Cracking the System Design Interview [Theory Basics]

Courtesy of Artem Sapegin on Unsplash

System design interviews are in general 45–60 minutes long where you are given a broad topic like “Design Twitter”. You’ll be expected to generate a high-level design, showing the different system components that will be required, how they’re connected, and any trade-offs in the approach you’ve taken. Time permitting, the interviewers will ask you to go deeper into how some of your system components would function and talk about how you would handle several scenarios and edge cases.

For a junior, mid-level engineer, this article aims to give you a fair good start at the System Design Interview and it’s not an exhausting list since system design is a vast topic. But from here, you can dig deeper with other resources.

Before starting with the theory basics, let's briefly talk about how you should approach the interview.

1. **Understand the goal and requirements.**
   Never go head-first into the design, make sure you understand the problem and ask clarifying questions — What is the goal of the system? Who are the users of the system? What do they need it for? How are they going to use it? Clarify your assumptions.

2. **Establish the scope**
   Go through all the features of the system that you can think of and write them down.

3. **Establish the scale**
   What is the scale of the system? How many customers does the system need to serve per day? How many transactions? What are the performance requirements (e.g. latency, throughput, availability)? What is the expected read-to-write ratio? What's the average expected response time?

4. **Design high-level.** The first 20 min of the interview you should have a draft of your high level design. Make sure to discuss the trade offs and why you chose those types of technologies. What type of database would you use and why? What caching solutions are out there? Which would you choose and why?

5. **Drill-down**. You should drill-down into more details, starting with the components that you are most familiar with. This is to ensure that you have time to go in-depth into the topics you have the most knowledge in.

6. **Wrap-up.** Before the end of your interview, you should spend some time to summarize your solution and highlight important bottlenecks or improvement opportunities.

Now let's touch onto the most encountered components you'll probably have to talk about.

Distributed Systems

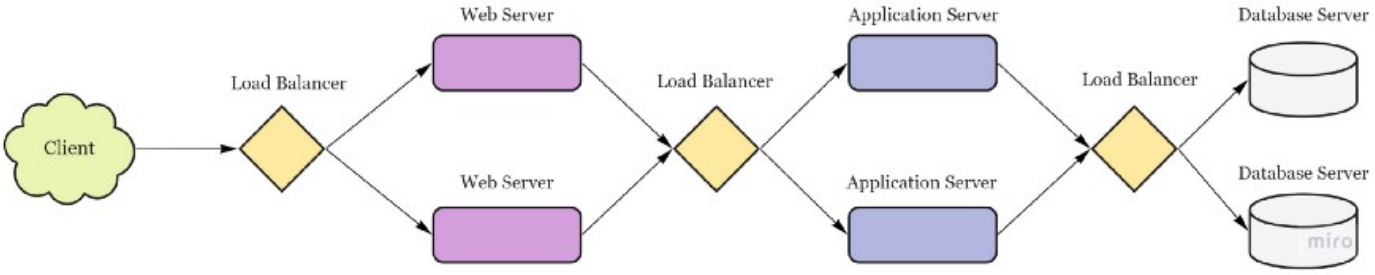
= group of computers working together as to appear as a single computer to the end-user. They have a shared state, operate concurrently and can fail independently without affecting the whole system's uptime.

Key Characteristics:

- **Stability** or **Serviceability** or **Manageability** = *simplicity and speed with which a system can be repaired or maintained.*
- **Reliability** = *probability a system will fail in a given period*. A distributed system is considered reliable if it keeps delivering its services even when one or several of its software or hardware components fail.
- **Scalability** = *capability of a system, process, or a network to grow and manage increased demand.*
- **Efficiency/Performance** = *response time* (or latency = delay to obtain the first item) + *throughput* (or bandwidth = the number of items delivered in a given time unit)
- **Availability** = *simple measure of the percentage of time that a system, service, or a machine remains operational under normal conditions.*
- **Fault tolerance** and **Catastrophe-preparedness**

Load Balancer

component that acts as a reverse proxy and distributes network or application traffic across a number of servers to improve responsiveness and availability of applications, websites or databases.



- **Benefits**: faster, uninterrupted service, less downtime and higher throughput, predictive analytics that predict traffic bottlenecks (smart LB), fewer failed or stressed components.

