MAD 1 WEEK 7

LEC 1:

Memory Hierarchy

Backend Systems

Types of Storage elements:

On-chip Registers (10s to 100s of bytes):

- **Purpose**: These are tiny, super-fast memory locations directly inside the CPU (central processing unit).
- **Function**: They store the data the CPU needs immediately for calculations, like a "working scratchpad" for the processor.
- **Speed**: The fastest type of memory, used to keep the CPU working as efficiently as possible.

SRAM (Static Random Access Memory) - Cache Memory (0.1s to 1MB):

- **Purpose**: This is a bit slower than on-chip registers but still very fast. It's stored close to the CPU and helps speed up data access.
- **Function**: It holds copies of data and instructions that the CPU might need repeatedly, so the CPU doesn't have to go all the way to main memory (DRAM) each time.
- **Speed**: Very fast but more expensive, so we use only a small amount as cache.

DRAM (Dynamic Random Access Memory) - Main Memory (10s to 100s of GB):

- **Purpose**: This is your computer's main memory, also known as RAM.
- **Function**: It temporarily holds data and programs currently in use, so the CPU can quickly access them. When you open a program, it loads from your storage (like SSD or HDD) into DRAM for faster access.
- **Speed**: Much faster than storage like SSDs and hard drives but slower than the CPU cache.

SSD (Solid State Drive) - Flash Storage (100GB to 1TB):

- **Purpose**: This is a type of permanent storage that holds your files and programs even when the computer is turned off.

- **Function**: SSDs use flash memory, which has no moving parts, making them much faster than traditional hard drives for accessing and saving data.
- **Speed**: Much faster than hard drives, so SSDs are great for quickly loading your operating system, applications, and frequently used files.

Magnetic Disk (Hard Drive) - HDD Storage (1TB and up):

- **Purpose**: Traditional hard drives store large amounts of data more affordably than SSDs.
- **Function**: HDDs store data on spinning magnetic disks inside the drive. They're slower than SSDs because of the time it takes to physically move parts inside the drive, but they're still useful for storing large files and data you don't access often.
- **Speed**: Slower than SSDs, so usually used for bulk storage where speed is less important, like for archiving files or storing large multimedia libraries.

Latency
It refers to the time it takes for a data packet to travel from the source to the destination (here storage element).Lower latency is more desirable.
REGISTER<SRAM<DRAM<SSD<HDD

Throughput
It is the number of data packets that can be transferred from one place to another in a given amount of time.Higher is more desirable.
DRAM<SSD<HDD

Density
It is the amount of data that can be stored in a given amount of physical space. Higher density often leads to more efficient resource utilization, reduced physical footprint.
HDD>SDD>DRAM<SRAM>REGISTER

Computer Architecture and Memory Hierarchy

Computer Architecture is about how different components of a computer, especially memory, are organized to make the system efficient and fast.

Memory Hierarchy:

- **Purpose**: The memory hierarchy organizes different memory levels based on their distance from the CPU and their performance. By arranging memory in layers with varying speeds, sizes, and costs, the system can access frequently used data quickly and cost-effectively store less-used data.
- **Structure**:

    - **CPU Registers and Caches (L1, L2, L3)**: These are the closest to the CPU, blazing-fast but small in size. Registers hold data for immediate processing, and caches store recently accessed data to prevent repeated data fetching from slower memory.
    - **Main Memory (RAM)**: This is larger than cache but slower, used to store data and programs that the CPU needs while the system is running.
    - **Secondary Storage (HDD/SSD)**: This is high-capacity, long-term storage. SSDs are faster than HDDs but cost more per gigabyte. They store files and programs not in active use but needed long-term.
- **Goal**: The hierarchy's purpose is to move frequently accessed data closer to the CPU for faster access, optimizing the system's overall speed and efficiency.

Cold Storage

Cold Storage is a type of data storage optimized for long-term retention of inactive or rarely accessed data.

- **Purpose**: Cold storage is designed to hold data that isn't accessed often but still needs to be stored for compliance, archival, or business purposes.
- **Examples**: Services like Amazon Glacier and Google Coldline are popular cold storage solutions.
- **Characteristics**:
    - **Cost-Effective**: Cold storage is cheaper than other storage options because it's optimized for data that doesn't need immediate access.
    - **Efficiency**: It's meant for infrequent data retrieval, offering lower costs and reduced power usage.
- **Use Cases**: Ideal for storing old backups, records, or legal documentation.
- **Application Consideration**: When planning for an application's storage needs, cold storage is a good fit for inactive data, freeing up faster memory for active data, which can help maintain the app's speed.

