**Must Know 18 System Design Concepts**

1. **Domain Name System (DNS)**

The Domain Name System(DNS) serves as a fundamental component of the internet infrastructure, translating user-friendly domain names into their corresponding IP addresses. It acts as a phonebook for the internet, enabling users to access websites and services by entering easily memorable domain names, such as [www.scaler.com](http://www.scaler.com), rather than the numerical IP addresses like “192.0.2.1” that computers utilize to identify each other.

When you input a domain name into your web browser, the DNS is responsible for finding the associated IP address and directing your request to the appropriate server. This process commences with your computer sending a query to a recursive resolver, which then searches a series of DNS servers, beginning with the root server, followed by the Top-Level Domain (TLD) server, and ultimately the authoritative name server. Once the IP address is located, the recursive resolver returns it to your computer, allowing your browser to establish a connection with the target server and access the desired content.

1. **Load Balancer**

A load balancer is a networking device or software designed to distribute incoming network traffic across multiple servers, ensuring optimal resource utilization, reduced latency, and maintained high availability. It plays a crucial role in scaling applications and efficiently managing server workloads, particularly in situations where there is a sudden surge in traffic or uneven distribution of requests among servers.

Load balancers employ various algorithms to determine the distribution of incoming traffic. Some common algorithms include:

* Round Robin: Requests are sequentially and evenly distributed across all available servers in a cyclical manner.
* Least Connections: The load balancer assigns requests to the server with the fewest active connections, giving priority to less-busy servers.
* IP Hash: The client’s IP address is hashed, and the resulting value is used to determine which server the request should be directed to. This method ensures that a specific client’s requests are consistently routed to the same server, helping maintain session persistence.

1. **API Gateway**

An API Gateway serves as a server or service that functions as an intermediary between external clients and the internal microservices or API-based backend services of an application. It is a vital component in contemporary architectures, particularly in microservices-based systems, where it streamline the communication process and offers a single entry point for client to access various services.

The primary functions of an API Gateway encompass:

1. Request Routing: The API Gateway directs incoming API requests from clients to the appropriate backend service or microservice, based on predefined rules and configurations.
2. Authentication and Authorization: The API Gateway manages user authentication and authorization, ensuring that only authorized clients can access the services. It verifies API keys, tokens, or other credentials before routing requests to the backend services.
3. Rate Limiting and Throttling: To safeguard backend services from excessive load or abuse, the API Gateway enforces rate limits or throttles requests from clients according to predefined policies.
4. Caching: In order to minimize latency and backend load, the API Gateway caches frequently-used responses, serving them directly to clients without the need to query the backend services.
5. Request and Response Transformation: The API Gateway can modify requests and responses, such as converting data formats, adding or removing headers, or altering query parameters, to ensure compatibility between clients and services.
6. **CDN**

A Content Delivery Network (CDN) is a distributed network of servers that store and deliver content, such as images, videos, stylesheets, and scripts, to users from locations that are geographically closer to them. CDNs are designed to enhance the performance, speed, and reliability of content delivery to end-users, irrespective of their location relative to the origin server.

Here’s how CDN operates:

1. When a user requests content from a website or application, the request is directed to the nearest CDN server, also known as an edge server.
2. If the edge server has the requested content cached, it directly serves the content to the user. This process reduces the latency and improves the user experience, as the content travels a shorter distance.
3. If the content is not cached on the edge server, the CDN retrieves it from the origin server or another near by CDN server. Once the content is fetched, it is cached on the edge server and served to the user.
4. To ensure the content stays up-to-date, the CDN periodically checks the origin server for changes and updates its cache accordingly.
5. **Forward Proxy Vs. Reverse Proxy**

A forward proxy, also referred to as a “proxy server” or simply “proxy” is a server positioned in front of one or more client machines, acting as an intermediary between the clients and the internet. When a client machine requests a resource on the internet, the request is initially sent to the forward proxy. The forward proxy then forwards the request to the internet on behalf of the client machine and returns the response to the client machine.

On the other hand, a reverse proxy is a server that sits in front of one or more web servers, serving as an intermediary between the se servers and the internet. When a client requests a resource on the internet, the request is first sent to the reverse proxy. The reverse proxy then forwards the request to one of the web servers, which returns the response to the reverse proxy. Finally, the reverse proxy returns the response to the client.