Algorithms (used to balance the load):

- **Least Connection Method** (directs traffic to the server with the fewest active connections)
- **Least Response Time Method** (directs traffic to the server with the fewest active connections)
- **Least Bandwidth Method** (selects the server that is currently serving the least amount of traffic)
- **Round Robin Method** (cycles through a list of servers and sends each new request to the next server)
- **Weighted Round Robin Method** (servers with higher weights (= an integer value that indicates the processing capacity) receive new and more connections before those with less weights) — to handle servers with different characteristics (e.g. processing power, availability, load)
- **IP Hash** (a hash of the IP address of the client is calculated to redirect the request to a server)

Caching

= temporary storage area that stores the result of expensive responses or frequently accessed data in memory so that subsequent requests are served more quickly.

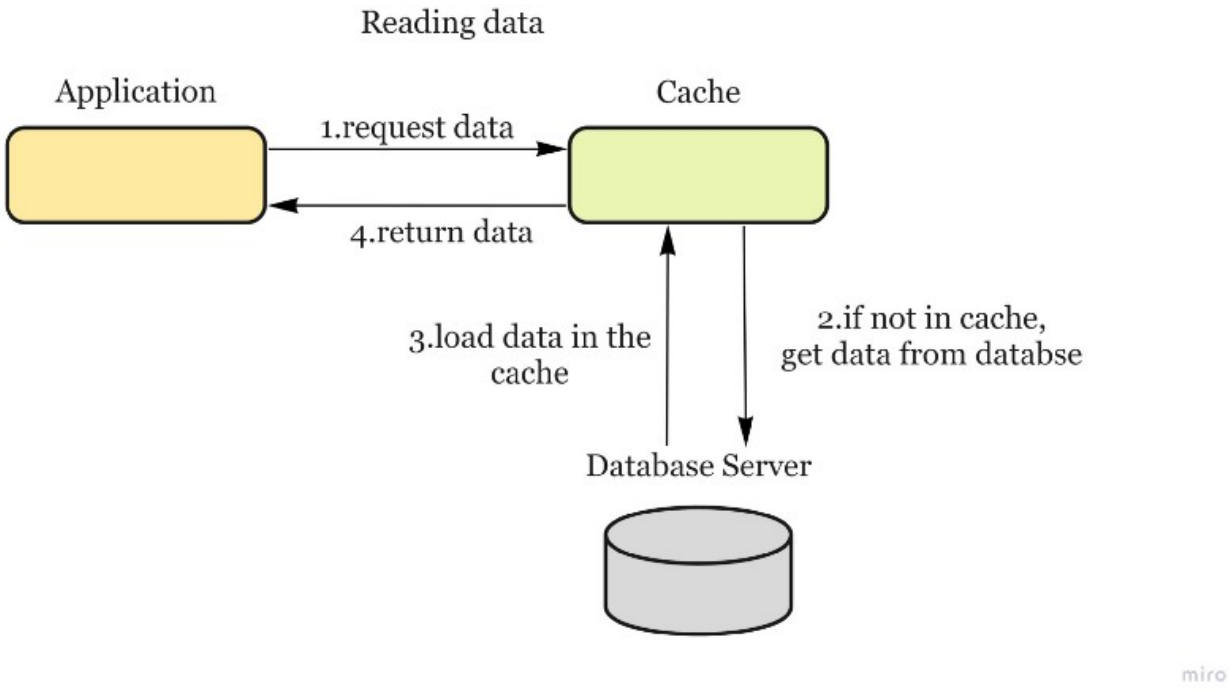
When to use it — when data is read frequently but modified infrequently

Considerations:

- Set an **expiration policy** that is not too short (as it reloads data too fast from the database) and not too long (since data can become stale).
- How to keep the cache and database **consistent** (see the *cache invalidation* section below)
- The cache can easily become a **single point of failure,** you might need to use multiple caches
- Choose an **eviction policy** (see the *cache eviction policies* section below)

