures.

Each parity serves a different purpose:

- **P (XOR parity)** handles **one disk failure**.
- **Q (Reed-Solomon parity)** can handle a second failure that isn't linearly dependent on the first one.

This combination allows RAID 6 to recover from **two simultaneous disk failures**. In simple terms:

- If we lose 1 disk, we can **recover using XOR** (as in RAID 5).
- If we lose 2 disks, the system **solves a set of linear equations** (using P and Q) **to recover the missing data**.

This requires the two parities to be **independent**, so their information isn't redundant.

The **disk configuration requires** $N + 2$ **disks**. N are data disks, and 2 are distributed parity blocks (P and Q) per stripe. The **minimum configuration** requires at least **4 data disks**, for a total of **6 disks minimum**.

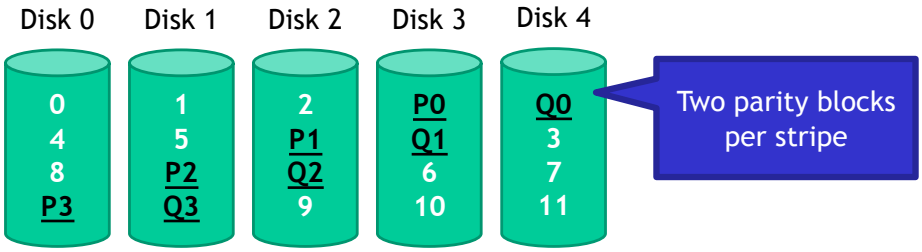
Write Performance

Write performance is **slower than with RAID 5**. Each write requires updating both parity blocks, necessitating at least **6 disk accesses**.

1. Read *old data*, read *parity block P*, and read *parity block Q*
2. Write *new data*, write *new parity block P*, and write *new parity block Q*

High overhead makes RAID 6 **slower on random writes**, but **safer**.

In summary, **RAID 6** adds a **second**, **independent parity block** ($P + Q$) per stripe, enabling **recovery from two simultaneous disk failures**. RAID 6 uses Reed-Solomon coding for the Q parity and distributes both parity blocks across all disks. Although **RAID 6 is more fault-tolerant than RAID 5**, it suffers from higher write overhead, requiring at least 6 I/O operations per write.



Comparison and characteristics of RAID levels

The following table shows eight fundamental properties of the RAID levels.

- N: number of drives
- R: random access speed
- S: sequential access speed
- D: latency to access a single disk

	RAID 0	RAID 1	RAID 4	RAID 5
Capacity	N	$N \div 2$	$N - 1$	$N - 1$
Reliability	0	$1 \vee : N \div 2$	1	1
Sequential Read	$N \times S$	$(N \div 2) \times S$	$(N - 1) \times S$	$(N - 1) \times S$
Sequential Write	$N \times S$	$(N \div 2) \times S$	$(N - 1) \times S$	$(N - 1) \times S$
Random Read	$N \times R$	$N \times R$	$(N - 1) \times R$	$N \times R$
Random Write	$N \times R$	$(N \div 2) \times R$	$R \div 2$	$(N \div 4) \times R$
Read	D	D	D	D
Write	D	D	$2 \times D$	$2 \times D$

Table 8: Comparison of RAID levels.

Where the throughput is:

- Sequential Read
- Sequential Write
- Random Read
- Random Write

And the latency is:

- Read
- Write

RAID	Capacity	Reliability	Read Perf.	Write Perf.	Rebuild Performance
0	100%	N/A	✔ Very good	✔ Very good	✔ Good
1	50%	✔ Excellent	✔ Very good	✔ Good	✔ Good
5	$\frac{(n-1)}{n}$	✔ Good	✔ Good	✘ Fair	✘ Poor
6	$\frac{(n-2)}{n}$	✔ Excellent	✔ Very good	✘ Poor	✘ Poor
$1+0$	50%	✔ Excellent	✔ Very good	✔ Good	✔ Good

Table 9: Characteristics of RAID levels.

- Best **Performance and Capacity**: `RAID 0`.
- **Maximum Fault Tolerance**: `RAID 1` or `RAID 6`, with RAID $1+0$ **often preferred** over $0+1$.
- **Balanced Solution** (Space, Performance, Recoverability): `RAID 5`.

We report the RAID level comparison here. This is for utility reasons. It is on page 60.

RAID	Striping	Redundancy	Tolerates Failure?	Main Benefit
0	✔	✘ None	✘ No	Max speed, no protection
1	✘	✔ Mirroring	✔ 1 disk	Reliability via copies
5	✔	✔ Parity (rotated)	✔ 1 disk	Balance of perf + space
6	✔	✔ Dual Parity	✔ 2 disks	Extra protection

Final Considerations and Adopted Techniques:

- **Hot Spare Disk**. A *hot spare* is a **physical disk** already installed and powered in the system but **not actively used** to store data or parity. It's **just waiting**.

  This is important because when **a disk** in a RAID array **fails**, **rebuilding** the array, i.e., reconstructing lost data using parity, can **take hours or days**, depending on the disk size and load. However, with a hot spare:

  - ✔ The controller **immediately** starts rebuilding the lost data onto the spare.
  - ✔ No need for human intervention to replace the disk.
  - ✔ It reduces downtime and risk of further data loss during the rebuild period.

  The mechanics of a Formula 1 team are a **great analogy**. The pit crew remains inactive for most of the race. However, the moment the car enters the pit lane, the crew acts immediately and precisely, replacing tires or making adjustments in a matter of seconds. Similarly, the hot spare disk is not part of the active array during normal operations. However, as soon as a failure occurs, the hot spare disk begins the rebuild process, helping to quickly restore full redundancy.

