SYSTEM DESIGN TOPICS PART 1

Introduction to System Design

System Design is the process of **defining the architecture, components, modules, interfaces, and data flow** of a software or hardware system to meet specific requirements. It's like planning a building before constructing it — you design how everything will work together before writing code.

Why is System Design important?

- Ensures the system is **scalable**, **reliable**, and **efficient**
- Helps teams understand how all components fit together
- Reduces risk of failures and rework later
- Crucial for building **large-scale applications** (like Amazon, Netflix, WhatsApp)

Types of System Design

1. **High-Level Design (HLD)**
    - Also called **architectural design**
    - Focuses on **overall system architecture**
    - Includes major components like databases, APIs, services, caching, etc.
2. **Low-Level Design (LLD)**
    - Deals with **internal logic of individual components**
    - Includes class diagrams, flowcharts, method-level logic

Core Concepts in System Design

Concept	Description
Scalability	Can the system handle more users or data when needed?
Load Balancing	Distributes traffic across servers to avoid overload
Caching	Temporarily stores data to reduce load and latency
Database Sharding	Splits database into smaller parts for performance

Data Consistency	Ensures all users see the same data (eventually or immediately)
Latency & Throughput	Speed of response and amount of work the system can handle
Fault Tolerance	System continues to work even if some parts fail
API Design	Interfaces for communication between services

Example

Designing "PhotoVerse" – A Complete System Design Walkthrough

Anshika and her team had an idea: to build **PhotoVerse**, a simple app where users can share photos, like others' posts, and scroll through a feed. They built an initial version using basic tools — one backend, one database, and everything tightly connected.

It worked smoothly in the beginning, but as more users joined, things started breaking:

- The app got slower
- Photos took time to load
- Some likes weren't saved
- Sometimes the app crashed when many users were active

They realized it was no longer just a coding issue — they needed **System Design**.

1. High-Level Design – Building the Big Picture

First, they stepped back and thought about the entire structure. What are the main parts of the system?

They split it into **different services**:

- A **User Service** to handle logins, signups, and profiles
- A **Photo Service** to manage uploads and storage
- A **Feed Service** to generate the user's photo timeline
- A **Notification Service** to send alerts for likes and comments

They also decided to store photo files separately in a **cloud storage system** (like a photo locker) and deliver them using a **Content Delivery Network (CDN)** so photos load quickly for users anywhere in the world.

To keep things fast, they added a **Redis cache** to store frequently accessed data like profile info or recent feeds. And to prevent the app from breaking under heavy traffic, they introduced a **Load Balancer**, which spread incoming traffic across multiple servers.

Also, to handle tasks like sending notifications without slowing the main app, they used a **Message Queue**. This way, the app could respond quickly to the user and do heavy tasks in the background.

So now, they had a well-structured system — with services, storage, caching, and background processing — all working together.

2. Low-Level Design – Going Deeper Inside Each Component

Now that the big picture was clear, it was time to dive into the details — how each part would actually work internally.

They started with the **User Service**: it would store details like username, email, password (safely encrypted), and profile image. It had simple functions like registering a new user, logging in, and updating the profile.

Then they focused on the **Photo Service**: it would first validate the photo, then send it to cloud storage, and finally store the photo's link and caption in the main database. To avoid delays, uploading a photo didn't immediately send notifications — instead, a message was sent to a background queue, and another service would handle that later.

Next came the **Feed Service**: this was more complex. It had to collect the latest photos from people the user followed, sort them by time or popularity, and show them fast. To avoid hitting the database every time, they used caching — so once a feed was generated, it was saved for a short time and reused.

They also designed how the **Notifications** would work. When a user liked a photo or left a comment, the app would create a notification — but it would not send it immediately. Instead, it would be added to a queue, and another service would pick it up and deliver the notification (as a popup or message) without slowing anything else.

They decided what information to store — which pieces of data would go in the database, how they would connect (like users to their photos), and how to keep everything fast and clean. They also planned how to ensure data is not lost, how to retry failed operations, and how to handle small errors without affecting the whole system.

3. Scalability and Reliability – Making it Strong and Future-Proof

They knew that if PhotoVerse became popular, the system had to grow — maybe to millions of users. So they thought long-term:

- They made sure more servers could be added anytime, and the load balancer would automatically divide traffic among them.
- They planned for the database to be split when it grew too large — so users' data would be divided across different storage areas.

- For reliability, they made sure if one service failed, others would still keep working.
- They used retries and fallback systems to handle temporary errors.
- Logs and monitoring were set up so they'd be alerted if anything unusual happened.

he Result

Now, PhotoVerse wasn't just an app — it was a **well-designed system**.

- It could handle heavy traffic smoothly
- It responded quickly even with slow networks
- It recovered gracefully from failures
- It was easy to add new features — like stories or reels — in the future

The Lesson:

System Design is not about writing code — it's about planning how every part of your system will work together efficiently, grow easily, and never fall apart under pressure.

It starts with a broad view (High-Level), then dives into logic and structure (Low-Level), and finishes with scalability, performance, and fault-tolerance thinking.