LEC 2:

https://pdsaiitm.github.io/week-2/summary.html#complexity
https://pdsaiitm.github.io/week-2/summary.html#searching-algorithm

O(1) - Constant Time

- **Description**: The algorithm's runtime does not depend on the input size; it remains constant.

- **When Used**: Ideal for operations like accessing an element in an array or dictionary (hash table) by index/key. It’s very efficient and does not change with the size of the data.
- **Example**: Retrieving a value from a dictionary by key, calculating the sum of two numbers.

O(log N) - Logarithmic Time

- **Description**: The runtime grows very slowly compared to the input size; doubling the input size increases runtime only slightly.
- **When Used**: Often seen in algorithms that reduce the problem size at each step, such as binary search.
- **Example**: Searching for an item in a sorted array using binary search.

O(N) - Linear Time

- **Description**: The runtime grows proportionally with the input size.
- **When Used**: Common for algorithms that need to look at each element in the input once, such as linear search or summing all elements in an array.
- **Example**: Checking each item in a list to find a specific value.

O(N^k) - Polynomial Time (Quadratic, Cubic, etc.)

- **Description**: The runtime grows quickly as the input size increases, especially if k is large. It’s inefficient for large datasets.
- **When Used**: Often occurs in algorithms that involve nested loops, where each element is compared with every other element.
- **Examples**:
    - **O(N^2)** (Quadratic): Common in algorithms like bubble sort, insertion sort, and selection sort, where there are nested loops.
    - **O(N^3)** (Cubic): Less common but may appear in algorithms involving complex matrix operations or graph algorithms with triple nested loops.

O(k^N) - Exponential Time

- **Description**: Runtime doubles with each additional input element, leading to extremely rapid growth. This complexity is unmanageable for large inputs.

- **When Used**: Common in brute-force algorithms that explore all possible combinations, such as recursive solutions to the traveling salesman problem or generating all subsets of a set.
- **Example**: Solving the traveling salesman problem by checking every possible route, recursive calculation of Fibonacci numbers (unless optimized with memoization).

Summary Table

Complexity	Runtime Growth Rate	Typical Use Case
O(1)	Constant	Array or dictionary access by index/key
O(log N)	Logarithmic	Binary search
O(N)	Linear	Linear search, summing all elements
O(N^k)	Polynomial	Sorting algorithms like bubble sort, nested loops
O(k^N)	Exponential	Combinatorial problems, brute-force solutions

LEC 3:

Tabular Databases

Tabular databases, like MySQL, organize data into **tables** with **rows and columns**. They are highly structured and often used for applications that require fast data retrieval.

Indexes in Tabular Databases

Indexes are like shortcuts that speed up searching within a database by pre-organizing certain columns. By creating an index on specific columns, you allow the database to find values in those columns much faster, without needing to scan every row.

Indexes in databases are usually stored as tree structures, commonly **B-trees** or **hash indexes**. Here's how they work:

B-Tree Index

- **How it works**: The B-tree index organizes data in a way that allows for fast searching by comparison, making it efficient for finding data within a range of values.
- **When used**: B-trees work well with **comparison operators** like `=`, `>`, `>=`, `<`, `<=`, and `BETWEEN`. They can also handle some `LIKE` searches as long as the search term doesn't start with a wildcard (`%`).
  Examples:
  - `SELECT * FROM table_name WHERE column LIKE 'Pat%'` - This can use a B-tree index to find all names starting with "Pat".
  - `SELECT * FROM table_name WHERE column BETWEEN 100 AND 200` - This can use a B-tree index to efficiently retrieve all values between 100 and 200.

Hash Index

- **How it works**: Hash indexes use a hash function to find the exact location of a value directly, making them very fast for exact matches.
- **When used**: Hash indexes are best for **equality comparisons** (using `=`). However, they are **not** good for range searches (`<`, `>`, etc.) or sorting (`ORDER BY`) because they don't organize values in order.
  Example:
  - `SELECT * FROM table_name WHERE column = 'Pat'` - This uses a hash index to find rows where the column exactly matches "Pat."

Hash indexes work well for **key-value lookups** (similar to a dictionary or phone book) where each search is for a specific, exact match.

In summary:

- **B-Tree Indexes** are flexible, handling range-based searches and certain pattern matches.
- **Hash Indexes** are faster but limited to exact matches, making them ideal for key-value lookups.