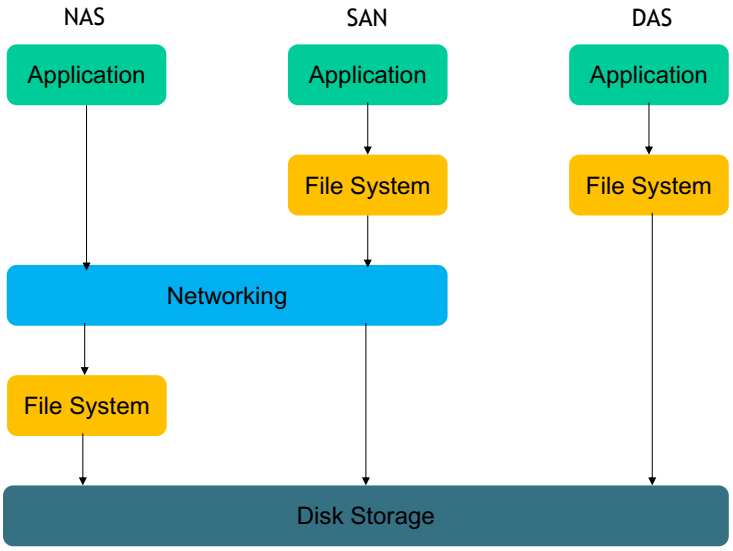
- **Hardware vs Software RAID**
  - **Hardware RAID**. Implemented via a **dedicated RAID controller card** or built into the motherboard.
    - **Pros**
      - ✔ **High performance**: RAID logic is offloaded from the CPU.
      - ✔ Usually includes a battery-backend **cache to improve write speed** and **protect against power loss**.
    - **Cons**
      - ✘ **Portability issue**: if the controller fails and we move the disks to another machine with a different controller, the array might not be recognized.
      - ✘ **Expensive**.
  - **Software RAID**. Managed by the **operating system** (e.g., Linux `mdadm`, Windows Storage Spaces).
    - **Pros**
      - ✔ **Cheap** and **flexible**.
      - ✔ **Easy to migrate**: disks can often be moved to a different machine and reassembled.
    - **Cons**
      - ✘ **Slower performance** because the host CPU handles parity and I/O operations.
      - ✘ **Less reliable** under high load or system crashes.
      - ✘ Vulnerable to the **consistent update problem** (page 67).

2.2.2.5 DAS, NAS and SAN

As the last argument, we introduce **three different typologies** of storage systems:

- **Direct Attached Storage (DAS)** is a **storage system directly attached to a server or workstation**. They are visible as disks/volumes by the client OS.
- **Network Attached Storage (NAS)** is a **computer connected to a network that provides only file-based data storage services** (e.g. FTP, Network File System) to other devices on the network and is visible as File Server to the client OS.
- **Storage Area Networks (SAN)** are **remote storage units connected to a PC using a specific networking technology** (e.g. Fiber Channel) and are visible as disks/volumes by the client OS.

In the following schema, we can see a simple architectural comparison.



✔ **DAS features**

DAS is a **storage system directly attached to a server or workstation**. The term is used to differentiate non-networked storage from SAN and NAS. The **main features** are:

- Limited scalability.
- Complex manageability.
- Limited performance.
- To read files in other machines (the "file sharing" protocol of the OS must be used).

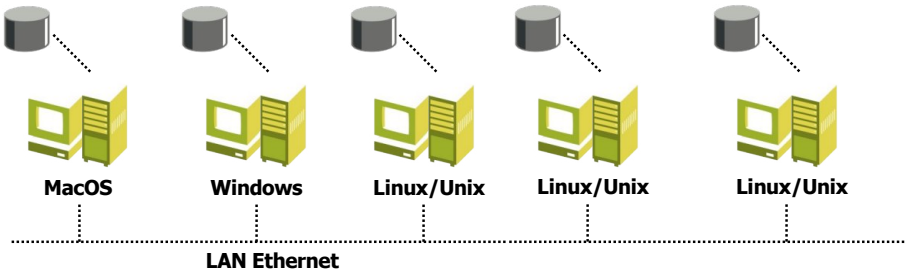


Figure 9: DAS architecture.

Note that all the **external disks connected to a PC with a point-to-point protocol can be considered DAS**.

✔ NAS features

A NAS unit is a **computer connected to a network that provides only file-based data storage services to other devices on the network**. NAS systems contain one or more hard disks, often organized into logical redundant storage containers or RAID. Finally, **NAS provides file-access services to the hosts connected to a TCP/IP network through Networked File Systems/SAMBA**. Each NAS element has its IP address. Furthermore, each NAS has good scalability.

The **main differences between DAS and NAS** are:

- DAS is simply an **extension of an existing server** and is **not necessarily networked**.
- NAS is designed as an easy and self-contained solution for **sharing files over the network**.

Regarding **performance**, NAS depends mainly on the speed and congestion of the network.

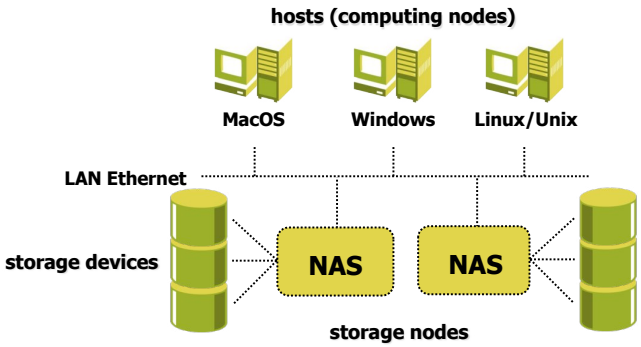


Figure 10: NAS architecture.

✔ SAN features

SANs are remote **storage units connected to servers using a specific networking technology**. SANs have a particular network dedicated to accessing storage devices. It has two distinct networks: one TCP/IP and another dedicated network (e.g. Fiber Channel). It has a high scalability.

The **main difference between a NAS and a SAN** is that:

- **NAS appears to the client OS as a file server**. Then, the client can map network drives to shares on that server.
- **A disk available through a SAN still appears to the client OS as a disk**. It will be visible in the disks and volumes management utilities (along with the client's disks) and available to be formatted with a file system.

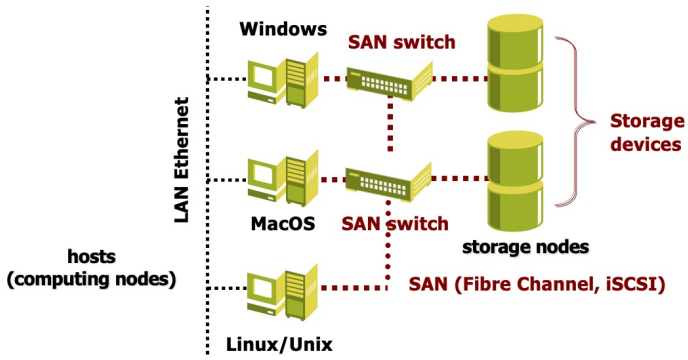


Figure 11: SAN architecture.

	Application Domain	Advantages	Disadvantages
DAS	• Budget constraints • Simple storage solutions	• Easy setup • Low cost • High performance	• Limited accessibility • Limited scalability • No central management and backup
NAS	• File storage and sharing • Big Data	• Scalability • Greater accessibility • Performance	• Increased LAN traffic • Performance limitations • Security and reliability
SAN	• DBMS • Virtualized environments	• Improved performance • Greater scalability • Improved availability	• Costs • Complex setup and maintenance

Figure 12: DAS vs. NAS vs. SAN

2.2.3 Networking (architecture and technology)

2.2.3.1 Fundamental concepts

In the data centre, servers' *performance increases* over time, and the demand for inter-server *bandwidth also increases*.