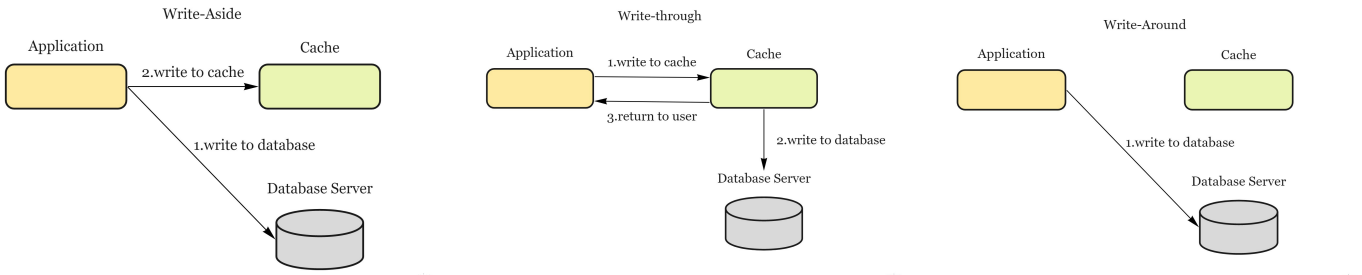
- Think how you handle **concurrency** (aka the reader-writer problem) — When multiple clients are trying to update the cache at the same time, there can be conflicts. A solution would be to use **commit logs**. To update the cache, we can store all the mutations into logs rather than update immediately. And then some background processes will execute all the logs asynchronously. This strategy is commonly adopted in database design.

Reading from cache:

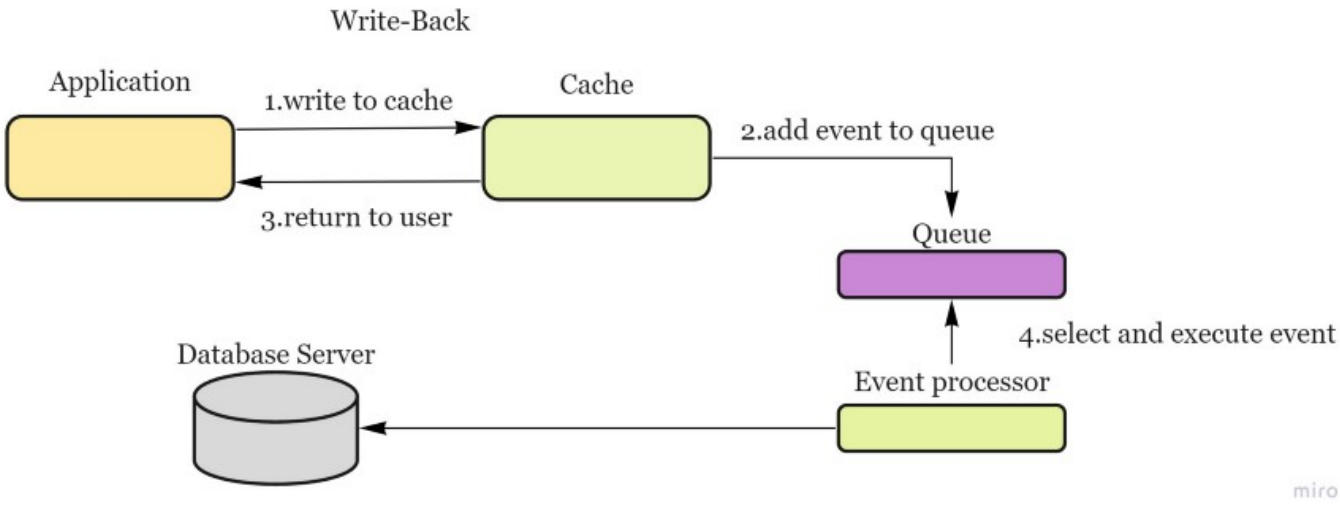


Cache Invalidation = happens when data is modified in the database and becomes inconsistent with the cache.

Updating the cache:



- **Write-aside cache** — data is written into the cache and the corresponding database at the same time (also known as *lazy loading*) — cons: higher latency
- **Write-through cache** — application adds or updates an entry in the cache. Cache synchronously writes entry to the data store. Cache returns the data to the application.
- **Write-around cache** — data is written directly to permanent storage, bypassing the cache — cons: read request for recently written data will create a "cache miss".
- **Write-back cache** — data is written to cache alone and completion is immediately confirmed to the client and then data is *asynchronously* updated to the data store — pros: low latency and high throughput; cons: risk of data loss in case of a crash



