Week 7: Backend Systems

L7.1: *Memory Hierarchy*

Backend Systems

Types of Storage elements

- On-chip registers: 10s to 100s of bytes
- SRAM (cache): 0.1s to 1MB
- DRAM (main memory): 10s to 100s of GB
- SSD (solid state drive) - Flash: 100GB - 1TB
- Magnetic disck (hard drive): 1TB,...
- There are various other types of storage elements like optical, magnetic tape, holographic, etc...
  Note: Above numbers are just example, there's no limit.

Storage Parameters

- **Latency**
  - It refers to the time it takes for a data packet to travel from the source to the destination (here storage element).
  - It is often measured in milliseconds (ms) and represents the delay between the initiation of an action and the response or result.
  - Lower latency is desirable, especially in real-time applications like online gaming, video conferencing, and financial transactions, where quick responses are crucial.
  - `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.
  - It is often measured in megabits per second (Mbps) or gigabits per second (Gbps).
  - Higher throughput is desirable, especially in applications like video streaming, where a large amount of data is transferred continuously.
  - `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 and increased scalability.
  - `HDD > SSD > DRAM > SRAM > Registers`

Computer Architecture

It deals with how a computer system organizes its various memory components.
This hierarchy consists of different memory levels, each with distinct properties in terms of speed, capacity, and cost.
The goal of the memory hierarchy is to effectively handle data access and storage, ultimately improving the overall performance of the computer system.

- Memory hierarchy organizes memory into multiple levels based on proximity to the CPU and performance characteristics.
- Levels include blazing-fast CPU Registers and on-chip L1, L2, and L3 Caches for quick data access.
- Main Memory (RAM) is larger but slower, while Secondary Storage (HDD/SSD) offers high-capacity but even slower storage.
- The hierarchy optimizes data access by moving frequently used data closer to the CPU, enhancing overall performance.

Cold Storage

- Cold storage is a computer system designed for retaining inactive data, such as information required for regulatory compliance, at low cost and high efficiency.
- Examples: Amazon Glacier, Google Coldline, Microsoft Azure Cool Blob Storage, etc...

Impact on application development

- Plan the storage needs based on application growth
- Speed of an app determined by types of data stored, how stored, and how accessed
- Some data stores are more efficients for some types of read/write operations.

> Developer must be aware of choices and what kind of database to choose for a given application.

L7.2: *Data Search*

Refer to these PDSA notes:

$O()$ Notation 🔗

Searching for an element in memory 🔗

L7.3: *Database Search*

Tabular databases

- Tables with rows and columns
- To search quickly on some columns, we can create **INDEX** of those columns.
- Index are stored in a tree structure, so searching is fast.
- Example: MySQL use B-Tree and Hash Indexes. 🔗

B-Tree

A B-tree index can be used for column comparisons in expressions that use the =, >, >=, <, <=, or BETWEEN operators.
The index also can be used for LIKE comparisons if the argument to LIKE is a constant string that does not start with a wildcard character.
For example, the following SELECT statements use indexes:

```
SELECT * FROM tbl_name WHERE key_col LIKE 'Patrick%';
SELECT * FROM tbl_name WHERE key_col LIKE 'Pat%_ck%';
```

- In the first statement, only rows with `Patrick` <= key_col < `Patricl` are considered. In the second statement, only rows with `Pat` <= key_col < `Pau` are considered.
- For more, check here 🔗

Hash Index

Hash indexes have somewhat different characteristics from those just discussed:

- They are used only for equality comparisons that use the = or <=> operators (but are very fast). They are not used for comparison operators such as < that find a range of values. Systems that rely on this type of single-value lookup are known as "key-value stores"; to use MySQL for such applications, use hash indexes wherever possible.
- The optimizer cannot use a hash index to speed up ORDER BY operations. (This type of index cannot be used to search for the next entry in order.)
- For more, check here 🔗

Query Optimization

- Query optimization is database specific.
- MySQL, SQLite, PostgreSQL, etc... have different query optimization techniques.
- MySQL 🔗
- SQLite 🔗
- PostgreSQL 🔗