A **solution** can be to double the aggregate compute capacity or the aggregate storage simply by **doubling the number of compute or storage elements**.

The **doubling leaf bandwidth** is used since the networking has no straightforward horizontal scaling solution. Then, with **twice as many servers**, we will have **twice as many network ports and thus twice as much bandwidth**.

What is a bisection bandwidth?

Bisection bandwidth is a **measure of network performance**, **defined as the bandwidth available between two equal-sized partitions when a network is bisected**. This measure accounts for the *bottleneck bandwidth of the entire network*, providing a representation of the actual bandwidth available in the system. The bisection should be done to minimize the bandwidth between the two partitions. It is often used to evaluate and compare networks for parallel architectures, including point-to-point communication systems or on-chip micro-networks.

Assuming that every server needs to talk to every other server, we need to double not just leaf bandwidth but bisection bandwidth.

How to design a data centre network?

There are many design principles to follow:

- **Very scalable** in order to support a vast number of servers;
- **Minimum cost** in terms of basic building blocks (e.g. switches);
- **Modular** to reuse simple basic modules;
- **Reliable** and **resilient**;
- It may exploit novel/proprietary technologies and protocols incompatible with legacy Internet.

The **Data Center Network (DCN)** connects a data centre's computing and storage units to achieve optimum performance. It can be **classified** into **three main categories**:

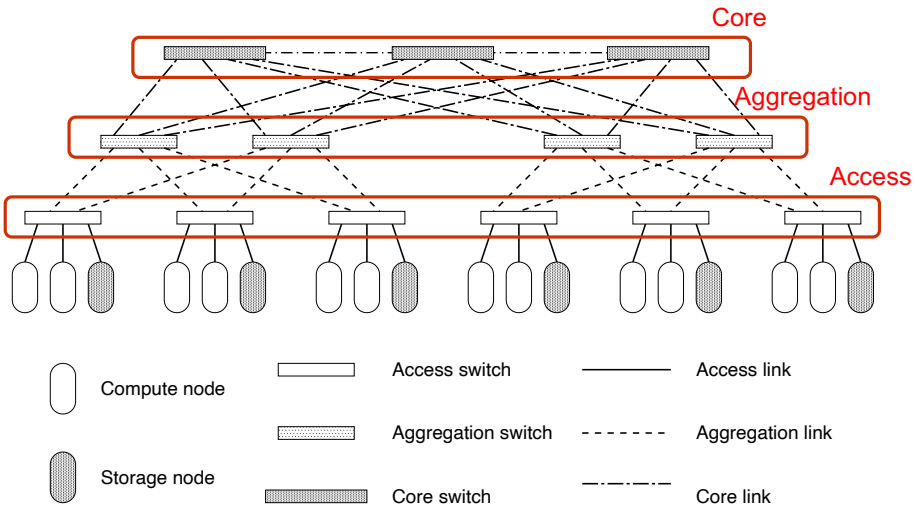
- **DCN Switch-centric architectures**. DCN uses **switches to perform packet forwarding**.[4]
- **DCN Server-centric architectures**. DCN **uses servers with multiple Network Interface Cards (NICs)**[5] **to act as switches** and perform other computational functions.
- **DCN Hybrid architectures**. DCN **combines switches and servers** for packet forwarding.

[4] **Packet forwarding** is the passing of packets from one network segment to another by nodes on a computer network.

[5] A **Network Interface Cards (NICs)** is a computer hardware component that connects a computer to a computer network.

2.2.3.2 Switch-centric: classical Three-Tier architecture

The **Three-Tier architecture**, also called **Three Layer architecture**, **configures the network in three different layers**:



It is a simple Data Center Network topology.

- The **servers** are **connected to the DCN through access switches**.
- **Each access-level switch is connected to at least two aggregation-level switches**.
- **Aggregation-level switches are connected to core-level switches** (gateways).

✔ **Advantages**

1. **Bandwidth can be increased** by **increasing the switches at the core and aggregation layers**, and by using routing protocols such as Equal Cost Multiple Path (ECMP) that equally shares the traffic among different routes.
2. **Very simple** solution.

👎 **Cons**

1. **Very expensive** in large data centers because the upper layers require faster network equipments.
2. **Cost very high** in term of acquisition and energy consumption.

In the **access layer**, there are two possible architectures:

- **ToR (Top-of-Rack) architecture**. All servers in a rack are connected to a ToR access switch within the same rack. The aggregation switches are in dedicated racks or shared racks with other ToR switches and servers.

✔ **Advantages**

1. **Simpler cabling** because the number of cables is limited.
2. **Lower costs** because the number of ports per switch is limited.

👎 **Cons**

Higher complexity for switch management.

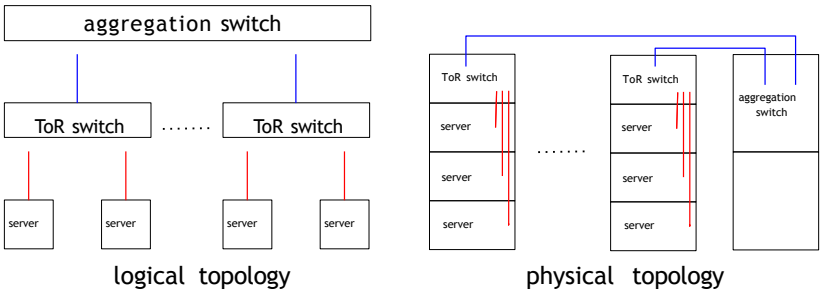


Figure 13: ToR (Top-of-Rack) architecture.

- **EoR (End-of-Row) architecture**. Aggregation switcher are positioned one per corridor, at the end of a line of rack. Servers in a racks are connected directly to the aggregation switch in another rack. Exists a patch panel to connect the servers to the aggregation switch.

✔ **Advantages**

Simpler switch management.

👎 **Cons**

The aggregation switches must have a larger number of ports, then:

1. **Complex cabling**.
2. **Longer cables** then **higher costs**.

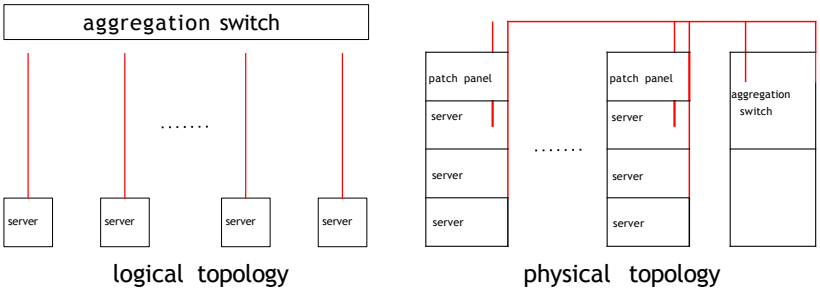


Figure 14: EoR (End-of-Row) architecture.