Cache eviction policies

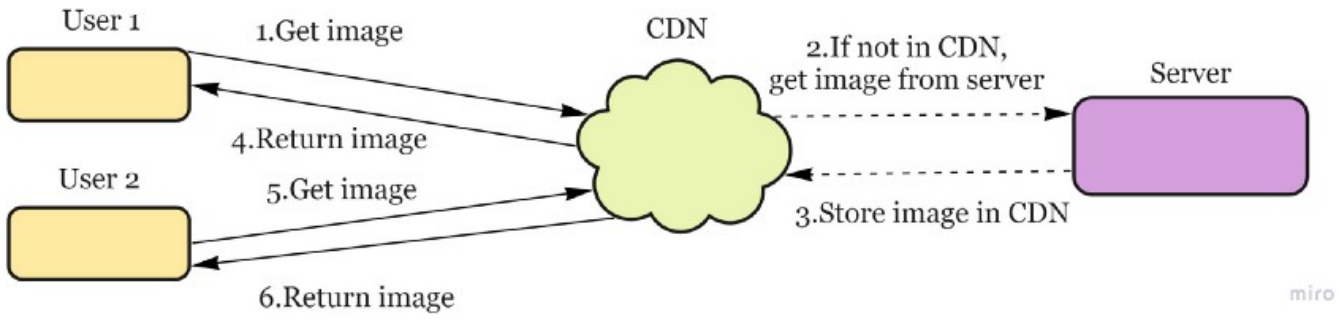
- **First In First Out (FIFO)** — evicts the first block accessed first
- **Last In First Out (LIFO)** — evicts the block accessed most recently
- **Least Recently Used (LRU)** — evicts the least recently used items first. When the client requests resource A, it happens as follow:
  If A exists in the cache, we just return immediately.
  If not and the cache has extra storage slots, we fetch resource A and return to the client. In addition, insert A into the cache.
  If the cache is full, we kick out the resource that is least recently used and replace it with resource A.

- **Most Recently Used (MRU)** — evicts the most recently used items first
- **Least Frequently Used (LFU)** — counts how often an item is needed. Those that are used least often are discarded first.
- **Random Replacement (RR)** — randomly evicts a candidate item

Concurrency — reader-writer problem — When multiple clients are trying to update the cache at the same time, there can be conflicts. => use **commit logs**. To update the cache, we can store all the mutations into logs rather than update immediately. And then some background processes will execute all the logs asynchronously. This strategy is commonly adopted in database design.

Content Distribution Network (CDN)

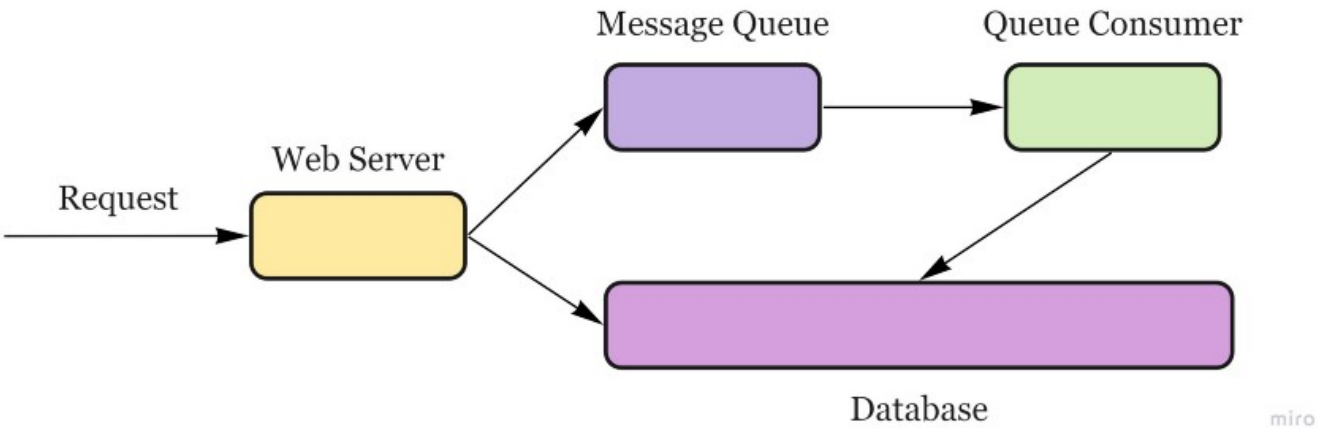
= system of distributed servers (network) that deliver webpages and other web content to a user based on the geographic locations of the user, the origin of the webpage and a content delivery server.