EOT

SOME BASIC TERMINOLOGIES, Vertical & Horizontal Scaling

1. Vertical Scaling : Vertical scaling, also known as scaling up, involves increasing the capacity of a single machine by adding more resources such as memory, storage, or processing power. This approach enhances the throughput of the system without adding new resources, making the existing resources more efficient.

2. Horizontal scaling, also known as scaling out, involves adding more instances of the same type to the existing pool of resources to handle increased traffic and load. Unlike vertical scaling, which increases the capacity of a single machine, horizontal scaling distributes the load across multiple machines, enhancing the overall system performance and reliability.

3. "master-slave architecture" involves a single central unit, referred to as the "master," that governs and guides the activities of several slaves, or subordinate units. The master node in this configuration controls and assigns tasks to the slave nodes, who carry them out and report back to the master. This architecture is commonly used in distributed systems to manage resources efficiently and streamline data processing.

4. A distributed system is a collection of independent computers that appear to the users as a single coherent system. These computers, or nodes, work together, communicate over a network, and coordinate their activities to achieve a common goal by sharing resources, data, and tasks.

5. Microservices Architecture is a method of software development where we break down an application into small, independent, and loosely coupled services. These services are developed, deployed, and maintained by a small team of developers. These services have a separate codebase that is managed by a small team of developers.

6. Load balancing is a technique used to distribute network traffic across multiple servers to ensure high availability, efficient utilization of resources, and improved performance. It acts

as a "traffic cop" that sits in front of your servers and directs client requests across all servers, ensuring no single server is overwhelmed

7. Decoupling (separating out responsibilities) In computing, the term decoupled architecture describes a processor in a computer program that uses a buffer to separate the fetch and decode stages from the execution stage. A decoupled architecture allows each component to perform its tasks independently of the others, while also enabling structural variations between source and target.

8. Logs are a historical record of the various events that occur within a software application, system, or network. They are chronologically ordered so that they can provide a comprehensive timeline of activities, errors, and incidents, enabling you to understand what happened, when it happened, and, often, why it happened.

9. Metrics focus on aggregating numerical data over time from various events, intentionally omitting detailed context to maintain manageable levels of resource consumption. For instance, metrics can include the following:

Number of HTTP requests processed.

The total time spent processing requests.

The number of requests being handled concurrently.

The number of errors encountered.

CPU, memory, and disk usage.

10. Extensibility is a design principle in software development that allows for the addition of new functionality or modification of existing features without altering the system's core code. It's about creating flexible, modular systems that can adapt to changing requirements over time.

11. High-Level Design documents are like big-picture plans that help project managers and architects understand how a system will work and low-Level Design documents are more detailed and are made for programmers.

Vertical Scaling

It is defined as the process of increasing the capacity of a single machine by adding more resources such as memory, storage, etc. to increase the throughput of the system. No new resource is added, rather the capability of the existing resources is made more efficient. This is called Vertical scaling. Vertical Scaling is also called the Scale-up approach.

Example: MySQL

Advantages of Vertical Scaling

It is easy to implement

Reduced software costs as no new resources are added

Fewer efforts required to maintain this single system

Disadvantages of Vertical Scaling

Single-point failure

Since when the system (server) fails, the downtime is high because we only have a single server

High risk of hardware failures

A Real-time Example of Vertical Scaling

When traffic increases, the server degrades in performance. The first possible solution that everyone has is to increase the power of their system. For instance, if earlier they used 8 GB RAM and 128 GB hard drive now with increasing traffic, the power of the system is affected. So a possible solution is to increase the existing RAM or hard drive storage, for e.g. the resources could be increased to 16 GB of RAM and 500 GB of a hard drive but this is not an ultimate solution as after a point of time, these capacities will reach a saturation point.

Horizontal Scaling

It is defined as the process of adding more instances of the same type to the existing pool of resources and not increasing the capacity of existing resources like in vertical scaling. This

kind of scaling also helps in decreasing the load on the server. This is called Horizontal Scaling

Horizontal Scaling is also called the Scale-out approach.

In this process, the number of servers is increased and not the individual capacity of the server. This is done with the help of a Load Balancer which basically routes the user requests to different servers according to the availability of the server. Thereby, increasing the overall performance of the system. In this way, the entire process is distributed among all servers rather than just depending on a single server.

Example: NoSQL, Cassandra, and MongoDB

Advantages of Horizontal Scaling

Fault Tolerance means that there is no single point of failure in this kind of scale because there are 5 servers here instead of 1 powerful server. So if any of the servers fail, then there will be other servers for backup. Whereas, in Vertical Scaling, there is the single point of failure i.e, if a server fails, then the whole service is stopped.

Low Latency: Latency refers to how late or delayed our request is processed.

Built-in backup

Disadvantages of Horizontal Scaling

Not easy to implement as there are several components in this kind of scale

The cost is high

Networking components like, router, load balancer are required

A Real-time Example of Horizontal Scaling

For example, if there exists a system of the capacity of 8 GB of RAM and in future, there is a requirement of 16 GB of RAM then, rather than the increasing capacity of 8 GB RAM to 16 GB of RAM, similar instances of 8 GB RAM could be used to meet the requirements.