2.2.3.3 Switch-centric: Leaf-Spine architectures

In the following section we present two Leaf-Spine architectures: the Leaf-Spine model and the Pod-based model (Fat Tree Network).

The **Leaf-Spine architecture** consists of **two levels of interconnection**:

1. The **leaf** (which is a ToR switch);
2. The **spine** (which has dedicated switches, aggregation switches).

In practice, servers have two interfaces connected to two ToR switches to provide fault tolerance.

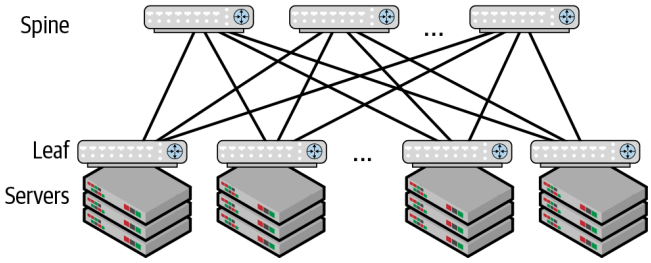


Figure 15: Leaf-Spine architecture.

Now we will explain the Leaf-Spine architecture. If m is the number of **mid-stage switches** and n is the **number of inputs and outputs**, the Leaf-Spine topology is as follows:

- Each switch module is bi-directional.
  - *Leaf* has $2k$ bidirectional ports per module;
  - *Spine* has k bidirectional ports per module.
- Each path traverses either 1 or 3 modules.

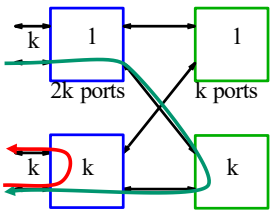


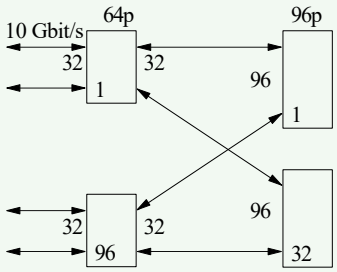
Figure 16: Explanation of Leaf-Spine architecture.

The **advantages** are: use of homogeneous equipment, simple routing, the number of hops is the same for any pair of nodes, small blast radius.

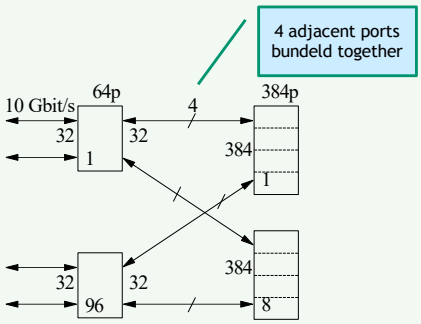
Example 10

There are 3072 servers and 3072 ports available at 10 Gbit/s. Provides a leaf-spine design.
There are **two possible designs**.

1. The first consists of 96 switches with 64 ports and 32 switches with 96 ports.



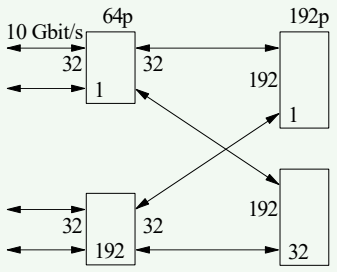
2. The second has only 8 switches but they have more ports: 384 ($8 \times 384 = 3072$).



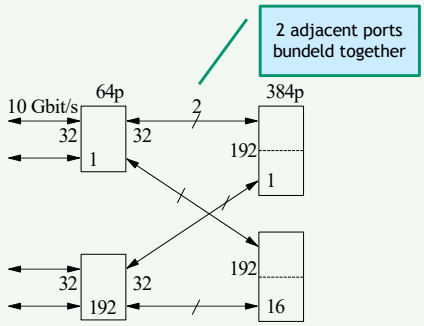
Example 11

There are 6144 servers and 6144 ports available at 10 Gbit/s. Provides a leaf-spine design.
There are **two possible designs**.

1. The first consists of 192 switches with 64 ports and 32 switches with 192 ports.



2. The second has only 16 switches but they have more ports: 384 ($16 \times 384 = 6144$).



The **Pod-based model**, also called **Fat Tree Network**, is another network architecture used to **increase the scaling** feature respecting the leaf-spine.

It transforms each group of spine-leaf into a **PoD (Point of Delivery)**[6] and adds a super spine layer.

It is a **highly scalable** and **cost-effective** DCN architecture designed to **maximise bisection bandwidth**. It can be built using standard Gigabit Ethernet switches with the same number of ports.

It is composed by a ***leaf*** of $2k^2$ bidirectional ports:

- k^2 ports to the servers;
- k^2 ports to the data center network.

In general, let k^2P servers: there are $2kP$ switches with $2k$ ports and k^2 switches with P ports. Using the Fat-Tree model, **the P value is 2k, so for $2k^3$ servers, there are $5k^2$ switches with 2k ports** ($k^2 + 2k \cdot 2k$).

At the **edge layer**, there are $2k$ pods (groups of servers), each with k^2 servers.

- Each edge switch is directly connected to k servers in a pod and k aggregation switches.
- A Fat-Tree network with $2k$-port commodity switches can accomodate $2k^3$ servers in total.
- k^2 core switches with $2k$-port each, each one connected to $2k$ pods.
- Each aggregation switch is connected to k core switches.

[6] A Point Of Delivery is a module or group of network, compute, storage and application components that work together to deliver a network service. The PoD is a repeatable pattern and its components increase the modularity, scalability and manageability of data.

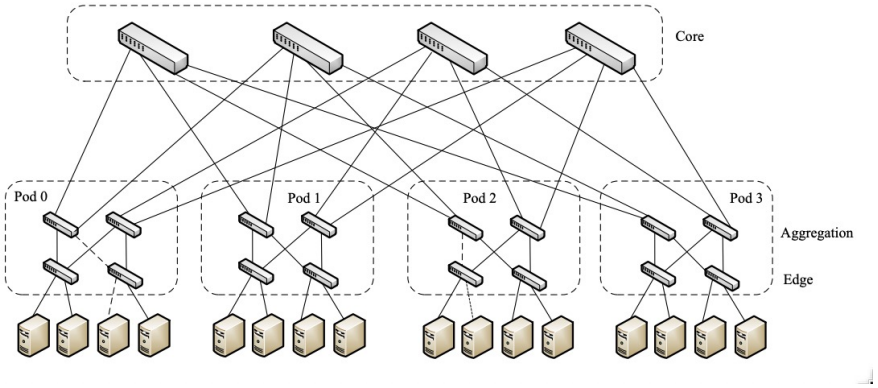


Figure 17: Fat-Tree Network, with $k = 2$, 4 pods, 16 servers, 20 switches.