System flow: a request will first ask the CDN for a piece of static media; the CDN will serve that content if it has it locally available. If it isn't available, the CDN will query the back-end servers for the file, cache it locally, and serve it to the requesting user.

Message Queues

= durable component, stored in memory, that supports asynchronous communication. Helps reduce request times for expensive operations that would otherwise be performed in-line. Receives, holds, and delivers messages.



If an operation is too slow to perform inline, you can use a message queue with the following workflow:

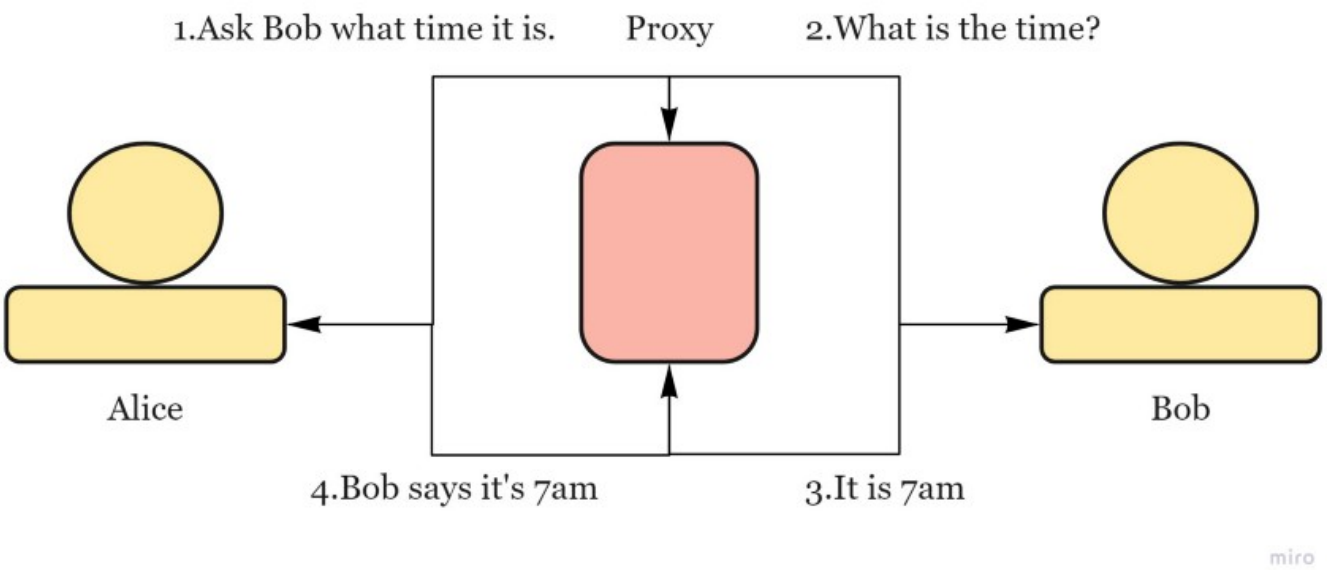
- An application publishes a job to the queue, then notifies the user of job status.
- A worker picks up the job from the queue, processes it, then signals the job is complete.

e.g. RabbitMQ, Amazon SQS

Proxies

= piece of software or hardware that acts as an intermediary for requests from clients seeking resources from other servers.

- used to filter requests, log requests, or sometimes transform requests. Its cache can serve a lot of requests (If multiple clients access a particular resource, the proxy server can cache it and serve it to all the clients without going to the remote server.)



Types:

- **Open Proxy** — accessible by any Internet user. Allows users within a network group to store and forward Internet services such as DNS or web pages to reduce and control the bandwidth used by the group.

Anonymous Proxy — reveals its identity as a server but does not disclose the initial IP address

Transparent Proxy — identifies itself, and with the support of HTTP headers, the first IP address can be viewed.

- **Reverse Proxy** — retrieves resources on behalf of a client from one or more servers

Databases

SQL vs NoSQL

NoSQL Types

Sharding or Data Partitioning

= process of splitting up a DB/table across multiple machines to improve the manageability, performance, availability, and load balancing of an application

Partitioning Methods:

- **Horizontal Partitioning** — put different rows into different tables — **cons**: unbalanced servers (if the value whose range is used for sharding isn't chosen carefully)
- **Vertical Partitioning** divide our data to store tables related to a specific feature in their own server (instagram use case — user profile information stored on one DB server, friend lists on another) — **cons**: if our application experiences additional growth, then it may be necessary to further partition a feature specific DB across various servers
- **Directory Based Partitioning** — create a lookup service which knows your current partitioning scheme and abstracts it away from the DB access code.