LEC 4:

SQL Query example:

-- Select all rows and columns from the 'users' table
```
SELECT *
FROM users;
```

-- Insert a new row into the 'users' table with the name 'Alice' and age 25
```
INSERT INTO users (name, age)
VALUES ('Alice', 25);
```

-- Update the age to 26 for the user with the name 'Alice' in the 'users' table
```
UPDATE users
SET age = 26
WHERE name = 'Alice';
```

-- Delete all rows from the 'users' table where the age is greater than 30
```
DELETE FROM users
WHERE age > 30;
```

NO SQL Query example:

// Select all documents from the 'users' collection
```
db.users.find({});
```

// Insert a new document with the name 'Alice' and age 25 into the 'users' collection
```
db.users.insertOne({ name: 'Alice', age: 25 });
```

// Update the age to 26 for the document with the name 'Alice' in the 'users' collection
```
db.users.updateOne(
    { name: 'Alice' },
    { $set: { age: 26 } }
);
```

// Delete all documents from the 'users' collection where the age is greater than 30
```
db.users.deleteMany({ age: { $gt: 30 } });
```

Alternate Ways to Store Data

- **Key-Value Stores**
    - Data stored in key-value pairs (like a dictionary)
    - Fast for accessing specific keys, not ideal for range queries
    - **Examples**: Redis, DynamoDB, Python Dictionary
    - **Uses**: Caching, session management
- **Column Stores**
    - Data stored in columns instead of rows, good for analytical queries
    - **Examples**: Cassandra, HBase
    - **Uses**: Large-scale data analysis, business intelligence
- **Graph Databases**
    - Data stored in nodes and edges, useful for finding relationships
    - **Examples**: Neo4j, OrientDB
    - **Uses**: Social networks, recommendation engines
- **Time Series Databases**
    - Data organized by time (timestamps), ideal for tracking trends
    - **Examples**: InfluxDB, Prometheus
    - **Uses**: Monitoring, log analysis
    - **Example Queries**: Hits between T1 and T2, average hits per day
- **Search Engines**
    - Optimized for fast search and data analytics
    - **Examples**: Elasticsearch, Solr
    - **Uses**: Full-text search, real-time visualization

ACID Properties of Databases

- **Atomicity**: Each transaction is all or nothing
- **Consistency**: Only valid data is written to the database
- **Isolation**: Transactions don't interfere with each other
- **Durability**: Data remains safe even after failures

LEC 5:

Scaling

- **Scaling Up (Vertical Scaling)**
    - Increasing the power of a single machine (e.g., upgrading CPU, RAM)
    - Good for handling more complex tasks but has hardware limits.
- **Scaling Out (Horizontal Scaling)**
    - Adding more machines or nodes to distribute the workload

    - Allows handling more data and users by spreading tasks across multiple systems.

Replication and Redundancy

- **Replication**
    - Making copies of data across different servers or locations
    - Ensures availability and reliability if one server goes down.
- **Redundancy**
    - Adding extra components or systems that can take over in case of failure
    - Increases system reliability and prevents downtime.

ACID vs BASE

- **ACID** (for traditional databases)
    - **Atomicity**: All or nothing, the transaction must complete fully or not at all
    - **Consistency**: Data is always in a valid state after a transaction
    - **Isolation**: Transactions don't affect each other
    - **Durability**: Data persists even after a crash
- **BASE** (for NoSQL databases)
    - **Basically Available**: The system is always available for reading or writing
    - **Soft State**: Data might be temporarily inconsistent
    - **Eventual Consistency**: The system will become consistent over time, but not immediately

LEC 6:

SQL Injection Example:

A typical HTML form:

```
<form action="/login" method="post">
    <input type="text" name="username" />
    <input type="password" name="password" />
    <input type="submit" value="Login" />
</form>
```

Code to store the data in a database:

```
name = form.request.get('username')
password = form.request.get('password')
query = "SELECT * FROM users WHERE name = '" + name + "' AND password
= '" + password + "'"
db.execute(query)
```

Scenario:

If a user enters a **normal username and password**, the final query will look like this:

```
SELECT * FROM users WHERE name = 'alice' AND password = '1234';
```

This is a **valid query** and will return the user's data.

Potential Threat:

If the user enters **malicious data** (SQL Injection), the query may become:

```
SELECT * FROM users WHERE name = '' OR '1'='1' --' AND password = '';
```

Here's how it works:

- `''`: A blank string for the username.
- **`OR '1'='1'`**: This part of the query always evaluates to **TRUE**, bypassing authentication.
- **`--`**: This turns the rest of the query into a comment, ignoring the password check.