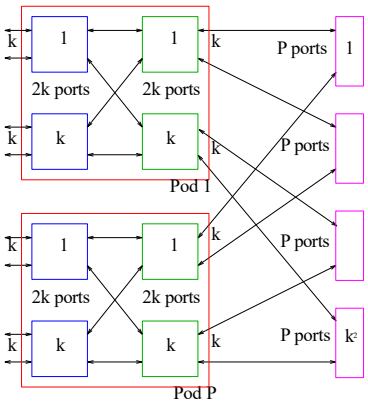


Figure 18: Explanation of Fat-Tree Network.

2.2.3.4 Server-centric and hybrid architectures

CamCube is a **server-centric architecture** typically proposed for building container-sized data centres.

Advantages

It can **reduce implementation and maintenance costs** by using only servers to build the Data Center Network. It also exploits network locality to **increase communication efficiency**.

Disadvantages

It **requires servers with Multiple Network Interface cards** to build a 3D tours network, **long paths** and **high routing complexity**.

The hybrid architectures are **DCell**, **BCube** and **MDCube**.

A **DCell** is a **scalable and cost-effective hybrid architecture** that uses switches and servers for packet forwarding. It is a recursive architecture and uses a basic building block called $DCell_0$ to construct larger DCells.

$DCell_k$ ($k > 0$) denotes a level-k DCell **constructed by combining** $n + 1$ **servers** in $DCell_0$. A $DCell_0$ has n ($n < 8$) **servers directly connected by a commodity switch**.

Disadvantages: **long communication paths**, many required Network Interface Cards and **increased cabling costs**.

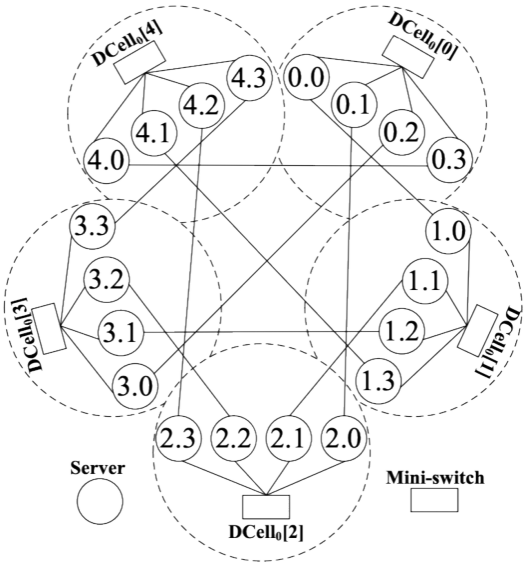


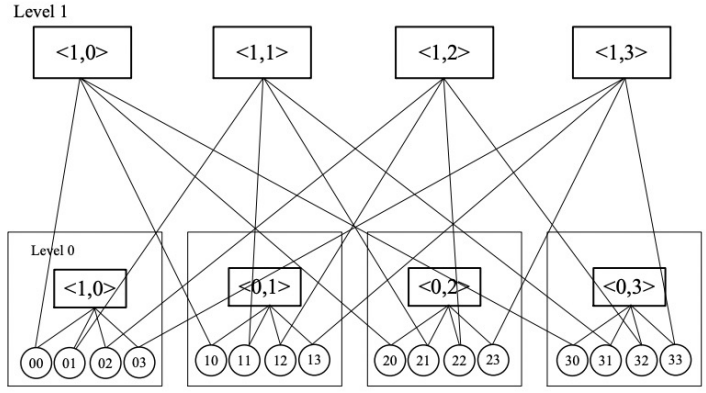
Figure 19: DCell hybrid architecture.

BCube is a **hybrid and cost-effective architecture** that scales through recursion. It provides **high bisection bandwidth** and **graceful throughput degradation**.

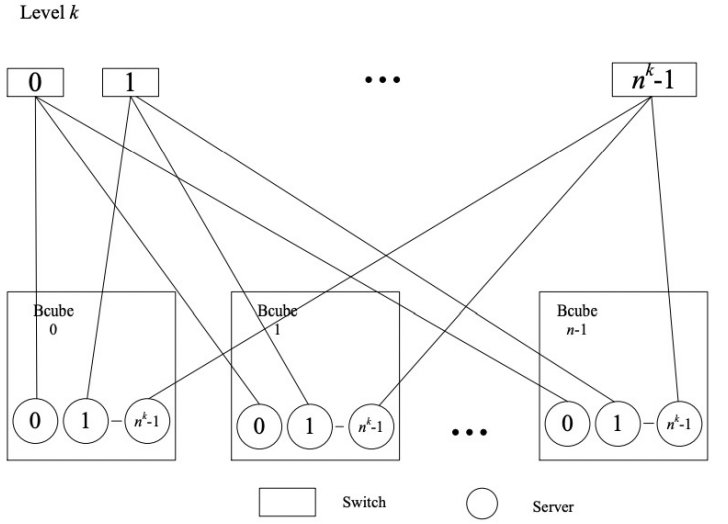
It uses BCube as a building block, consisting of n servers connected to an n-port switch.

A $BCube_k$ $(k > 0)$ is constructed with n $BCube_{k-1}s$ and n^k n-port switches. In a $BCube_k$ there are $n^{(k+1)}$ $k+1$-port servers and $k+1$ layers of switches.

Disadvantages: **limited scalability** and **high cabling costs** (NICs reason).



(a) Request phase.



(b) Grant phase.

Figure 20: BCube hybrid architecture.

MDCube is designed to **reduce the number of cables used to connect containers**.

- Each container has an ID which is mapped to a multidimensional tuple.
- Each container is connected to a neighbouring container with a different tuple in one dimension.
- There are two types of connections: Intra-container links and high-speed inter-container links.

