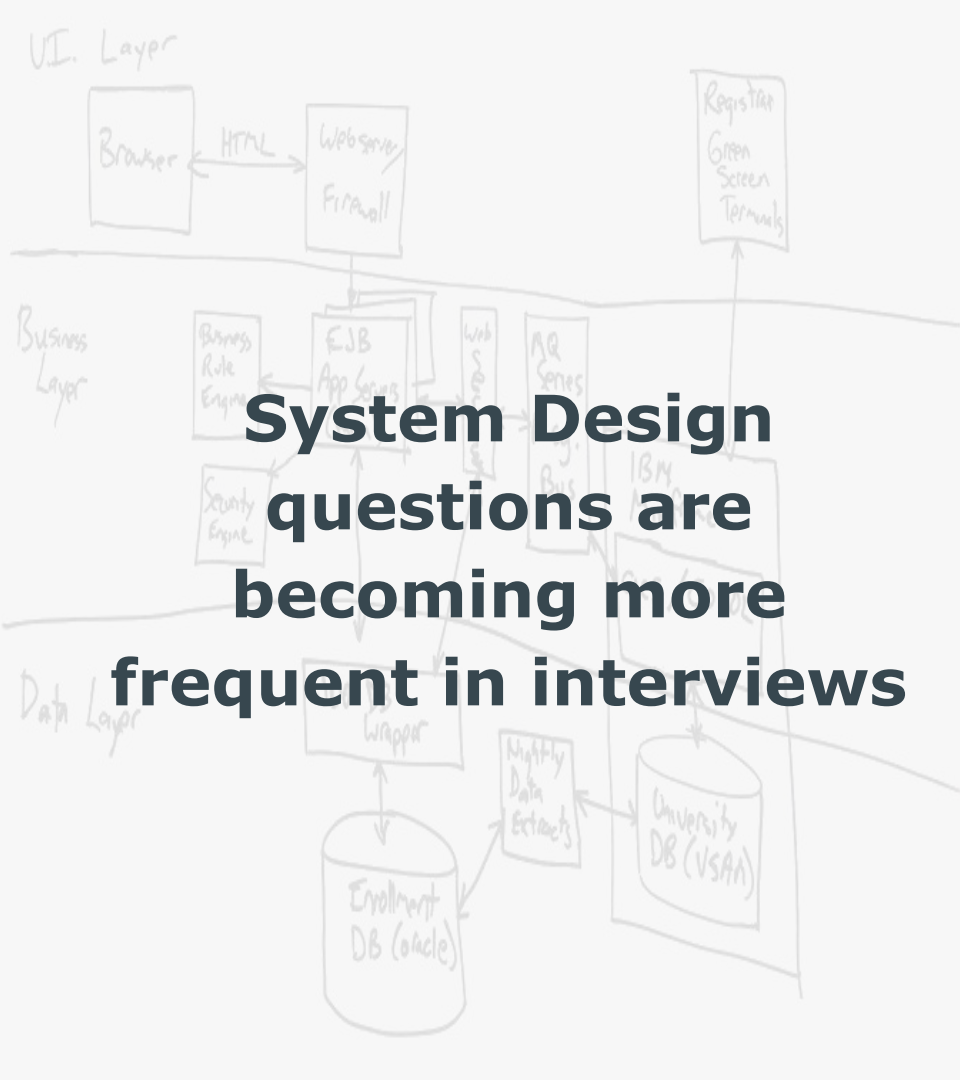
Module: System Design

...

Preparing for System Design questions in Interviews

Why?

I thought we just needed to worry about whiteboarding algorithms



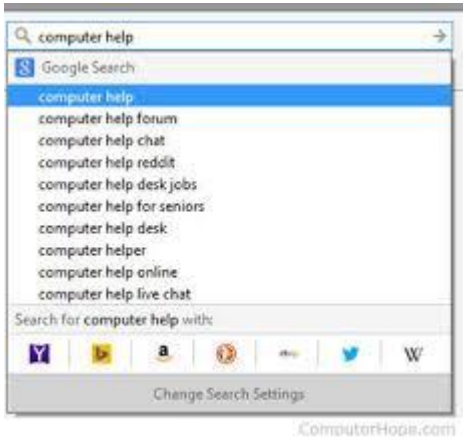
System Design questions are becoming more frequent in interviews

We’re going to do this in two steps

1) We’ll go step by step through the system design interview process
2) We’ll look at the some of the tools available to us in system design

What kind of System Design questions might we see?