This results in an **unauthorized user gaining access**, because the query returns data for all users, effectively bypassing the login system.

Preventing SQL Injection:

Use Prepared Statements:
This ensures that user inputs are treated as data, not executable code, preventing malicious injection.

Example in Python with `SQLite`:

```
query = "SELECT * FROM users WHERE name = ? AND password = ?"
db.execute(query, (name, password))
```

- **Use ORM Libraries**:
  Object-Relational Mapping (ORM) libraries like SQLAlchemy automatically handle input sanitization, making SQL injections less likely.

Web Application Security

Vulnerabilities

- **SQL Injection**:
  A code injection technique that exploits vulnerabilities in the database layer of an application, allowing attackers to manipulate SQL queries.
- **Buffer Overflow**:
  A vulnerability in low-level code where excess data overflows into adjacent memory, causing crashes and potentially allowing attackers to run malicious code.
- **Input Overflow**:
  Occurs when more data is inputted into a variable than it is intended to hold, potentially leading to unexpected behaviors or vulnerabilities.
- **Cross-Site Scripting (XSS)**:
  A vulnerability that allows attackers to inject malicious code into web applications, affecting users when they interact with the site.

Solutions for Web Application Security

- **HTTPS**:
  - **Purpose**: Encrypts data transferred between the client and server.
  - **Protocol**: Uses **SSL** and **TLS** for secure data transmission.
  - **Limitations**:
    - Only secures the link, not the data itself.
    - Doesn’t validate or check the data for integrity.
    - May impact caching and cause slight overhead.
- **Data Encryption**:
  - Encrypt sensitive data, like passwords or credit card numbers, before storing or transmitting to prevent unauthorized access.
- **Cross-Site Request Forgery (CSRF) Tokens**:
  - **Purpose**: Ensures that requests made by users are legitimate and not from attackers.
  - **Method**: Generate unique, unpredictable tokens on the server-side and include them in HTTP requests to prevent unauthorized actions.
- **Input Validation**:

    - Implement strict validation on server-side to sanitize user inputs, preventing injection of malicious code.

Example (Flask):

```
from flask import Flask, request

app = Flask(__name__)

@app.route('/login', methods=['POST'])

def login():

    username = request.form.get('username')

    password = request.form.get('password')

    # Validate input

    if not username or not password:

        return 'Invalid input', 400

    # Authenticate user

    if username == 'admin' and password == 'password':

        return 'Login successful', 200

    else:

         return 'Invalid credentials', 401
```

IMPORTANT FORMULAS:

Bandwidth=Bus Width×Clock Speed×Data Rate Multiplier

PA:

Q1:You have a DRAM module with bus width of 64 bits, clock speed of 1 GHz, and operating in DDR (double-data-rate or two values per clock cycle) mode. What is the maximum bandwidth (in Giga-bytes per second) of data transfer achievable with this module?

ANS: **Bandwidth=Bus Width×Clock Speed×Data Rate Multiplier**

Where:

- **Bus Width** is 64 bits (8 bytes since 1 byte = 8 bits),
- **Clock Speed** is 1 GHz (1 billion cycles per second),
- **Data Rate Multiplier** is 2 for DDR (Double Data Rate), meaning data is transferred twice per clock cycle.

Bandwidth=64 bits×1 GHz×2 = 128 GB/s

Converting from bits to bytes (since 1 byte = 8 bits):

Bandwidth=128 Gbps/8=16 GB/s

Q2:A magnetic disk operating at 7200 rpm is being used to store data. The disk can only spin in one direction at a constant speed of 7200 revolutions per minute. If the operating system sends a request to the disk controller to fetch data from the disk, what is the worst case latency (in msec) before it can start retrieving data?

ANS: Convert RPM to Rotational Latency

First, we need to determine how long it takes for one full revolution of the disk:

- 7200 RPM means the disk completes 7200 revolutions in one minute.
- To convert that to revolutions per second (RPS), divide by 60: 7200 RPM÷60=120 RPS

Time for One Full Revolution

The time for one full revolution (in seconds) is the inverse of the revolutions per second:

Time per revolution=1/120 seconds=0.00833 seconds

Now, convert this to milliseconds (since 1 second = 1000 milliseconds):

0.00833 seconds×1000=8.33 milliseconds

Q3:You have two search functions, one of which takes time t1(N) = 1000N+200 nanoseconds, while the other takes t2(N) = N^3 + 5 nanoseconds to operate on an input of size N. Which function is better if you are sure that the input size N is always less than 10?