IN REAL LIFE

WE TAKE QUALITIES FROM VERTICAL SCALING AS

1. ITS CONSISTENCY

2. INTER-PROCESS COMMUNICATION

AND FROM HORIZONTAL SCALING

1. SCALES

2. RESILIENT (NOT A SINGLE POINT SO IF FAILURE OCCURS IT WILL MANAGE)

If each request requires a lot of processing, then certain scenarios require vertical scaling. If the System is facing request drops at load balancer due to too much traffic, then horizontal scaling is the solution.

Feature / Point	**Vertical Scaling (Scale Up)**	**Horizontal Scaling (Scale Out)**
Kya hota hai?	Ek hi server ko powerful banana	Zyada servers add karna
Kaise hota hai?	RAM, CPU, storage badhate ho	Naye servers system mein add karte ho
Example	8GB RAM server ko 16GB RAM kar dena	3 servers ke badle 6 servers use karna
Architecture par impact	System mostly same rehta hai	System architecture ko thoda change karna padta hai
Limitations	Ek limit tak hi scale kar sakte ho (hardware ki limit)	As per need, servers badhate jao (cloud mein easy)
Cost	Powerful machines mehengi hoti hain	Chhoti machines relatively sasti padti hain
Failure ka risk	Ek server down hua to system ruk sakta hai	Ek server fail hua to baaki chal sakte hain
Use kab hota hai?	Jab traffic kam ya medium ho, aur architecture simple chahiye	Jab traffic bahut zyada ho, ya app large scale pe ho
Commonly used in	Traditional apps, small projects	Web-scale systems, cloud-based apps like Google, Facebook

EOT

Capacity Estimation, Network Flow (later modules – discussed in detail)

Capacity estimation in systems design is the process of predicting or determining the maximum load or demand that a system can handle within its operational parameters. This involves analyzing various aspects such as hardware capabilities, software performance, network bandwidth, and user behavior patterns.

The goal is to ensure that the system can accommodate the expected workload without experiencing performance degradation, bottlenecks, or failures.

Capacity estimation is crucial for designing and scaling systems effectively to meet current and future demands, whether it's a website, a network infrastructure, or any other complex system.

Network Layer Flow

The user types a message or opens a website

➤ Ye hota hai Application Layer mein.

➤ Yahan protocols hote hain jaise:

HTTP (web ke liye),

SMTP (email ke liye),

FTP (file transfer ke liye).

Iske baad →

Transport Layer data ko chhoti-chhoti units mein todta hai (segments)

➤ TCP ya UDP use hota hai:

TCP: Reliable, slow but safe (delivery confirm hoti hai).

UDP: Fast but risky (delivery confirm nahi hoti).

Iske baad →

Network Layer har segment ko ek IP address assign karti hai

➤ Yahan se packet ban jaata hai.

➤ IP address se pata chalta hai ki data kis device pe jaana hai.

➤ Routing bhi yahi layer karti hai — best path decide karti hai.

Iske baad →

Data Link Layer IP packet ko frame bana deti hai

➤ Frame ke andar hota hai:

Header: Source & destination ke MAC addresses

Trailer: Error-checking bits

➤ Ye layer ensure karti hai ki local network par data thik se deliver ho.

Iske baad →

Physical Layer frame ko actual signal mein convert karti hai

➤ Signals ho sakte hain:

Electric (copper wire ke liye)

Light (fiber optic ke liye)

Radio waves (Wi-Fi ke liye)

Ye signals medium ke through bheje jaate hain.

Midway – Routers (beech mein routers kya karte hain?)

Har router:

Frame ka MAC header hata deta hai

IP packet ko padhta hai

Next best route choose karta hai

Naya frame banata hai with next-hop MAC address

RECEIVER SIDE PE DATA KA UNPACKING START

Receiver Physical Layer signal ko receive karta hai

➤ Signal wapas frame mein convert hota hai.

Iske baad →

Data Link Layer frame ka MAC address check karti hai

➤ Agar match hota hai, toh error-checking hoti hai

➤ Trailer se errors detect kiye jaate hain

Iske baad →

Network Layer IP packet ko read karti hai

➤ IP address check hota hai (yehi device ke liye hai ya nahi)

➤ Fragmented packets ko jodti bhi hai agar zarurat ho

Iske baad →

Transport Layer segments ko wapas jodti hai

➤ Sequence numbers use karke sahi order mein arrange karti hai

➤ Agar TCP hai toh delivery confirm bhi hoti hai

Iske baad →

Application Layer data ko show karta hai

➤ Browser kholta hai webpage

➤ Email app message show karta hai

➤ User ko final output milta hai

EOT

What happens when you enter “XYZ.com”?

- **What is a WebPage** -> Before getting started, I want to first explain what a webpage is. A webpage is basically a text file formatted a certain way so that your browser (ie. Chrome, Firefox, Safari, etc) can understand it; this format is called HyperText Markup Language (HTML). These files are located in computers that provide the service of storing said files and waiting for someone to need them to deliver them. They are called servers because they serve the content that they hold to whoever needs it.
- **Servers** -> These servers can vary in classes, the most common and the one that we'll be talking about in the main portion of this article is a web server, the one that serves web pages. We can also find application servers, which are the ones that hold an application base code that will then be used to interact with a web browser or other applications. Database servers are also out there, which are the ones that hold a database that can be updated and consulted when needed.
- **IP Addresses** -> These servers in order to deliver their content, much like in physical courier services, need to have an address so that the person needing to be said content can make a "letter" requesting the delivery; the person requesting the content in turn also has an address where the server can deliver the content to. These addresses are called IP (Internet Protocol) Addresses, a set of 4 numbers that range from 0 to 255 (one byte) separated by periods (ie. 127.0.0.1).
- **Protocols for Delivery** -> Another concept that is important to know is that the courier service traffic for the delivery can be one of two: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). Each one determines the way the content of a server is served or delivered.