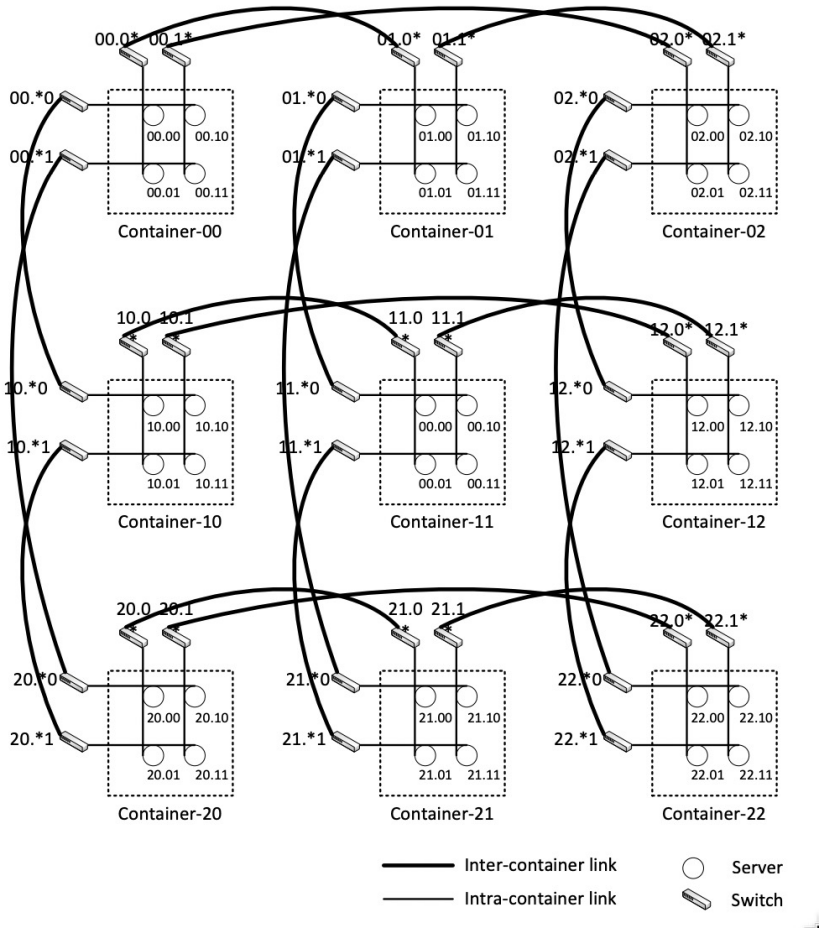


Figure 21: MDCube hybrid architecture.

4.2 Disk performance

4.2.1 HDD

We can calculate some performance metrics related to the types of delay of HDD (page 41).

- **Full Rotation Delay** R is:

$$R = \frac{1}{\texttt{DiskRPM}} \tag{50}$$

And in seconds:

$$R_{\text{sec}} = 60 \times R \tag{51}$$

From the R_{sec} we can also calculate the **total rotation average**:

$$T_{\text{rotation AVG}} = \frac{R_{\text{sec}}}{2} \tag{52}$$

It is half of a full rotation, because on average, the sector will be halfway around the platter from the current head position.

- **Seek Time**. The time to seek from one track to another depends on the distance moved. In real systems, this relation isn't perfectly linear, but it's often approximated as (**seek average**):

$$T_{\text{seek AVG}} = \frac{T_{\text{seek MAX}}}{3} \tag{53}$$

Where $T_{\text{seek MAX}}$ is the **time for the longest possible seek** (from *outermost to innermost* track) and the division by 3 assumes a **uniform random distribution of seeks** across the disk.

- **Transfer time**. It is the **time that data is either read from or written to the surface**. It **includes** the time the **head needs to pass on the sectors** and the **I/O transfer**:

$$T_{\text{transfer}} = \frac{\text{R/W of a sector}}{\text{Data transfer rate}} \tag{54}$$

The **total time to complete a disk I/O operation** is called the $T_{\text{I/O}}$ **Service Time**:

$$T_{\text{I/O}} = T_{\text{seek}} + T_{\text{rotation}} + T_{\text{transfer}} + T_{\text{overhead}} \tag{55}$$

If the disk is shared among processes, we must also consider **queueing time**. And the **Response Time** is:

$$T_{\text{response}} = T_{\text{queue}} + T_{\text{I/O}} \tag{56}$$

Where T_{queue} depends on queue length, disk utilization rate, variance in request time, arrival rate of I/O requests.

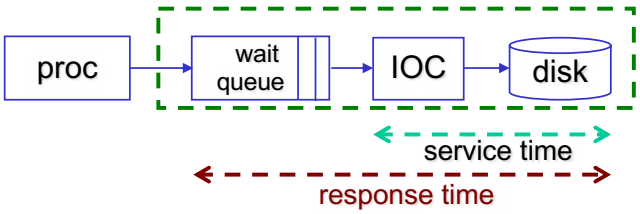


Figure 40: Service and response time.

Exercise 1: Mean Service Time of an I/O operation

The data of the exercise are:

- Read/Write of a sector of 512 bytes (0.5 KB)
- Data transfer rate: 50 MB/sec
- Rotation speed: 10000 RPM (Round Per Minute)
- Mean seek time: 6 ms
- Overhead Controller: 0.2 ms

The goal is to calculate the average I/O service time. To calculate the *service time* $T_{\text{I/O}}$, we need the following information:

- ✔ T_{seek}, which we already have, and it is 6 ms.
- ✘ T_{rotation}
- ✘ T_{transfer}
- ✔ T_{overhead}, which we already have, and it is 0.2 ms.

We also know the rotation and transfer information, but we want to know the *mean* service time. Then we calculate the total rotation average $T_{\text{rotation AVG}}$:

$$\begin{aligned} R &= \frac{1}{\texttt{DiskRPM}} = \frac{1}{10000} = 0.0001 \\ R_{\text{sec}} &= 60 \cdot R = 60 \cdot 0.0001 = 0.006 \text{ seconds} \\ T_{\text{rotation AVG}} &= \frac{R_{\text{sec}}}{2} = \frac{0.006}{2} = 0.003 \text{ seconds} = 3 \text{ ms} \end{aligned}$$

Finally, the transfer time is easy to calculate because we have the R/W of a sector and the data transfer rate. First we do a conversions from

megabytes to kilobytes:

$$\begin{aligned} \text{Data transfer rate:} \quad & 50 \text{ MB/sec} \\ &= 50 \cdot 1024 \text{ KB/sec} \\ &= 51200 \text{ KB/sec} \\ T_{\text{transfer}} &= \frac{0.5 \text{ KB/sec}}{51200 \text{ KB/sec}} \\ &= 0.000009765625 \text{ sec} \cdot 1000 \\ &= 0.009765625 \text{ ms} \approx 0.01 \text{ ms} \end{aligned}$$

The exercise can be completed by calculating the mean I/O service time required:

$$\begin{aligned} T_{\text{I/O}} &= T_{\text{seek}} + T_{\text{rotation}} + T_{\text{transfer}} + T_{\text{overhead}} \\ T_{\text{I/O}} &= 6 + 3 + 0.01 + 0.2 = 9.21 \text{ ms} \end{aligned}$$

The I/O service time computed in the previous exercise (9.21 ms) assumes a **worst-case scenario**. This is very useful for understanding disk behavior, but it **doesn't always reflect what happens in real workloads**. It assumes:

1. **Every time** we read a small file or sector,
2. The **read must seek** to a new track,
3. And then **wait for rotation** to bring the right sector under the head

This is worst-case behavior, and happens when **files are very small**, so each one is just one block (e.g., 512 bytes); or disk is **heavily externally fragmented**, so even larger files are broken into scattered blocks. In such cases, **every access pays both seek and rotational delay**, making the access time slow and constant.