Partitioning Criteria:

- **Key or Hash-based partitioning** — apply a hash function to some key attributes of the entity we are storing — **cons**: adding new servers means changing the hash function which would require redistribution of data and downtime for the service. (use Consistent Hashing)

- **List partitioning** — each partition is assigned a list of values, so whenever we want to insert a new record, we will see which partition contains our key and then store it there
- **Round-robin partitioning** — With 'n' partitions, the 'i' tuple is assigned to partition (i mod n).
- **Composite partitioning** — combine any of the above partitioning schemes to devise a new scheme

Complexities:

Joins and Denormalization — not feasible to perform joins that span database shards (denormalize the database so that queries that previously required joins can be performed from a single table)

Referential Integrity — enforcing foreign keys in a sharded database can be extremely difficult. (not supported by most RDBMS — has to be enforced in application code.)

Celebrity problem — how would you deal with hotspots? (one way would be to add a shard for each celebrity)

Rebalancing/Resharding — data distribution is not uniform and there is a lot of load on a shard => have to create more DB shards or have to rebalance existing shards.

Strategies:

- **Fixed number of partitions** — Create many more partitions than there are nodes, and assign several partitions to each node
- **Dynamic partitioning** — when a partition grows to exceed a configured size, it is split into 2 partitions so that approximately half of the data ends up on each side of the split (no of partitions adapt to the total data volume)
- **Partitioning proportionally to nodes** — have a fixed no of partitions per node. The size of each partition grows proportionally to the dataset size while the no of nodes remains unchanged

Examples of Real World Scenarios:

- ***Read about Tinder's Geosharded Recommendations Part 1: Sharding Approach*** — Based on Google's S2 which uses the **Hilbert curve** (=*a space-filling curve that preserves spatial locality: two points that are close on the Hilbert curve are close in physical space. Each smallest Hilbert curve clone is a cell, and 4 adjacent cells form a bigger cell.*) Create the geoshards by enumerating all the possible container sizes, and calculate the standard deviation of each sharding configuration, the one with smallest standard deviation will be the most balanced geo sharding configuration we are looking for.
- ***Read about Uber's Unwinding Uber's Most Efficient Service*** — using a QuadTree — Take a flat projection of your search space and divide it into quarters that we'll call cells. You then divide each of those cells into quarters recursively until you hit a defined maximum depth which will be the leaves of the tree.
- ***Read about Instragram's Sharding & IDs at Instagram***

Replication

= keeping a copy of the same data on multiple machines that are connected via a network.

Reasons:

- **Latency** — Keep data geographically close to your users (reduce access latency)
- High **availability** — Allow the system to continue working even if some parts of it have failed
- **Scalability** — Scale out the number of machines that can serve read queries