1. **Caching**

Cache is a high-speed storage layer positioned between the application and the original data source, such as a database, file system, or remote web service. When an application requests data, the cache is checked first. If the data is present in the cache, it is returned to the application. If the data is not found in the cache, it is retrieved from its original source, stored in the cache for future use, and then returned to the application. In a distributed system, caching can occur in multiple locations, including the client, DNS, CDN, load balancer, API gateway, server, database and more.

1. **Data Partitioning**

In a database, horizontal partitioning, often referred to as sharding, entails dividing the rows of table into smaller tables and storing them on distinct servers or database instances. This method is employed to distribute the database load across multiple servers, thereby enhancing performance.

Conversely, vertical partitioning involves splitting the columns of a table into separate tables. This technique aims to reduce the column count in a table and boost the performance of queries that only access a limited number of columns.

1. **Database Replication**

Database replication is a method employed to maintain multiple copies of the same database across various servers or locations. The main objective of database replication is to enhance data availability, redundancy, and fault tolerance, ensuring the system remains operational even in the face of hardware failures or other issues.

Fault Tolerance: A system’s ability to continue operating without interruption when one or more of its components fails.

In a replicated database configuration, one server serves as the primary (or master) database, while others acts as a replicas (or slaves). This process involves synchronizing data between the primary database and replicas, ensuring all possess the same up-to-date information. Database replication provides several advantages, including:

1. Improved Performance: By distributing read queries among multiple replicas, the load on the primary database can be reduced, leading to faster query response times.
2. High Availability: If the primary database experiences failure or downtime, replicas can continue to provide data, ensuring data uninterrupted access to the application.
3. Enhanced Data Protection: Maintaining multiple copies of the database across different locations helps safeguard against data loss due to hardware failures or other disasters.
4. Load Balancing: Replicas can handle read queries, allowing for better load distribution and reducing overall stress on the primary database.
5. **Distributed Messaging Systems**

Distributed messaging systems provide a reliable, scalable, and fault-tolerant means for exchanging messages between numerous, possibly geographically dispersed applications, services, or components. These systems facilitate communication by decoupling sender and receiver components, enabling them to develop and function independently. Distributed messaging systems are especially valuable in large-scale intricate systems, like those seen in microservices architecture or distributed computing environments. Examples of these systems include Apache Kafka, RabbitMQ.

1. **Microservices**

Microservices represent an architectural style wherein an application is organized as an assembly of small, loosely-coupled, and autonomously deployable services. Each microservice is accountable for a distinct aspect of functionality or domain within the application and communicates with other microservices via well-defined APIs. This method deviates from the conventional monolithic architecture, where an application is constructed as a single, tightly-coupled unit.

The primary characteristic of microservices include:

1. Single Responsibility: Adhering to the Single Responsibility Principle, each microservice focuses on a specific function or domain, making the services more straightforward to comprehend, develop, and maintain.
2. Independence: Microservice can be independently developed, deployed, and scaled, offering increased flexibility and agility in the development process. Teams can work on various services, simultaneously without impacting the entire system.
3. Decentralization: Typically, microservices are decentralized, with each service possessing its data and business logic. This approach fosters separation of concerns and empowers teams to make decisions and select technologies tailored to their unique requirements.
4. Communication: Microservice interact with each other using lightweight protocols, such as HTTP/REST, gRPC, or message queues. This fosters interoperability and facilitates the integration of new services or the replacement of existing ones.
5. Fault Tolerance: As microservices are independent, the failure of one service does not necessarily result in the collapse of the entire system, enhancing the application’s overall resiliency.
6. **NoSQL Databases**

NoSQL databases, or “Not Only SQL” databases are designed to store, manage, and retrieve unstructured or semi-structured data. They offer an alternative to traditional relational databases, which rely on structured data and predefined schemas. NoSQL databases have become popular due to their flexibility, scalability, and ability to handle large volumes of data, making well-suited for modern applications, big data processing, and real-time analytics.