N<10

ANS: **For t1(N) = 1000N + 200:**

- When N = 1, t1(1) = 1000(1) + 200 = 1200 nanoseconds
- When N = 2, t1(2) = 1000(2) + 200 = 2200 nanoseconds
- When N = 3, t1(3) = 1000(3) + 200 = 3200 nanoseconds
- When N = 4, t1(4) = 1000(4) + 200 = 4200 nanoseconds
- When N = 5, t1(5) = 1000(5) + 200 = 5200 nanoseconds
- When N = 6, t1(6) = 1000(6) + 200 = 6200 nanoseconds
- When N = 7, t1(7) = 1000(7) + 200 = 7200 nanoseconds
- When N = 8, t1(8) = 1000(8) + 200 = 8200 nanoseconds
- When N = 9, t1(9) = 1000(9) + 200 = 9200 nanoseconds

For t2(N) = N^3 + 5:

- When N = 1, t2(1) = 1^3 + 5 = 6 nanoseconds
- When N = 2, t2(2) = 2^3 + 5 = 13 nanoseconds
- When N = 3, t2(3) = 3^3 + 5 = 32 nanoseconds
- When N = 4, t2(4) = 4^3 + 5 = 69 nanoseconds
- When N = 5, t2(5) = 5^3 + 5 = 130 nanoseconds
- When N = 6, t2(6) = 6^3 + 5 = 221 nanoseconds
- When N = 7, t2(7) = 7^3 + 5 = 354 nanoseconds
- When N = 8, t2(8) = 8^3 + 5 = 517 nanoseconds
- When N = 9, t2(9) = 9^3 + 5 = 734 nanoseconds

N	**t1(N)**	**t2(N)**
1	1200	6
2	2200	13

3	3200	32
4	4200	69
5	5200	130
6	6200	221
7	7200	354
8	8200	517
9	9200	734

Q9: You have a datacenter where each server has a 1 gigabit per second network connection, but the overall datacenter has a 10 gigabit per second connection. As your app becomes more popular, you start getting 10,000 requests per second, and each request needs a 50 Kbyte response. Which of the following is a better option?

scale up on a single server
scale out to multiple servers in the datacenter

50KbyteX10000=5,00,000 Kbyte/sec (40,00,000)Kb=4Gb
Our app:1gb/s=
Overall datacenter:10Gb/s

In this scenario, **scaling out to multiple servers in the datacenter** is the better option.

Here's why:

1. **Server's Bandwidth Limitation:**
    - Each server has a 1 gigabit per second (Gbps) network connection. Since 1 Gbps equals 125 megabytes per second (MB/s), this means each server can handle 125 MB per second of traffic. However, you need to assess how much traffic each request will generate and how much total traffic your application will require.
2. **Traffic Generated by Requests:**
    - Each request requires a 50 Kbyte response. For 10,000 requests per second, this means: 10,000 requests/sec×50 Kbytes=500,000 Kbytes/sec=500 MB/sec.10,000 \text{ requests/sec} \times 50 \text{ Kbytes} = 500,000 \text{ Kbytes/sec} = 500 \text{ MB/sec}.10,000 requests/sec×50 Kbytes=500,000 Kbytes/sec=500 MB/sec.

    - So, you need to handle 500 MB/sec of traffic.
3. **Total Bandwidth Requirements:**
    - Your datacenter's total bandwidth is 10 Gbps, or 1,250 MB/sec. Since your total traffic requirement (500 MB/sec) is much smaller than the datacenter's total bandwidth, the limitation is not on the overall datacenter bandwidth, but on the bandwidth available to each server.
4. **Single Server Limitation:**
    - A single server can only handle 125 MB/sec, which is much lower than the 500 MB/sec your app requires. This means a single server will be overwhelmed by the incoming traffic.
5. **Scaling Out to Multiple Servers:**
    - By scaling out to multiple servers, you can distribute the load. If you deploy multiple servers, each server can handle part of the total traffic (up to 125 MB/sec per server). For example, you would need at least **4 servers** (since 500÷125=4500 \div 125 = 4500÷125=4) to handle the required 500 MB/sec of traffic.
6. **Network Bottleneck Considerations:**
    - With multiple servers, each server can independently serve a portion of the requests, reducing the risk of overloading any single server's network connection and utilizing the datacenter's total available bandwidth more efficiently.

Conclusion:

Scaling out is the better choice because a single server cannot handle the total bandwidth required, but multiple servers can handle the load more efficiently, leveraging the datacenter's full capacity.