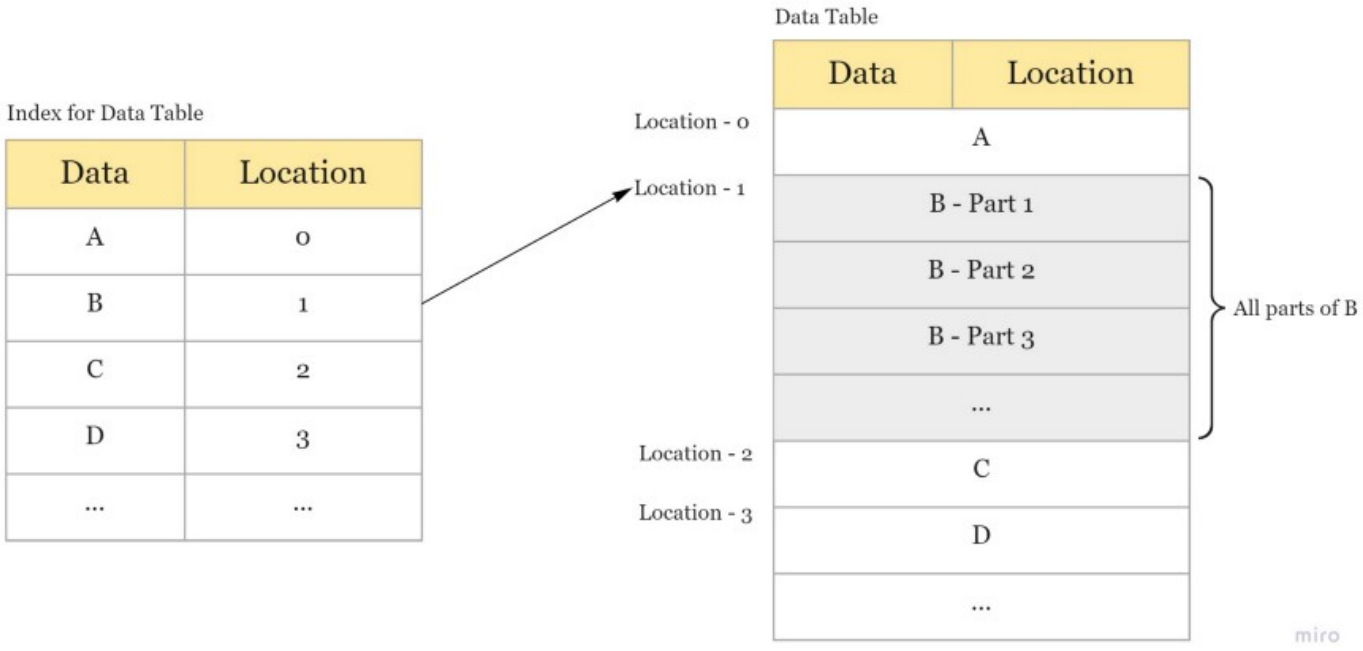
Leader-based replication:

- One of the replicas is designated as the **leader**. When clients want to write to DB, they send their requests to the leader, which writes the data to its local storage
- The other replicas (**followers**) receive the data from the leader in a process called **replication log** or **change stream**.
- When a client wants to read from the DB, it can query both leader/any of the followers.

Indexes

= data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure.

Can be created using one or more columns of a database table.



ACID

= set of properties of database transactions intended to guarantee validity even in the event of errors, power failures, etc.

In the context of databases, a sequence of database operations that satisfies the ACID properties (and these can be perceived as a single logical operation on the data) is called a **transaction**. For example, a transfer of funds from one bank account to another, even involving multiple changes such as debiting one account and crediting another, is a single transaction.

Atomicity — means that you guarantee that either all of the transaction succeeds or none of it does. You don't get part of it succeeding and part of it not. If one part of the transaction fails, the whole transaction fails. With atomicity, it's either "all or nothing".

Consistency — this ensures that you guarantee that all data will be consistent. All data will be valid according to all defined rules, including any constraints, cascades, and triggers that

have been applied on the database.