NoSQL databases can be categorized into four main types:

1. Document-Based: These database store data in document-like structures, such as JSON or BSON. Each document is self-contained and can have its own unique structures, making them suitable for handling heterogeneous data. Examples of document-based NoSQL databases include MongoDB and Couchbase.
2. Key-Value: These databases store data as key-value pairs, where the key acts as a unique identifier, and the value holds the associated data. Key-Value databases are highly efficient for simple read and write operations, and they can be easily partitioned and scaled horizontally. Examples of Key-Value NoSQL databases include Redis, and Amazon DynamoDB.
3. Column-Family: These databases stores data in column families, which are groups of related columns. They are designed to handle write-heavy workloads and are highly efficient for querying data with a known row and column keys. Examples of column-family NoSQL databases include Apache Cassandra and HBase.
4. Graph-Based: These databases are designed for storing and querying data that has complex relationships and interconnected structures, such as social networks or recommendation systems. Graph database use nodes, edges, and properties to represent and store data, making it easier to perform complex traversals and relationships-based queries. Examples of graph-based NoSQL databases include Neo4j and Amazon Neptune.
5. **Database Index**

Database indexes are data structures that enhance the speed and efficiency of query operations within a database. They function similarly to an index in a book, enabling the database management system (DBMS) to swiftly locate data associated with a specific value or group of values, without the need to search through every row in a table. By offering a more direct route to the desired data, indexes can considerably decrease the time required to retrieve information from a database.

Indexes are typically constructed on one or more columns of a database table. The B-tree index is the most prevalent type, organizing data in a hierarchical structure, which allows for rapid search, insertion, and deletion operations. Other types of indexes, such as bitmap indexes and hash indexes, exists as well, each with their particular use cases and advantages.

Although indexes can significantly enhance query performance, they also involve certain trade-offs:

1. Storage Space: Indexes require additional storage space since they generate and maintain separate data structure alongside the original table data.
2. Write Performance: When data is inserted, updated, or deleted in a table, the corresponding indexes must also be updated, which may slow down write operations.
3. **Distributed File Systems**

Distributed file systems are storage systems designed to manage and grant access to files and directories across multiple servers, nodes, or machines, frequently distributed across a network. They allow users and applications to access and modify files as though they were situated on a local file system, despite the fact that the actual files may be physically located on various remote servers. Distributed file systems are commonly employed in large-scale or distributed computing environments to offer fault tolerance, high availability, and enhanced performance.

1. **Notification System**

These are used to send notifications or alerts to users, such as emails, push notifications or text messages.

1. **Full-text Search**

Full-text search allows users to search for particular words or phrases within an application or website. Upon receiving a user query, the application or website delivers the most relevant results. To accomplish this rapidly and effectively, full-text search utilizes an inverted index, a data structure that associates words or phrases with the documents where they are found. Elastic Search is an example of such systems.

1. **Distributed Coordination Services**

Distributed Coordination Services are systems engineered to regulate and synchronize the actions of distributed applications or services, or nodes in a dependable, efficient, and fault-tolerant way. They assist in maintaining consistency, addressing distributed synchronization, and overseeing the configuration and state diverse components in a distributed setting. Distributed Coordination Services are especially valuable in large-scale or intricate systems, like those encountered in microservices architectures, distributed computing environments, or clustered databases. Apache Zookeeper, etcd, and Consul are examples of such services.

1. **Heartbeat**

In a distributed environment, work/data is distributed among servers. To efficiently route requests in such a setup, servers need to know what other servers are part of the system. Furthermore, servers should know if other servers are alive and working. In a decentralized system, whenever a request arrives at a server, the server should have enough information to decide which server is responsible for entertaining that request. This makes the timely detection of server failure an important task, which also enables the system to take corrective actions and move the data/work to another healthy server and stop the environment from further deterioration.

To solve this each server periodically sends a heartbeat message to a central monitoring server or other servers in the system to show that it is still alive and functioning.

Heartbeating is one of the mechanisms for detecting failures in a distributed system. If there is a central server, all servers randomly choose a set of servers and send them a heartbeat message every few seconds. This way, if no heartbeat message is received from a server for a while, the system can suspect that the server might have crashed. If there is no heartbeat within a configured timeout period, the system can conclude that the server is not alive anymore and stop sending requests to it and start working on its replacement.

1. **Checksum**

In a distributed system, while moving data between components, it is possible that the data fetched from node may arrive corrupted. This corruption can occur because of faults in a storage device, network, software, etc. How can a distributed system ensure data integrity, so that the client receives an error instead of corrupt data?

To solve this, we can calculate a checksum and store it with data.

To calculate a checksum, a cryptographic hash function like MD4, SHA-1, SHA-256, or SHA-512 is used. The hash function takes the input data and produces a string (containing letters and numbers) of fixed length; this string is called the checksum.

When a system storing some data, it computes a checksum of the data and stores the checksum with the data. When a client retrieves the data, it verifies that the data it retrieved from the server matches the checksum stored. If not, then the client can opt to retrieve the data from another replica.