TCP -> TCP is usually used to deliver static websites such as Wikipedia or Google and also email services and to download files to your computer because TCP makes sure that all the content that is needed gets delivered. It accomplishes this by sending the file in small packets of data and along with each packet a confirmation to know that the packet was delivered; that's why if you are ever downloading something and your internet connection suddenly drops when it comes back up you don't have to start over because the server would know exactly how many packets you have and how many you still need to receive. The downside to TCP is that because it has to confirm whether you got the packet or not before sending the next, it tends to be slower.

UDP-> UDP, on the other hand, is usually used to serve live videos or online games. This is because UDP is a lot faster than TCP since UDP does not check if the information was

received or not; it is not important. The only thing UDP cares about is sending the information. That is the reason why if you've ever watched a live video and if either your internet connection or the host's drops, you would just stop seeing the content; and when the connection comes back up you will only see the current stream of the broadcast and what was missed is forever lost. This is also true for online videogames (if you've played them you know exactly what this means)

How a Website Loads –

Step 1: You type a website (like `www.amazon.com`) in the browser

It's like saying:

"I want a book from Amazon – Do I already know where it is?"

Step 2: The Browser checks its memory (cache)

Browser thinks:

"Have I visited this website before? Do I already know its address?"

If yes → Use that saved address (IP) directly

If no → Ask the operating system (Windows, Linux, etc.) for help

Step 3: OS checks the hosts file

Like checking a personal notebook:

"Is this website's address written here?"

If yes → Use that IP address

If no → Let's ask the Internet (via DNS)

Step 4: DNS (like a phonebook) is used to find the website's address

This is like calling different helpers to find the book's location:

1. First ask the local resolver (your internet provider)

➤ "Do you know where amazon.com is?"

2. If not, ask the root server

➤ “Where can I find a `.com` address?”

3. Then ask the `.COM server`

➤ “Do you know who owns amazon.com?”

4. Finally, ask the main name server (the owner of the site)

➤ It says: “Yes! The IP address is 192.0.2.1”

All this happens in milliseconds!

Step 5: Now we have the address (IP), time to send a message

The browser tells the OS:

“Here’s the address. Send a GET request to ask for the website’s content.”

Step 6: OS packs the request using the TCP protocol

Like putting your message in an envelope 📮

TCP ensures:

The message doesn’t break on the way

It goes in the correct order

If lost, it’s sent again

Step 7: Firewalls check your message

Think of a security guard:

“Is this a safe message to send/receive?”

If safe → Pass through

If risky → Block it

Step 8: Message reaches the main server (like a giant book store)

The message usually goes to a load balancer, which:

Distributes traffic

Chooses the best available server to answer

Then, the server sends back:

The IP it chose to respond from

SSL certificate for a secure connection (HTTPS)

Step 9: The Server sends back the website's files

You receive:

HTML (structure of the page)

CSS (styling, colors, fonts)

JavaScript (animations, actions)

Step 10: OS hands over files to the browser

The browser reads the files like a recipe:

Builds the page layout

Adds colors and styles

Adds interactivity

Final Step: The Website appears on your screen!

EOT

Relational Database

A **Relational Database (RDB)** is a type of database that **stores data in tables** (rows and columns), and the data in different tables is **related to each other** using **keys**.

Key Type	Purpose
Primary Key	Uniquely identifies a row (only one, no NULL)
Foreign Key	Links to primary key in another table
Candidate Key	All possible unique identifiers
Alternate Key	Candidate keys not chosen as primary
Super Key	Any column(s) that can uniquely identify rows (even with extras)
Composite Key	Combination of columns used as a unique identifier
Unique Key	Like primary, but allows one NULL and multiple per table

What is Normalization?

Normalization is the process of **organizing data** in a relational database to:

- Eliminate redundancy (duplicate data)
- Ensure data integrity
- Make the database efficient and easy to maintain

It is done in **stages**, called **Normal Forms (NFs)**.

1. First Normal Form (1NF)

- **Rule**: All values in a column must be **atomic** (indivisible).
- **No repeating groups or arrays** in a single column.

Example:
Not 1NF: Subjects = Math, English
1NF: One subject per row

Goal: Break down multi-valued fields into separate rows.

2. Second Normal Form (2NF)

- **Rule**: Must be in 1NF, and
- **No partial dependency** (i.e., non-key columns must depend on the whole primary key)

Example: In a table with composite key (student_id, course_id), if student_name only depends on student_id, not on the whole key — that's a partial dependency → **Break it!**

Goal: Eliminate partial dependencies by separating into multiple tables.