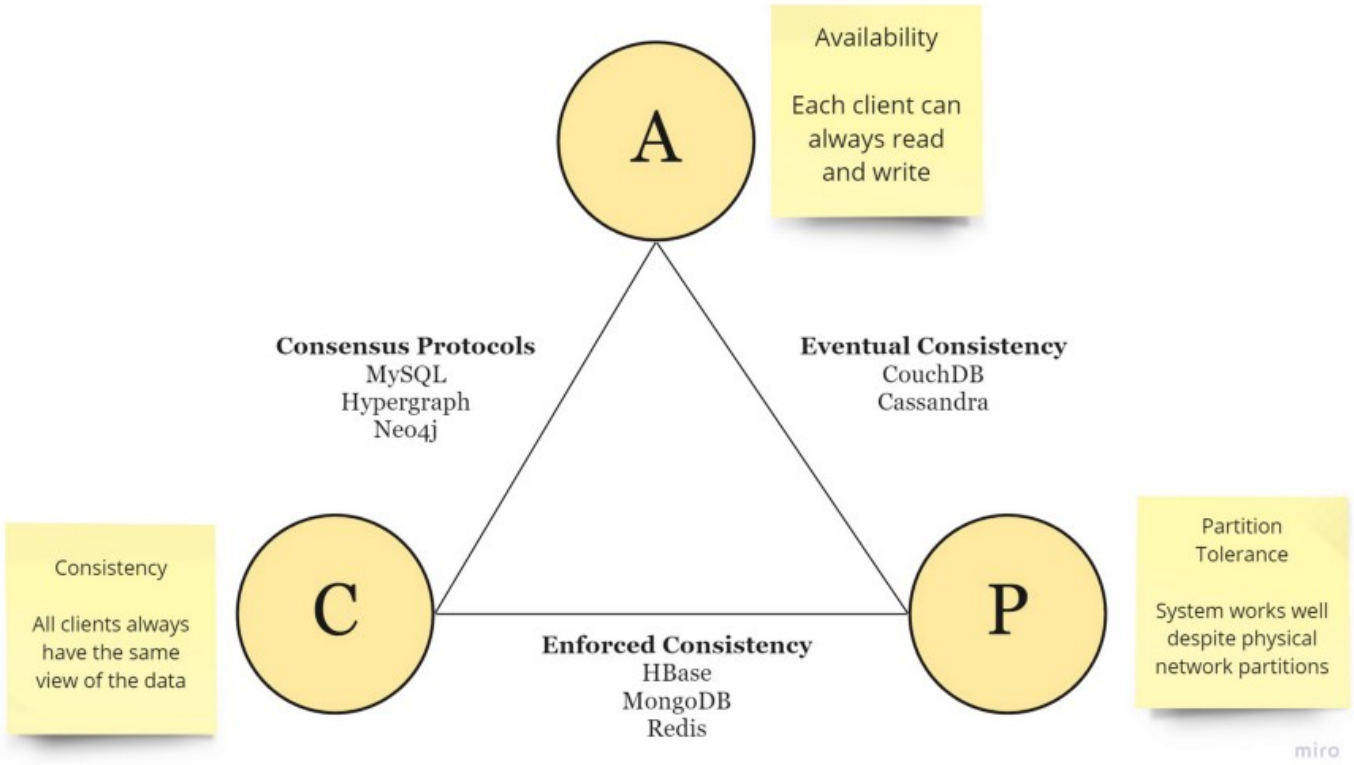
Isolation — guarantees that all transactions will occur in isolation. No transaction will be affected by any other transaction. So a transaction cannot read data from any other transaction that has not yet completed.

Durability — means that, once a transaction is committed, it will remain in the system — even if there's a system crash immediately following the transaction. Any changes from the transaction must be stored permanently. If the system tells the user that the transaction has succeeded, the transaction must have, in fact, succeeded.

CAP

impossible for a distributed software system to simultaneously provide more than two out of three of the following guarantees (CAP): Consistency, Availability, and Partition tolerance

- **Consistency**: All nodes see the same data at the same time. Consistency is achieved by updating several nodes before allowing further reads.
- **Availability**: Every request gets a response on success/failure. Availability is achieved by replicating the data across different servers.
- **Partition tolerance:** The system continues to work despite message loss or partial failure. Means that the cluster continues to function even if there is a "partition" (communication break) between two nodes (both nodes are up, but can't communicate).



All the combinations available are:

- **CA** — data is consistent between all nodes — as long as all nodes are online — and you can read/write from any node and be sure that the data is the same, but if you ever develop a partition between nodes, the data will be out of sync (and won't re-sync once the partition is resolved).
- **CP** — data is consistent between all nodes, and maintains partition tolerance (preventing data desync) by becoming unavailable when a node goes down.
- **AP** — nodes remain online even if they can't communicate with each other and will resync data once the partition is resolved, but you aren't guaranteed that all nodes will have the same data (either during or after the partition)

Conclusion

I hope this article can be the first stepping stone on your journey to ace your system design interview. Good luck and reach out for any questions or feedback 😀

This article is part of a *Software Engineer Interviews* series. For the other chapters check:

- Cracking the Product Interview for Engineers

- 17 Red Flags to Look for in Software Engineering Interviews

Recommended Resources

(in order of personal preference)

- **"System Design Interview — an Insider's Guide"** by Alex Xu
- **"Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems"** by Martin Kleppmann
- **"Building Microservices: Designing Fine-Grained Systems"** by Sam Newman
- **"Fundamentals of software architecture an engineering approach"** by Mark Richards and Neal Ford
- **"Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services"** by Brendan Burns
- **"Software Architecture: the Hard Parts"** by Pramod Sadalage, Mark Richards, Neal Ford, Zhamak Dehghani
- **"The Software Architect Elevator: Redefining the Architect's Role in the Digital Enterprise"** by Gregor Hohpe

- **“Architecting for Scale: How to Maintain High Availability and Manage Risk in the Cloud”** by Lee Atchison

Thanks to Alessandro Butler