L7.4: *SQL vs NoSQL*

SQL vs NoSQL

Parameter	SQL (using MySQL)	NoSQL (using MongoDB)
Data Model	Follows fixed schema with tables and columns.	Adapts to flexible schema (key-value, document, column, graph).
Query Language	Uses standardized SQL for querying (`SELECT`, `INSERT`, etc.).	Has query languages tailored to specific data models.
Scalability	Traditionally vertically scaled, can be challenging for horizontal scaling.	Designed for efficient horizontal scaling and distributed systems.
Consistency	Provides strong data consistency guarantees.	Prioritizes availability and partition tolerance over strong consistency.
Examples	MySQL (using Python and SQLite)	MongoDB (using Python and MongoDB)
Use Cases	Suitable for applications with strict data consistency and complex transactions.	Ideal for large-scale systems, unstructured data, real-time data analysis.

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;
```

NoSQL Query example:

```
// Find 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 way to store data:

Key-Value

- Key-values are stored in a hash table or search trees.
- Example: Redis, DynamoDB, Python Dicitonary, C++ OrderedMap etc...
- Very efficient key lookup, not good for range type queries.
- Often used for caching, session management, etc...

Column Stores

- Traditional databases store data in rows, but column stores store data in columns.
- Example: Cassandra, HBase, etc...

Graph

- Graph databases store data in nodes and edges.
- Different degrees, weights of edges, nodes etc... are ways to store data.
- Path finding more important than just search.
- Example: Neo4j, OrientDB, etc...

Time Series Databases

- Time series databases store data with timestamps.
- Used for log analysis, performance monitoring, etc...
- Queries example:
    - How many hits between T1 and T2 ?
    - Average hits per day ?
    - Country with most hits ?
- Example: RRD Tool, InfluxDB, Prometheus, etc...
- Search: elastic search, solr, grafana etc...

ACID

- ACID stands for **Atomicity, Consistency, Isolation, Durability**.
- It is a set of properties of database transactions
- Read more about it here 🔗 (page no. 15)

L7.5: *Scaling*

Replication and Redundancy

Parameter	Replication	Redundancy

Parameter	Replication	Redundancy
Definition	Replication involves creating and maintaining multiple copies of data on different servers or storage devices. Each copy is synchronized to ensure consistency and availability.	Redundancy refers to the practice of creating duplicate or additional components, such as servers or storage devices, to provide backup or failover in case of hardware failures or data loss.
Goal	To ensure that data is always available, even if one or more copies of the data fail.	To improve the reliability of a system by providing multiple copies of components or services.
Implementation	Can be implemented in a variety of ways, such as through mirroring, sharding, or erasure coding.	Can be implemented by duplicating components or services, such as servers, disks, or network links.
Benefits	Can improve availability, performance, and scalability.	Can improve reliability, fault tolerance, and disaster recovery.
Drawbacks	Can be complex to implement and manage.	Can increase costs and complexity.
Use cases	Often used in mission-critical applications, such as financial trading systems and online stores. It is commonly used in distributed systems, cloud environments, and data centers to improve data access speed and resilience against server failures.	Often used in applications that require high availability, such as web servers and database servers. It is applied in critical systems, high-availability setups, and mission-critical applications where downtime must be minimized to prevent data loss or service disruptions.

`ACID` vs `BASE`

Feature	`ACID`	`BASE`
Definition	`ACID` stands for Atomicity, Consistency, Isolation, and Durability.	BASE stands for Basically Available, Soft State, Eventual Consistency.
Guarantees	ACID guarantees that transactions are always atomic, consistent, isolated, and durable.	BASE guarantees that data is eventually consistent, even if there are temporary inconsistencies.
Implementation	ACID is typically implemented using traditional relational databases.	BASE is typically implemented using NoSQL databases.
Benefits	ACID provides strong guarantees for data integrity. BASE provides good availability and scalability.	