3. Third Normal Form (3NF)

- **Rule**: Must be in 2NF, and
- **No transitive dependency** (i.e., non-key column should not depend on another non-key column)

Example:
If student_id → department_id → department_name, then department_name should be in a separate table.

Goal: Every non-key column should depend **only** on the primary key.

4. Boyce-Codd Normal Form (BCNF)

- A stricter version of 3NF
- **Rule**: For every dependency A → B, A must be a **super key**

Example:
If a non-unique column can determine a unique column, BCNF breaks that.

Goal: Handle cases where **non-prime attributes** are determining keys.

5. Fourth Normal Form (4NF)

- **Rule**: Must be in BCNF, and
- No **multi-valued dependencies**

Example:
If a student has multiple phone numbers **and** multiple hobbies, these should be stored in separate tables — not combined in one.

Goal: Remove independent multivalued data into separate tables.

6. Fifth Normal Form (5NF) or PJ/NF (Project Join Normal Form)

- **Rule**: Must be in 4NF, and
- **Eliminate join dependencies** – avoid splitting/joining data unnecessarily

Use case is rare and very complex.

Goal: Ensure **no loss of data** or repetition when joining tables.

6NF (Sixth Normal Form) – Rarely used

- Breaks data even further for **temporal databases** (time-based records)

SQL vs NoSQL

Feature / Point	**SQL (Relational DB)**	**NoSQL (Non-Relational DB)**
Full Form	Structured Query Language	Not Only SQL
Data Storage	Tables (rows & columns)	Key-Value, Document, Column, Graph
Schema (Structure)	Fixed schema (predefined structure)	Flexible schema (dynamic, no need to define in advance)
Scalability	Vertically scalable (upgrade server)	Horizontally scalable (add more servers)
Best For	Structured data with clear relationships	Unstructured or semi-structured data
Examples	MySQL, PostgreSQL, Oracle, MS SQL Server	MongoDB, Cassandra, Redis, Firebase
Query Language	SQL (standard language for queries)	No standard language (uses JSON-like queries or APIs)
Data Integrity	High (ACID compliant: strong consistency)	Varies (often uses eventual consistency – BASE model)
Relationships	Strong support via foreign keys & joins	Limited or manual relationship handling
Performance (Read/Write)	Read-optimized, structured queries	Write-optimized, fast for big/real-time data
Use Cases	Banking, ERP, eCommerce (structured apps)	Social media, IoT, analytics, real-time apps

Joins	Supports complex joins	Doesn’t support joins (denormalization preferred)
Transactions	Strong multi-record transactions	Limited transactions (some NoSQL DBs support them now)
Flexibility	Less flexible (needs schema migration on change)	Highly flexible (schema-less)
Data Type Support	Mostly simple types (int, varchar, etc.)	Rich types like JSON, arrays, nested docs
Community Support	Mature, large community	Growing rapidly

1. Join is expensive in SQL database, and in NoSQL it's pretty cheaper to store in a single set scheme

2. In a NoSQL database, the schema is easily changeable

3. NoSQL is built for scaling and aggregations

4. if you have lots of updates, NoSQL is not really for it, and also here the ACID property is not guaranteed, toh transaction problems may occur, isliye finance sector m ye kam hi use hota hai

5. Each time we have to go to every block for reading , but in sql we just go to particular table and read

6. agr 2 nosql table join krna hai to it's very hard, all manual joins will be done

toh nosql use kro jb data is a block, updates kam hai, sab kuchh sath m hi rkhna hai data

Types of NoSQL Database

NoSQL databases can be classified into four main types, based on their data storage and retrieval methods:

Document-based databases, Instead of storing the data in rows and columns (tables), it uses the documents to store the data in the database. A document database stores data in JSON, BSON, or XML documents.

Documents can be stored and retrieved in a form that is much closer to the data objects used in applications which means less translation is required to use these data in the applications. In the Document database, the particular elements can be accessed by using the index value that is assigned for faster querying.

Key-value stores, The simplest form of a NoSQL database is a key-value store. Every data element in the database is stored in key-value pairs. The data can be retrieved by using a unique key allotted to each element in the database. The values can be simple data types like strings and numbers or complex objects.

Column-oriented databases, stores the data in columns instead of rows. That means when we want to run analytics on a small number of columns, we can read those columns directly without consuming memory with the unwanted data. Columnar databases are designed to read data more efficiently and retrieve the data with greater speed. A columnar database is used to store a large amount of data.

Graph-based databases, focus on the relationship between the elements. It stores the data in the form of nodes in the database. The connections between the nodes are called links or relationships, making them ideal for complex relationship-based queries.

Cassandra:

Query Language: Cassandra uses CQL (Cassandra Query Language) for querying data. CQL is similar to SQL in syntax but is designed specifically for Cassandra's data model.

Data Model: Cassandra is based on a wide-column store data model, which means that data is stored in rows and columns. It is optimized for write-heavy workloads and is designed to be highly scalable and fault-tolerant.

MongoDB:

Query Language: MongoDB uses a query language that is based on JSON-like documents. Queries are expressed using a rich and flexible syntax that allows for complex queries and aggregations.

Data Model: MongoDB uses a document-oriented approach, storing data as JSON-like documents. It is designed to be flexible and scalable and making it suitable for a wide range of use cases.