Build...



System Design Interview

A step by step guide

Don’t sweat it! System Design Interviews can be broken down to three things

Scoping the problem: Don’t make assumptions; Ask clarifying questions to understand the constraints and use cases.

Sketching up an abstract design Illustrating the building blocks of the system and the relationships between them.

Identifying and addressing the bottlenecks by using the fundamental principles of scalable system design.

Scope the problem

1. Define your requirements

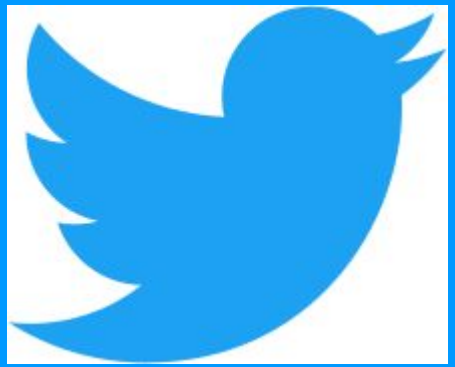
Never dive straight in. Start out by asking questions.

Who are our users and **what are their needs**?
What is the exact **scope** of the problem we're solving?
What are non-functional requirements?
What are our stretch goals?

Design questions are mostly open-ended, and they don't have ONE correct answer, that's why clarifying ambiguities early in the interview becomes critical.

Candidates who spend enough time to clearly define the end goals of the system always have a better chance to be successful in the interview.

Let’s say we’re designing



Ask clarifying questions

- How many total users do we expect? How many active users every day?
- Will users of our service be able to post tweets and follow other people?
- Should we also design to create and display user’s timeline?
- Will tweets contain photos and videos?
- Are we focusing on backend only or are we developing front-end too?
- Will users be able to search tweets?
- Would there be any push notification for new (or important) tweets?

2. Get a rough estimate of scale

It’s always a good idea to estimate the scale of the system you’re going to design. This would also help later when you’ll be focusing on scaling, partitioning, load balancing and caching.

- What scale is expected from the system? Number of total users, number of daily active users, number of new tweets, how many followers per user on average, etc.
- How much storage would we need? We’ll have different numbers if users can have photos and videos in their tweets.
- What network bandwidth usage are we expecting? This would be crucial in deciding how would we manage traffic and balance load between servers.

From information we've received from our interviewer, we know that we have:

- One billion total users
- 200 million Daily Active Users (DAU).
- 100 million new tweets every day
- On average each user follows 200 people.

Storage requirements: Let's say each tweet has 140 characters and we need two bytes to store a character without compression. Let's assume we need 30 bytes to store metadata with each tweet (like ID, timestamp, user ID, etc.). Total storage we would need:

100M tweets/day * (280 + 30) bytes/tweet => 30GB/day

How many total tweet-views our system will generate? Let's assume on average a user visits their timeline two times a day and visits five other people's pages. On each page if a user sees 20 tweets, total tweet-views our system will generate:

200M DAU * ((2 + 5) * 20 tweets) => 28B/day

Bandwidth estimate:

(28B * 280 bytes) / 86400s of time => 93MB/s

Note: This kind of depth lends itself more to backend heavy roles. You don't *have* to go this deep determining impacts of scale, but you *can* and doing so illustrates a deeper, more mature understanding of system design.

Sketch up an abstract design

3. Mock out a basic UI

This can be a very rough sketch.

Consider what will be required for your user(s) to complete a single transaction, from initiation to feedback. What UI elements are required?

4. Define your data model

- This is where the database(s) will be chosen, as well as block storage for things like photos and videos.
- Defining the data model early will clarify how data will flow among different components of the system.
- Later, it will guide towards data partitioning and management.

User
UserId
Name
Email
CreationDate
LastLogin
...

Tweet
TweetId
UserId
Content
TweetLocation
NumberOfLikes
Timestamp
...

UserFollowers
UserId
FollowerId
...

FavoriteTweets
UserId
TweetId
Timestamp
...

5. Define your APIs

Define what APIs are expected from the system. This would not only establish the exact contract expected from the system but would also ensure if you haven't gotten any requirements wrong.