We can introduce a new metric that comes from the idea of measuring locality: **how often we can avoid seek and rotation delays**. We define **Data Locality** DL as:

$$DL = \% \text{ of blocks that can be accessed without new seek or rotation} \quad (57)$$

If **locality is high**, then most of our **data** is laid out in a nice, **sequential way**, therefore performance improves significantly. So, **locality determines whether our performance is close to best-case or worst-case**.

Thanks to the Data Locality, it is possible to calculate the **Average Service Time** by modifying the terms of *seek* and *rotation* of the Service Time equation (page 130):

$$T_{\text{I/O AVG}} = (1 - DL) \cdot (T_{\text{seek}} + T_{\text{rotation}}) + T_{\text{transfer}} + T_{\text{controller}} \quad (58)$$

Exercise 2: Data Locality

The data of the exercise are:

- Read/Write of a sector of 512 bytes (0.5 KB)
- Data Locality: $DL = 75\%$
- Data transfer rate: 50 MB/sec
- Rotation speed: 10000 RPM (Round Per Minute)
- Mean seek time: 6 ms
- Overhead Controller: 0.2 ms

Since the Data Locality is 75%, only 25% of the operations are affected by the DL:

$$(1 - DL) = (1 - 0.75) = 0.25$$

See the exercise on page 131 to understand the values of T_{seek}, T_{rotation}, T_{transfer} and T_{overhead}:

- $T_{\text{seek}} = 6$
- $T_{\text{rotation}} = 3$
- $T_{\text{transfer}} = 0.01$
- $T_{\text{overhead}} = 0.2$

Finally the average time for read/write a sector of 0.5 KB with a DL of 75% is:

$$\begin{aligned} T_{\text{I/O AVG}} &= 0.25 \cdot (6 + 3) + 0.01 + 0.2 \\ &= 0.25 \cdot 9 + 0.21 \\ &= 2.46 \text{ ms} \end{aligned}$$

Exercise 3: Influence of "Not Optimal" Data Allocation

The data of the exercise are:

- 10 blocks of 1/10 MB for each block (10 blocks of 1/10 MB "not well" distributed on disk)
- $T_{\text{seek}} = 6$ ms
- $T_{\text{rotation AVG}} = 3$ ms
- Data transfer rate: 50 MB/sec

In the exercise you were asked to calculate the time taken to transfer a 1 MB file with 100% and 0% data locality:

- Data Locality equals to 100%:
  - An initial seek (6 ms)
  - A total rotation average (3 ms)
  - Now it's possible to do the 1MB global transfer directly because there are no blocks to seek or rotation latency:

$$1 \text{ MB of } 50 \text{ MB} = \frac{1}{50} = 0.02 \text{ seconds } \cdot 1000 = 20 \text{ ms}$$

  - The total time is:

$$T = 6 + 3 + 20 = 29 \text{ ms}$$

- Data Locality equals to 0%:
  - An initial seek (6 ms)
  - A total rotation average (3 ms)
  - In this case, it's not possible to do a global transfer directly, because each block is affected by the seek or rotation latency. Then we have to transfer block by block and calculate the delay:

$$1 \text{ MB of } 10 \text{ MB} = \frac{1}{10} = 0.1 \text{ seconds } \cdot 1000 = 100 \text{ ms}$$

  - The total time is:

$$T = (6 + 3 + 2) \cdot 10 = 110 \text{ ms}$$

    Where 10 is the number of blocks.

Note: the controller times is not considered.

4.2.2 RAID

We can calculate some performance metrics related to the RAID technology (page 57).

- Let's assume:
  - A constant Failure Rate;
  - An exponentially distributed time to failure;
  - The case of independent failures.

  (conditions usually used to determine the disk MTTF).

  The **Mean Time To Failure of a disk array** $\texttt{MTTF}_{\textbf{diskArray}}$ is equal to the relationship between the `MTTF` of a single disk and the number of disks:

$$\texttt{MTTF}_{\text{diskArray}} = \frac{\texttt{MTTF}_{\text{singleDisk}}}{\#\,\texttt{Disks}} \tag{59}$$

  Large disk arrays are **too unstable to be used without any fault tolerance approach**. Disks do not have huge `MTTF` since it is highly probable they will be replaced in a "short time". Note that the **RAID 0 has no redundancy**!

$$\texttt{MTTF}_{\texttt{RAID 0}} = \texttt{MTTF}_{\text{diskArray}} = \frac{\texttt{MTTF}_{\text{singleDisk}}}{\#\,\texttt{Disks}} \tag{60}$$

- RAID levels greater than level zero use redundancy to improve reliability. Then, when a disk fails, it should be replaced, and the information should be reconstructed on the new disk using the redundant information. The `MTTR` is the **time needed for this action**! As always, the N value is the number of disks in the array. The **Mean Time To Failure of a RAID** $\texttt{MTTF}_{\texttt{RAID}}$ (except the level zero!) is:

$$\texttt{MTTF}_{\texttt{RAID}} = \left(\frac{\texttt{MTTF}_{\text{singleDisk}}}{N}\right) \times \left(\frac{1}{\text{Probability}_{\text{additionalCriticalFailuresInMTTR}}}\right) \tag{61}$$

  Where:
  - $\frac{\texttt{MTTF}_{\text{singleDisk}}}{N}$ is the **MTTF for the array of N disks**.
  - $\frac{1}{\text{Probability}_{\text{additionalCriticalFailuresInMTTR}}}$ is the **probability of other critical failures in the array before repairing the failed disk**. The RAID level and type of redundancy determine it.

  In detail, the **Mean Time To Failure of each RAID level** (except the zero) is:

  - `RAID 1` - With a single copy of each disk, one drive can fail, and if we are lucky, $N \div 2$ drives can fail without data loss. Then the **MTTF of** $\texttt{RAID 1}$ $\texttt{MTTF}_{\textbf{RAID 1}}$ is:

$$\texttt{MTTF}_{\text{RAID 1}} = \left(\frac{\texttt{MTTF}_{\text{singleDisk}}}{N}\right) \times \left(\frac{1}{\text{Probability}_{\text{2ndCriticalFailureInMTTR}}}\right) \tag{62}$$

$$\text{Probability}_{\text{2ndCriticalFailureInMTTR}} = \left(\frac{1}{\texttt{MTTF}_{\text{singleDisk}}}\right) \times \texttt{MTTR} \quad (63)$$

Where:

* $\frac{1}{\texttt{MTTF}_{\text{singleDisk}}}$ is the **failure rate for the copy of the failed disk**.
* `MTTR` is the **period of interest before replacement**.