DATABASE INDEX

In SQL (normalized models), data is split across tables to avoid redundancy.

But in NoSQL (denormalized models) like DynamoDB, data is often duplicated to optimize for specific query access patterns (because joins are expensive or unavailable).

A database index is like the index in a book — it helps you find data faster without scanning the entire table.

How It Works:

It stores a copy of selected columns (like names, IDs, etc.) in a special data structure (like a B-Tree or Hash).

Along with these values, it stores pointers to the rows in the original table.

When you query the data, the database uses the index to find the matching rows faster.

EOT

Cache, Thrashing and Threads

Cache memory is a small, fast memory that holds copies of recently accessed instructions and data.

It means many instructions for local areas of the program are executed repeatedly. The active segments of the program are placed in cache memory to reduce the total execution time.

If we get the memory reference that is requested we call it “CACHE HIT” otherwise “CACHE MISS”.

The performance of cache memory is calculated in terms of hit ratio.

Hit ratio = hit/(hit + miss) = no. of hits/ total access.

Cache ka basic principle hota hai – Locality of Reference

Iska matlab hai:

Jo data CPU baar-baar ya aas-paas ka data use karta hai, usi ko ek chhoti aur fast memory (cache) mein store kiya jaata hai.

Taaki CPU ko woh data baar-baar door RAM se laane ki zarurat na pade, aur access time kam ho jaaye. Isse performance fast ho jaata hai.

1. Temporal Locality (Time se related)

Matlab: Agar CPU ne abhi koi data access kiya hai, toh chances hain ki wahi data phir se jaldi use karega.

Isiliye uss data ko cache mein rakhte hain.

Example:

Socho tum YouTube dekh rahe ho, ek video baar-baar play kar rahe ho.

System samajh jaata hai ki ye video baar-baar use ho raha hai, toh usse memory mein ready rakh leta hai – taaki agle time pe jaldi chal jaaye.

2. Spatial Locality (Jagah se related)

Matlab: Agar CPU ne ek memory location ka data access kiya hai, toh uske aas-paas ka data bhi jaldi access karega.

Isiliye ek saath aas-paas ka data cache mein le lete hain.

Example:

Socho tum ek notebook padh rahe ho, aur page 25 pe ho. Chances hain ki tum agla page ya usi ke aas-paas ka page padhoge.

Waise hi CPU bhi jab ek location ka data leta hai, toh uske aas-paas ka bhi cache mein la leta hai.

Cache chhoti memory hoti hai, lekin woh 1-1 byte mein organize nahi hoti.

Usko “cache lines” ya “blocks” mein divide kiya jaata hai.

Ek cache line mein 16 se 64 bytes tak data hota hai.

(Socho ek locker jisme ek baar mein 16-64 cheezein rakh sakte ho.)

Cache lines fixed address pe nahi hoti

Matlab:

Cache line RAM ke ek specific fixed address se linked nahi hoti.

Woh RAM ke kisi bhi jagah se data rakh sakti hai.

Isse flexibility milti hai ki jo bhi zarurat ka data ho, usse kisi bhi line mein daal do.

1. Fully Associative (Sabse flexible)

RAM ka koi bhi block cache ke kisi bhi line mein aa sakta hai.

Example: Socho tumhare paas ek almari hai jismein koi bhi saman tum kisi bhi drawer mein rakh sakte ho. Total freedom.

2. Direct Mapped (Sabse basic)

RAM ka ek specific block sirf ek specific cache line mein hi ja sakta hai.

Example: Socho tumhare paas 10 drawers hain, aur har item ka ek fixed drawer hai. Aapko wahi rakhna hai, chahe jagah ho ya na ho.

3. Set-Associative (Mix of both)

Cache ko sets mein divide karte hain. Har set mein kuch lines hoti hain (e.g., 2 ya 4).

Ek RAM block ek specific set mein jaa sakta hai, lekin us set ke kisi bhi line mein rakh sakta hai.

Example: Socho 5 almirah hain, har almirah ke 2 drawer hain. Item ek fixed almirah mein jaayega, par dono drawer mein se kisi mein ja sakta hai.

ab trashing p chlte hai

"Thrashing" ek important concept hai memory management mein, especially when dealing with virtual memory or caching.

Thrashing hota hai jab system baar-baar memory se data ko andar–bahar karta rehta hai lekin actual kaam kuch nahi ho pata.

Yaane System busy hota hai sirf data ko swap karne mein, processing karne mein nahi.

Socho tum ek student ho jiske paas ek small table hai (cache ya RAM jaisi).

Tumhare paas 10 books hain (processes ya data blocks), par table pe ek baar mein sirf 2 hi rakh sakte ho.

Ab agar tum baar-baar alag-alag books ka reference dekhte ho, toh har baar tumhe kisi purani book ko hata ke nayi book lana padta hai.

Result?

Tumhara poora time books laane-le jaane mein chala jaata hai, padhai (actual kaam) bilkul nahi hoti!

Thrashing kab hota hai?