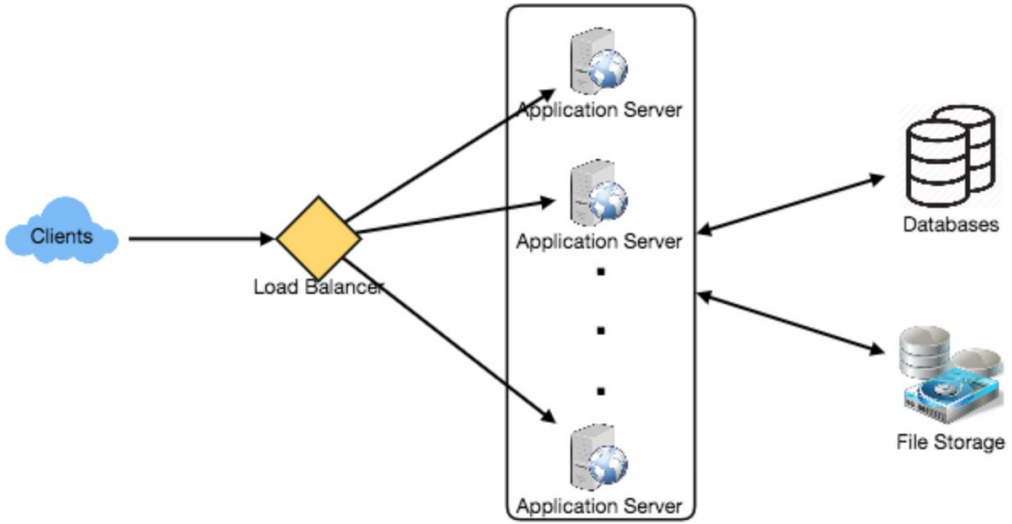
```
postTweet( userId, tweetData, tweetLocation, userLocation, timestamp, …)

generateTimeline( userId, currentTime, userLocation, …)

markTweetFavorite( userId, tweetId, timestamp, …)
```

6. High Level Design

Draw a block diagram with 5-6 boxes representing core components of your system.

You should identify enough components that are needed to solve the actual problem from end-to-end.

Think of this as sketching out your MVP.



7. Detailed Design

Dig deeper into these high level components. Think through the challenges presented for each one. You should be able to provide different approaches, their pros and cons, and why would you choose one.

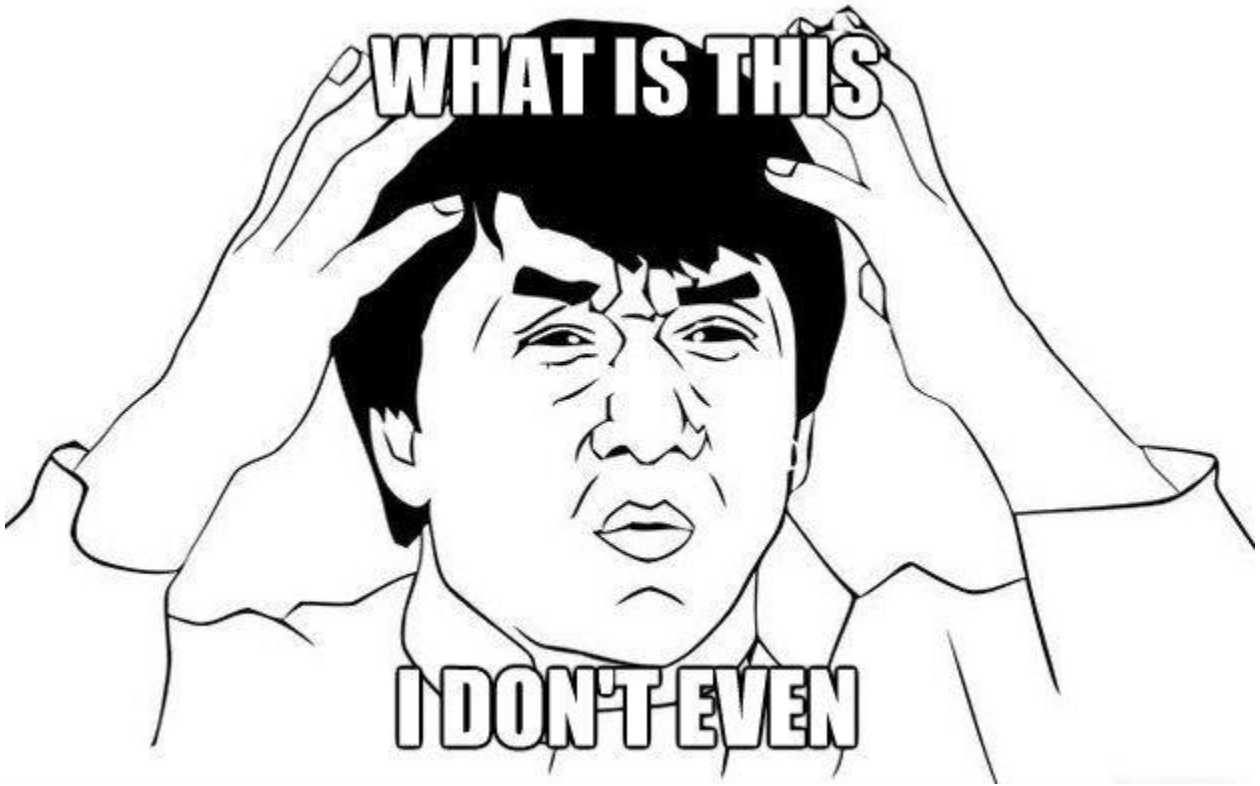
- Since we'll be storing a huge amount of data, how should we partition our data to distribute it to multiple databases? Should we try to store all the data of a user on the same database? What issue can it cause?
- How would we handle hot users, who tweet a lot, follow lots of people, or have lots of followers?
- Since user's timeline will contain most recent (and relevant) tweets, should we try to store our data in such a way that is optimized to scan latest tweets?
- How much and at which layer should we introduce caches to speed things up?
- What components need better load balancing?