– `RAID 0 + 1` - When one disk in a stripe group fails, the entire group goes off. Then the **MTTF of `RAID 01`** $\texttt{MTTF}_{\textbf{RAID 0 + 1}}$ is:

$$\texttt{MTTF}_{\text{RAID 01}} = \left(\frac{\texttt{MTTF}_{\text{singleDisk}}}{N}\right) \times \left(\frac{1}{\text{Probability}_{\text{2ndCriticalFailureInMTTR}}}\right) \quad (64)$$

It is not the same as RAID 1 because the probability is:

$$\text{Probability}_{\text{2ndCriticalFailureInMTTR}} = \left(\frac{G}{\texttt{MTTF}_{\text{singleDisk}}}\right) \times \texttt{MTTR} \quad (65)$$

Where:

* G is the **number of disks in a stripe group**.
* $\frac{G}{\texttt{MTTF}_{\text{singleDisk}}}$ is the **failure rate for one of the disks in the other group**.
* `MTTR` is the **period of interest before replacement**.

– `RAID 1 + 0` - To fail, the same copy in both groups has to fail, but multiple failure can be tolerated. Then the **MTTF of `RAID 10`** $\texttt{MTTF}_{\textbf{RAID 1 + 0}}$ is:

$$\texttt{MTTF}_{\text{RAID 10}} = \left(\frac{\texttt{MTTF}_{\text{singleDisk}}}{N}\right) \times \left(\frac{1}{\text{Probability}_{\text{2ndCriticalFailureInMTTR}}}\right) \quad (66)$$

It is not the same as RAID 1 because the probability is:

$$\text{Probability}_{\text{2ndCriticalFailureInMTTR}} = \left(\frac{1}{\texttt{MTTF}_{\text{singleDisk}}}\right) \times \texttt{MTTR} \quad (67)$$

Where:

* $\frac{1}{\texttt{MTTF}_{\text{singleDisk}}}$ is the **failure rate for the copy of the failed disk**.
* `MTTR` is the **period of interest before replacement**.

– `RAID 4` and `RAID 5` - To fail, two disks have to fail before replacement. Then the **MTTF of `RAID 4`** $\texttt{MTTF}_{\textbf{RAID 4}}$ and the **MTTF of `RAID 5`** $\texttt{MTTF}_{\textbf{RAID 5}}$ is:

$$\texttt{MTTF}_{\text{RAID 4}} = \texttt{MTTF}_{\text{RAID 5}} = \left(\frac{\texttt{MTTF}_{\text{singleDisk}}}{N}\right) \times \left(\frac{1}{\text{Probability}_{\text{2ndFailureInMTTR}}}\right) \quad (68)$$

And the probability is:

$$\text{Probability}_{\text{2ndFailureInMTTR}} = \left(\frac{(N-1)}{\texttt{MTTF}_{\text{singleDisk}}}\right) \times \texttt{MTTR} \quad (69)$$

Where:

  * $\frac{(N-1)}{\texttt{MTTF}_{\text{singleDisk}}}$ is the **failure rate for one of the other disks**.
  * `MTTR` is the **period of interest before replacement**.

- `RAID 6` - Two disks failures at the same time are tolerated. Then the **MTTF of `RAID 6` $\texttt{MTTF}_{\text{RAID 6}}$** is:

$$\texttt{MTTF}_{\text{RAID 6}} = \left(\frac{\texttt{MTTF}_{\text{singleDisk}}}{N}\right) \times \left(\frac{1}{\text{Probability}_{\text{2ndAnd3rdFailureInMTTR}}}\right) \tag{70}$$

And the probability is:

$$\text{Probability}_{\text{2ndAnd3rdFailureInMTTR}} = \text{Probability}_{\text{2ndFailure}} \times \text{Probability}_{\text{3ndFailure}} \tag{71}$$

Where:

  * $\text{Probability}_{\text{2ndFailure}}$:

$$\text{Probability}_{\text{2ndFailure}} = \left(\frac{(N-1)}{\texttt{MTTF}_{\text{singleDisk}}}\right) \times \texttt{MTTR} \tag{72}$$

    · $\frac{(N-1)}{\texttt{MTTF}_{\text{singleDisk}}}$ is the **failure rate for one of the other disks**.

    · `MTTR` is the **period of interest before the replacement**.

  * $\text{Probability}_{\text{3ndFailure}}$:

$$\text{Probability}_{\text{3ndFailure}} = \left(\frac{(N-2)}{\texttt{MTTF}_{\text{singleDisk}}}\right) \times \frac{\texttt{MTTR}}{2} \tag{73}$$

    · $\frac{(N-2)}{\texttt{MTTF}_{\text{singleDisk}}}$ is the **failure rate for one of the remaining disks**.

    · $\frac{\texttt{MTTR}}{2}$ is the **average overlapping period between first and second disk replacement** (both disk not yet replaced).

RAID level	Metric
RAID 0	$\texttt{MTTF}_{\texttt{RAID 0}} = \frac{\texttt{MTTF}_{\text{singleDisk}}}{N}$
RAID 1 + 0	$\texttt{MTTF}_{\texttt{RAID 10}} = \frac{\left(\texttt{MTTF}_{\text{singleDisk}}\right)^2}{(N \times \texttt{MTTR})}$
RAID 0 + 1	$\texttt{MTTF}_{\texttt{RAID 01}} = \frac{\left(\texttt{MTTF}_{\text{singleDisk}}\right)^2}{(N \times G \times \texttt{MTTR})}$
RAID 4	$\texttt{MTTF}_{\texttt{RAID 4}} = \frac{\left(\texttt{MTTF}_{\text{singleDisk}}\right)^2}{(N \times N-1 \times \texttt{MTTR})}$
RAID 5	$\texttt{MTTF}_{\texttt{RAID 5}} = \frac{\left(\texttt{MTTF}_{\text{singleDisk}}\right)^2}{(N \times N-1 \times \texttt{MTTR})}$
RAID 6	$\texttt{MTTF}_{\texttt{RAID 6}} = \frac{2 \times \left(\texttt{MTTF}_{\text{singleDisk}}\right)^3}{\left(N \times N-1 \times N-2 \times \texttt{MTTR}^2\right)}$

Table 11: MTTF summary RAID levels.