Jab CPU ko baar-baar different memory pages chahiye hote hain

Lekin RAM mein jagah kam hoti hai

System baar-baar swap file (ya virtual memory) se pages ko RAM mein laata hai aur replace karta hai

Is frequent page-in/page-out process se CPU ka zyada time swap mein chala jaata hai, kaam mein nahi

Thrashing: System ke slow hone ka vicious cycle

Starting Point: Low CPU Utilization

System dekhta hai ki CPU kaam kam kar raha hai (idle zyada hai).

Toh scheduler sochta hai: "New processes daalo!"

Degree of multiprogramming badha do (Matlab: Zyada programs ek saath chalne lagte hain)

Socho tumhara data RAM (main memory) mein nahi hai, balki hard disk (ya secondary storage) mein pada hai.

Aur CPU ko ussi waqt woh data chahiye hota hai.

Jab CPU kisi aise data ko access karna chahta hai jo RAM mein nahi hai, toh usko page fault kehte hain.

to mtlb ye ki anshika jb hum multiprogramming badha dete hai bss idle na rhe iske liye to nayi process jyada memory demand krti hai phir page faults aane lgte hai

ye sab kiski wjh se , jb hum CPU utilization badhane ke chakkar mein overloading kar dete hain aur kya .

At this point, we must decrease the degree of multiprogramming in order to increase CPU utilization and stop thrashing.

<u>to bachne k liye au r kuchh chezzein</u>

1. Local Replacement Algorithm

Jab ek process ko extra memory chahiye hoti hai, toh woh dusre process ke frames chheen leta hai.

Isse dusra process bhi thrashing mein chala jaata hai.

Local Replacement kya karta hai???? haa ??

Har process apne hi frames ke andar page replace karega

Matlab agar process A thrashing kar raha hai, toh uska effect B ya C process pe nahi padega

2. Working Set Model

Ye model kaam karta hai Locality of Reference ke concept pe:

Ek time pe process sirf kuch hi pages baar-baar use karta hai - unhi ko memory mein rakhna chahiye.

Parameter: Δ (Delta) - Working Set Window: Δ ek time window hai jisme system dekhta hai ki process ne last kitne pages use kiye.

WSSi (Working Set Size of ith process) har process ka ek working set size hota hai = WSSi

Matlab Us process ko jitne pages abhi actively chahiye, woh.

Agar system har process ko WSSi frames de de, toh har process smoothly chalega

Koi dusre ke frames nahi chheenega → No thrashing

Multiprogramming bhi high rakhi ja sakti hai (zyada processes at once)

Ye ek balance banata hai between performance and memory use.

3. Moving Window Concept

Δ ek sliding/moving window hota hai

Har naye memory access pe Ek new page reference window mein add hoti hai

Aur ek oldest reference nikal jaati hai

Socho jaise tumhare WhatsApp status — latest status add hota hai, purana automatically 24 hrs ke baad chala jaata hai.

Now Threads

A program is just a file that contains a set of instructions written to perform a specific task.

It is inactive – just sitting on your computer's storage (like on a hard disk or SSD).

Example: A .exe file or a Python script.

!!Think of it like a recipe written in a book. It tells you what to do, but it doesn't cook the food by itself.

A process is what happens when you run a program.

It is active and lives in the computer's memory (RAM).

The operating system gives it resources like memory and CPU time so it can do its job.

A process has:

Stack: Temporary memory – for things like function calls, local variables.

Data section: Stores global variables(things used throughout the program).

Heap: Memory for things created while the program is running (like objects or data structures).

Think of a process like cooking a recipe– it's using ingredients, tools, and steps actively.

A thread is the smallest unit of work inside a process.

A process can have multiple threads doing different tasks at the same time.

Each thread has its own:

Program counter (to keep track of what it's doing),

Registers constantly needs to store values temporarily

such as:

The current instruction address (via the program counter),

Values of variables being calculated,

The results of arithmetic/logical operations, etc.

And its own stack (for function calls and local stuff).

But all threads share:

The same code (instructions),

Data (global variables),

And system resources like files.

Think of threads as chefs in the same kitchen (process). Each chef has their own tools (stack), but they share the same recipe, ingredients, and kitchen space.

TYPESSSS of process

I/O-bound Process (Input/Output-bound)

These processes spend more time waiting for input or output operations to complete.

They do very little computation.

Most of their time goes in:

Reading/writing files

Accessing the internet

Communicating with devices like keyboards, printers, or disk.

CPU-bound Process (Processor-bound)

These processes spend most of their time doing heavy computations on the CPU.

They rarely need to read/write from/to external devices.

Examples include:

Complex calculations

Image processing

Data analysis and simulations

EOT

Consistent Hashing and Load Balancing

Consistent Hashing is a smart way to distribute data across multiple servers (or nodes) so that when the number of servers changes (e.g., one is added or removed), it minimizes the amount of data that needs to be moved.

Hashing refers to the process of generating a fixed-size output from an input of variable size using the mathematical formulas known as hash functions. This technique determines an index or location for the storage of an item in a data structure.

- Problem kya hoti hai normal hashing mein?

Maan le tere paas 3 servers hain:

Server A

Server B

Server C

Aur tu data store karne ke liye simple formula use kar rahi hai:

server = hash(data) % number_of_servers

Toh agar hash("video1") = 8 aur total 3 servers hain, toh:

8 % 3 = 2 → data jayega Server C pe

Ab agar ek naya Server D add kar diya (4 servers ho gaye), toh formula ho jaayega:

8 % 4 = 0 → data jayega Server A pe

!!! Problem: Sirf ek server add karne se almost saara data doosre servers mein shift ho jaata hai — jo bohot slow aur inefficient hota hai.