Identify and Address Bottlenecks

8. Identify and Resolve Bottlenecks

Try to discuss as many bottlenecks as possible and different approaches to mitigate them.

- Is there any single point of failure in our system? What are we doing to mitigate it?
- Do we have enough replicas of the data so that if we lose a few servers, we can still serve our users?
- Similarly, do we have enough copies of different services running, such that a few failures will not cause total system shutdown?
- How are we monitoring the performance of our service? Do we get alerts whenever critical components fail or their performance degrade?



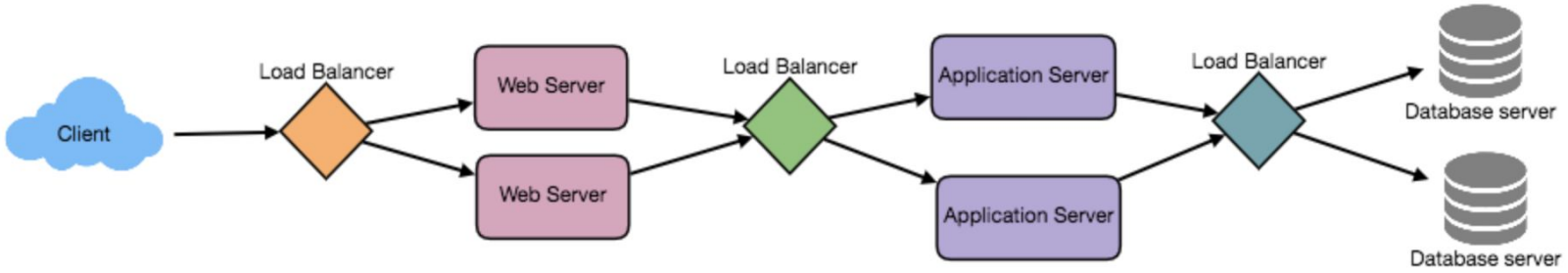
Don’t worry! These challenges can be broken down like any other.
You just need to be aware of the tools you have available to you...

Challenge:

We need to handle many client connections and many requests from those clients without overwhelming a single server

Load Balancers

- Load Balancing works to **distribute load** across multiple resources.
- It also **keeps track of the status** of all the resources while distributing requests.
- It can be utilized at various points throughout the system.
- Methods of distribution
  a. Round Robin, IP Hash, Least Response Time, and more...



Challenge:

We want to limit the number of times we're fetching data

Caching

- Load balancing helps you scale horizontally across an ever-increasing number of servers, but caching will enable you to make vastly better use of the resources you already have
- Caches take advantage of a simple principle: *recently requested data is likely to be requested again.*
- They are used in almost every layer of computing: hardware, operating systems, web browsers, web applications and more.