Feature	`ACID`	`BASE`
Drawbacks	ACID can be complex to implement and manage.	BASE can lead to temporary inconsistencies.
Use cases	ACID is well-suited for transactional applications that require strict data consistency.	BASE is well-suited for big data applications that require flexibility and scalability.
Examples	Examples of databases adhering to ACID principles include Oracle, MySQL, PostgreSQL, and SQL Server.	Examples of databases using BASE principles include Cassandra, Couchbase, MongoDB, and Riak.

Scale-up vs Scale-out

Approach	Scale-Up	Scale-Out
Definition	Increase capacity of a single server	Add more servers or resources
Resource Focus	Vertical growth	Horizontal growth
Advantages	Simpler management, limited scalability	Improved performance, higher scalability
Limitations	Hardware constraints, single points of failure	Complex management, specialized strategies
Use Cases	Moderate workloads, small-medium businesses	Large-scale systems, high-traffic websites
Examples	Upgrade CPU/RAM in a server	Use load balancer, sharding in distributed DB

L7.6: *Security of Databases*

SQL in context of an application

A typical HTML form

```html
<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)
```

If user enters a normal username and password, the final query will be:

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

It is a valid query and will return the user's data. But if the user enters some suspicious data, which makes our query like this:

```
SELECT * FROM users WHERE name = "" or "" AND password = "" or "";
```

This query will return all the data from the database, which is not what we want. This is called **SQL Injection**.

What about this input:

```
username = x; DROP TABLE users;
password = x
```

The final query will be:

```
SELECT * FROM users WHERE name = 'x'; DROP TABLE users; AND password = 'x';
```

This will delete the entire table from the database.

A real life *SQL Injection* performed on CCTV cameras on parking lots in Australia

Problem:

- Parameters are not sanitized, just taken as it is.

- The query is not checked for any malicious code.
- Validation must be done just before the database query, even if we have validation on frontend, because the user can bypass it.

Web Application Security

SQL Injection

- SQL injection is a code injection technique that exploits a security vulnerability occurring in the database layer of an application.

Buffer Overflow

- Buffer overflow is a vulnerability in low level codes of a program.
- It occurs when a program tries to store more data in a buffer than it was intended to hold.
- This extra data overflows into adjacent memory space, overwriting the data held in that space.
- This can cause the program to crash, and can be exploited by attackers to execute malicious code.

Input Overflow

- Input overflow is a vulnerability in high level codes of a program.
- It occurs when a program tries to store more data in a variable than it was intended to hold.

Cross-Site Scripting (XSS)

- Cross-site scripting is a vulnerability that allows attackers to inject malicious code into a web application.

There are many other vulnerabilities as well, the main point is how to prevent them.

Solutions

HTTPS

- HTTPS is a protocol that encrypts the data sent between a client and a server.
- Secure Sockets Layer (SSL) and Transport Layer Security (TLS) are the two main protocols used to implement HTTPS.
- Server certificates are used to verify the identity of a server.
- However:
  - Only secures the link for data transfer, not the data itself.
  - Doesn't perform any validation or safety checks on the data.
  - Negative impact on "caching" of resources like static files.
  - Some overhead on the server and client.

Data encryption

- This involves encrypting sensitive data, such as passwords and credit card numbers, before it is stored or transmitted over a network.

- This helps to protect the data from being intercepted by attackers.

SQL Injection Prevention

- Use parameterized queries or prepared statements in database interactions to prevent SQL injection attacks.
- Example of a parameterized query:

```sql
SELECT * FROM users WHERE name = ? AND password = ?
```

```sql
SELECT * FROM users WHERE name = :name AND password = :password
```

  - Both of these methods use placeholders to represent the values that will be inserted into the query.

Cross-Site Request Forgery (CSRF) Tokens

- A CSRF token is a unique, secret, unpredictable value that is generated by the server-side application and transmitted to the client in such a way that it is included in a subsequent HTTP request made by the client.
- Implement CSRF tokens to validate and ensure that legitimate requests are originating from the expected users.

Input Validation

- Implement strict input validation on the server-side to validate and sanitize user inputs, preventing malicious code injection and data manipulation.
- Example:

```python
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
```

Screencast

7.1 *Logging*

7.2 *Debugging*