- Consistent Hashing ka solution kya hai?

Isme hum modulo use nahi karte.

Instead:

Step-by-step:

Socho ek circle banaya jisme 0 se 360 degree tak numbers hain (yeh hash values represent karta hai).

Har server ko hash karo aur usse circle pe ek point pe daal do.

Jaise:

Server A → 30°

Server B → 120°

Server C → 270°

Ab har data item ko bhi hash karo aur usse circle pe daal do.

Jaise:

Video1 → 100°

Video2 → 290°

Har data item us server pe jaata hai jo circle pe uske baad aata hai (clockwise direction)

Video1 → 100° → next server clockwise is Server B (120°)

Video2 → 290° → wrap-around → next is Server A (30°)

!! Server add/remove hone pe kya hoga?

Ab maan lo tu Server D add karti hai jo 200° pe aata hai.

Sirf 120° se 200° ke beech ka data shift hoga Server B → Server D.

Baaki sab data wahi ka wahi rahega.

- Sirf thoda data shift hota hai
- Fast & scalable system ban jaata hai

Load balancing ka matlab hota hai — kaam ko barabar baantna between multiple servers taaki koi ek server overload na ho jaaye aur system fast aur reliable bana rahe.

IT/Servers mein kya hota hai?

Jab log ek website (jaise YouTube, Flipkart) kholte hain, toh:

Har request ek server ke paas jaati hai.

Agar ek hi server sabka kaam kare, toh crash ho sakta hai 😵

Isliye ek load balancer hota hai jo:

- Har request ko best server pe forward karta hai
- Monitor karta hai kis server pe kitna load hai
- Taaki sabka experience fast aur smooth ho

Consistent Hashing + Load Balancing?

Yeh dono complement karte hain:

Consistent Hashing: Data smartly distribute karta hai

Load Balancer: User requests smartly distribute karta hai

Jaise:

YouTube ke videos ka data consistent hashing se distributed hota hai

Par jab tu ek video play karti hai, toh load balancer decide karta hai tu kaunse server se video stream karegi

EOT

BLOOM FILTERS

Blooom filter ek data structure hai jo help karta hai batane mein “kya koi cheez set mein ho sakti hai ya nahi”.

mtlb bloom filter kya hai na bss false positive true positive results p based hai 1 is to 99 respectively

mtlb suppose instagram user name hai

to jo username available nahi hai mtlb already takes hai uske liye 100% surity ki vo bolega kuchh aur try kro pr kabhi esa bhi ho skta hai ki

available hai lekin tb bhi unavialable bta de vo

so that's what the point is ki isse koi major frk nahi padta hum kuchh aur try krlenge

but ye approach frk kaha laati hai speed m ye fast outcome deti hai itne saara data hone k baawajood

aur jb 70 to 80 % data tryup check ho jaata hai to we have a function in this called as reset()

Bloom Filters use hashing as an immutable function result, and marking the respective positions in the data structure guarantees that the subsequent search for the exact string will return true.

This data structure has an error rate when returning 'true', and we look into how the number of hash functions affects its performance. In practice, Bloom Filters can be used to check for membership and to avoid 'One Hit Wonders'.

Types of bloom filters :

counting bloom filter :

isme counters add hote hai agr kuchh add hua to +1 aur remove hua to -1 , jese agr kisine apna account permanately dlt krdiya ho kabhi to ye vo username provide kr skta hai na

scalable bloom filter :

standard waala fixed size ka hai to agr extra data add krne hai humein to false positive dene ki prob. badh jaati , isliye ye automatic naye filter create kr deta hai

compressed bf :

Iska kaam hai space bachana, jab aapko bloom filter ko network ke through bhejna ho ya store karna ho

Jaise zip file — ek folder ko compress karke bhejtae, taaki jldi transfer ho jaaye, lekinnn bss access thoda slow ho sakta h iska

sliding bf :

Ye filter real-time systems k liye hota hai jahan data continuously aata rehta hai aur sirf recent data important hota hai.

mtlb purana jo data hai vo apne aap expire hojayega , mtlb jese WhatsApp status apna 24hr hi last krta hai wese

Multi-layer bloom f :

Ye filter layers mein divided hota hai, jahan har layer ka purpose alag hota hai — jaise hierarchical search system.

Aap pehle ek top-level Bloom Filter check karte ho. Agar usme aaya yesss, toh next layer mein jaate ho — step by step deep search.

Isse search faster aur false positive control mein rehta hai.

Partitioned bf :

Bit collision aur false positive kam krta hai ye

Standard m kya ho rha tha multiple hash functions jb use horhe the tb ek hi bit ko overwrite kr rha tha baar baar , to yaha Bit array ko multiple partitions mein divide kar dete hain to har hash func apne assigned waale p hi work krega

Spectral Bloom Filter :

Standard ya Counting Bloom Filter bas presence check karte hain: Kya 'x' exist karta hai?

Lekin ye batayega ki x kitni baar aaya

EOT