Cache Invalidation

While caching is fantastic, it does require some maintenance for keeping cache coherent with the source of truth (e.g., database). If the data is modified in the database, it should be *invalidated* in the cache, if not, this can cause inconsistent application behavior.

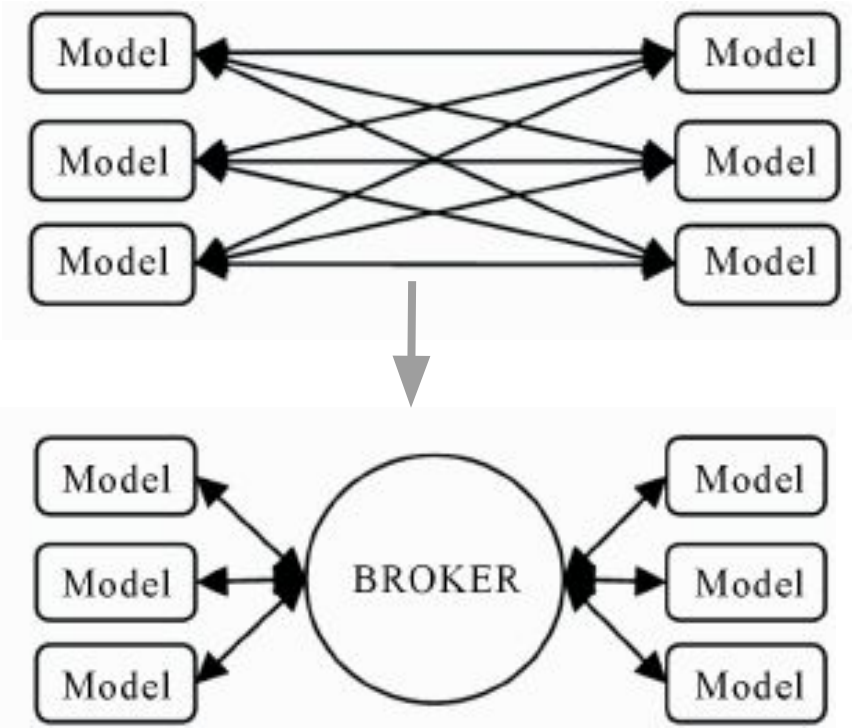
Solving this problem is known as **cache invalidation**, there are two basic writing schemes that are used:

1. **Write-through cache:** Under this scheme data is written into the cache and the corresponding database *at the same time*.
2. **Write-back cache:** Data is *written to cache alone*, and completion is immediately confirmed to the client. The write to the permanent storage is done later, after specified intervals or under certain conditions.

Challenge:

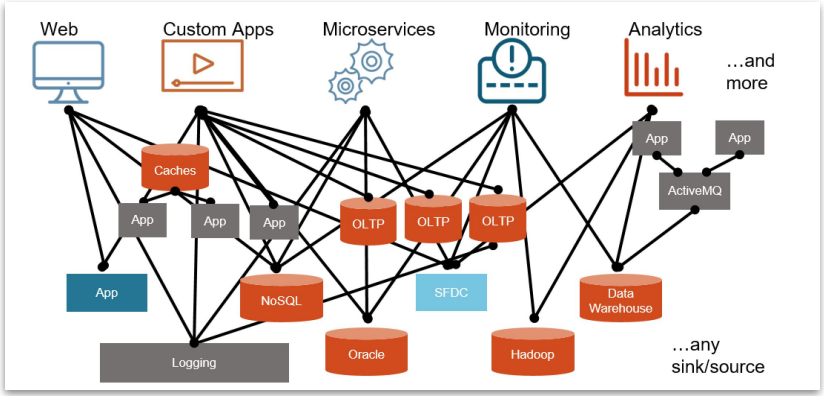
We’ve got a distributed system and there are SO many messages going back and forth between servers on the network

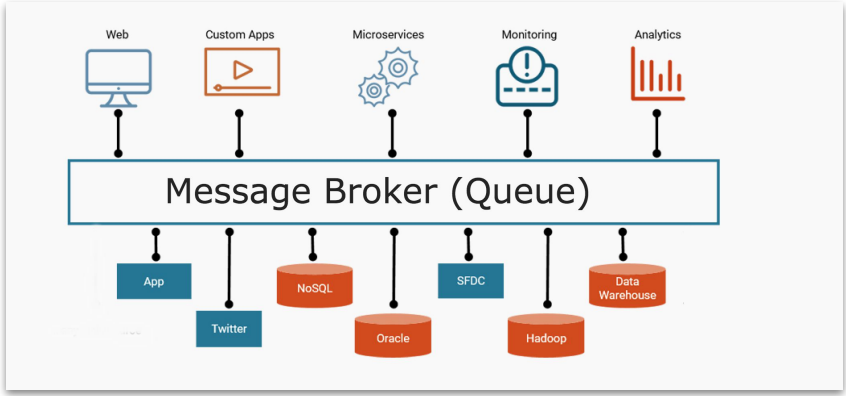
Message Brokers (Queues)



Message Brokers (Queues)

Queues are used to effectively manage requests in a large-scale distributed system. They allow us to decouple our processes and distribute/throttle processing load



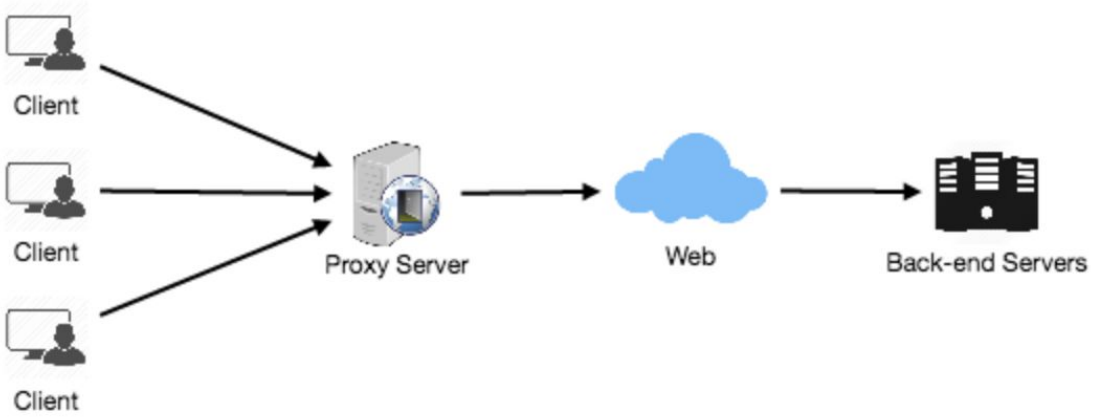


Challenge:

Our clients are in-house and we need to set up a shared cache or monitor/restrict internet usage

Proxies

- A proxy server is a piece of hardware/software that sits between the client(s) and the internet. It receives requests from clients and relays them to the back end servers over the internet.
- Typically, proxies are used to filter or log requests, block sites, provide anonymity, etc.
- Another advantage of a proxy server is that its cache can serve a lot of requests.



Challenge:

We need to add security, efficiency, or high availability to our back end

Reverse Proxies

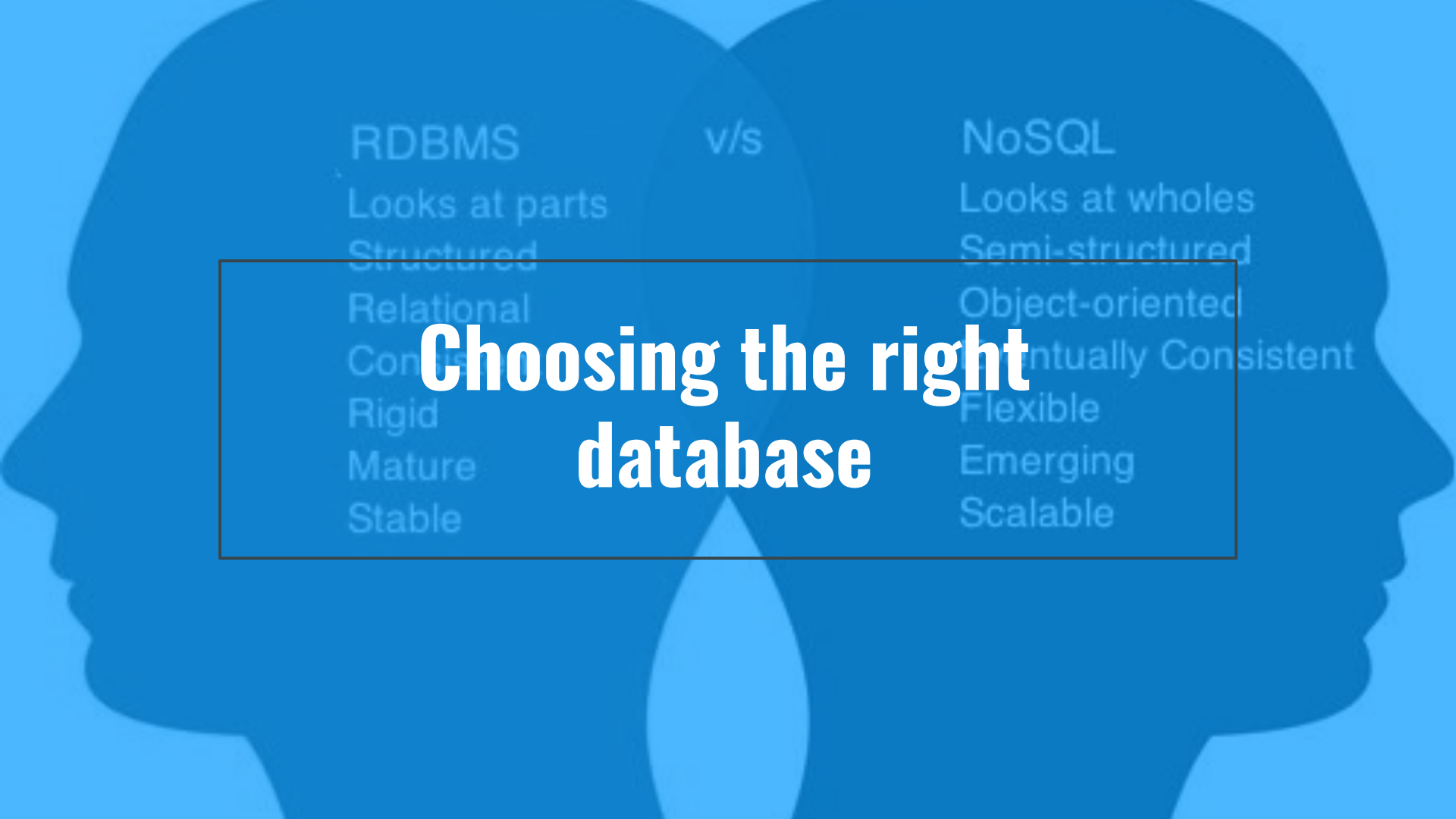
- A reverse proxy server is a piece of hardware/software that typically sits between the internet and the back-end server. It receives requests from clients and relays them to the origin servers.
- Typically, reverse proxies are used for load balancing, web acceleration via transforming requests (by adding/removing headers, encrypting/decrypting, or compression), and adding additional security for your back end servers.
- Another advantage of a proxy server is that its cache can serve a lot of requests.

Distributed Systems Summary

- Load Balancers
  - Distribution methods
- Cache
  - Cache coherence/invalidation
- Message Brokers
  - Fault tolerance via decoupling

Databases

- Relational databases are structured and have predefined schemas
- Non-relational databases are unstructured, distributed and have a dynamic schema



Choosing the right database

Reasons to use a Relational Database

You need to ensure **ACID compliance**.
Your data is **structured** and **unchanging**.

Generally, NoSQL databases sacrifice ACID compliance for scalability and processing speed, but for many e-commerce and financial applications, an ACID-compliant database remains the preferred option.

Reasons to use a Non-Relational Database

- You need to store **large volumes of data** that often require **little to no structure**.
- Making the most of cloud computing and storage for **horizontal scaling**.
- NoSQL is useful for **rapid development** as it doesn't need to be prepped ahead of time.

Big data is contributing to a large success for NoSQL databases, mainly because it handles data differently than the traditional relational databases. A few popular examples of NoSQL databases are MongoDB, CouchDB, Cassandra, and HBase.

Partitioning

Data partitioning (also known as **sharding**) is a technique to break up a big database (DB) into many smaller parts. It is the process of splitting up a DB/table across multiple machines to improve the manageability, performance, availability and load balancing of an application.

Horizontal partitioning: splitting databases up along ranges of data (zip codes less than 10000 are stored in one instance while those above are stored in another)

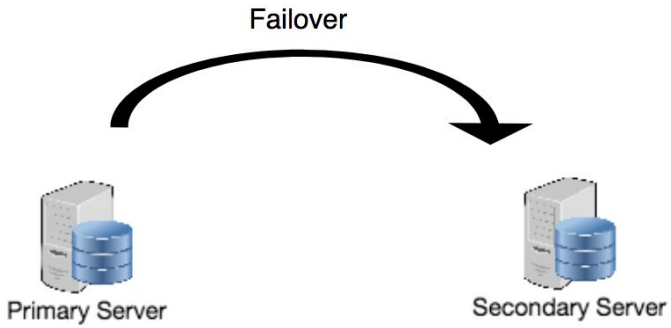
Vertical partitioning: splitting across features (all image and video information is partitioned to one instance while user and follower data is on another)

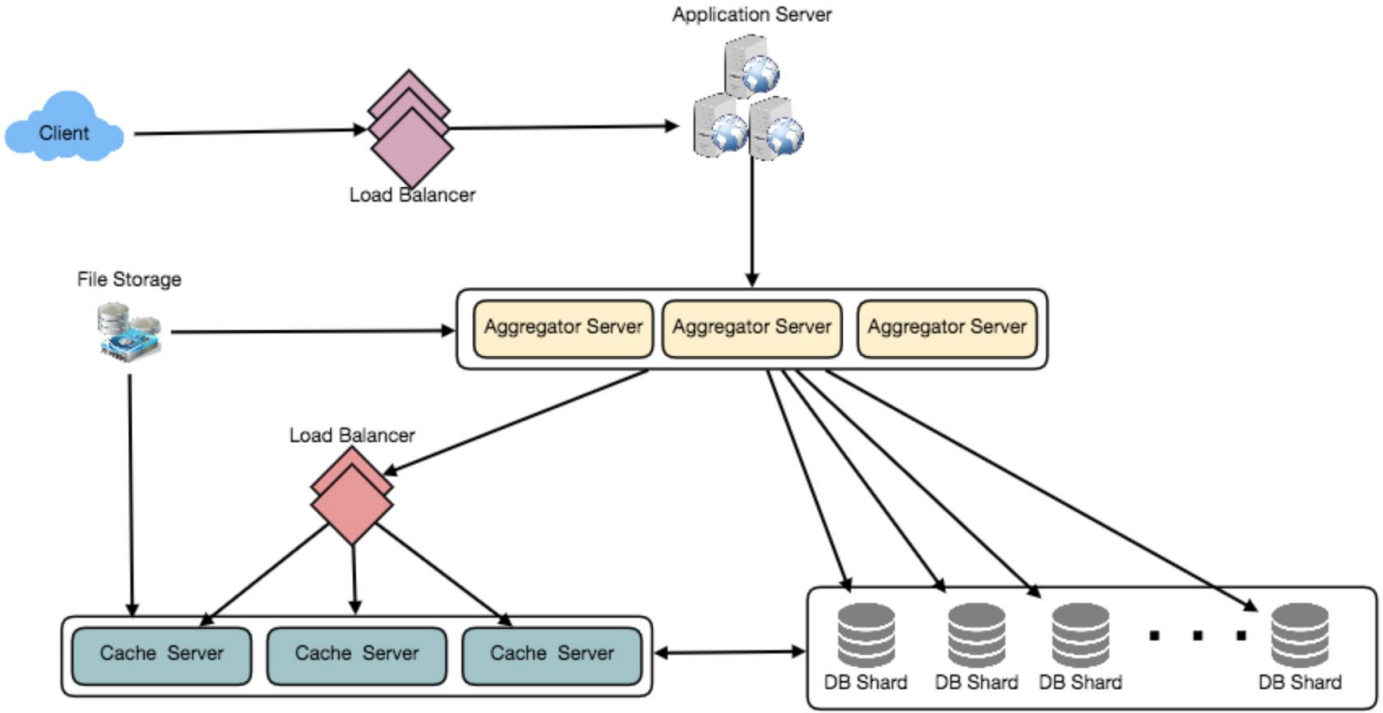
Directory partitioning: A loosely coupled approach that abstracts away the partitioning scheme from the DB access code so the scheme can change without impacting your application.

Partitioning does come with challenges around joins and referential integrity (you can't enforce foreign keys across database instances)

Redundancy and Replication

- Redundancy means duplication of critical data or services with the intention of increased reliability of the system.
- Creating redundancy in a system can **remove single points of failure** and provide **backups** if needed in a crisis.





Flesh out your high level design with your chosen solutions

Resources

8 Things you to know before an SDI

Hired in Tech: System Design

Palantir: How to ace their SDI