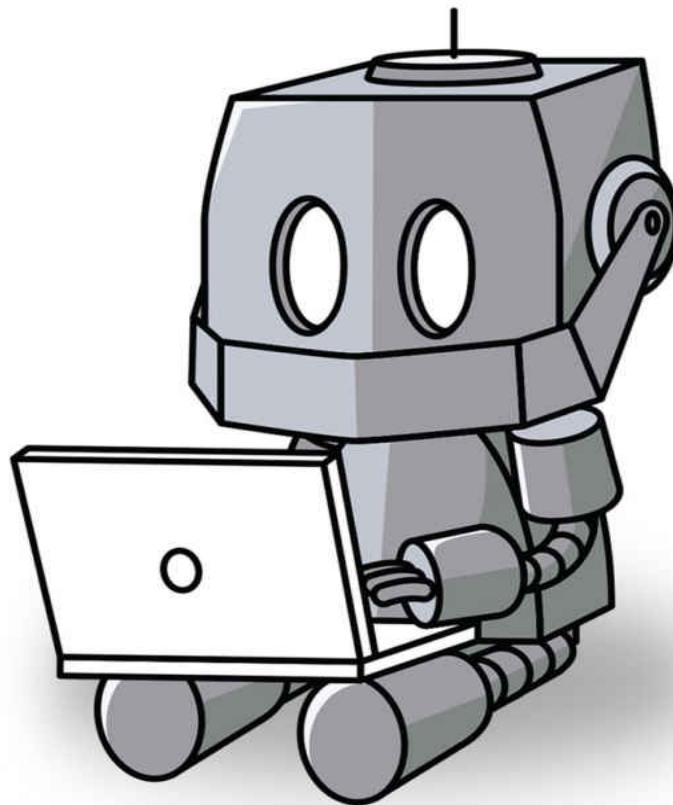


SYSTEM DESIGN INTERVIEW FUNDAMENTALS

SECOND EDITION



Rylan Liu

SYSTEM DESIGN INTERVIEW FUNDAMENTALS

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by an information storage and retrieval system—except by a reviewer who may quote brief passages in a review to be printed in a magazine or newspaper—without permission in writing from the publisher.

Although the author and publisher have made every effort to ensure that the information in this book was correct at press time, the author and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause.

Table Of Contents

Chapter 1: Understanding System Design Interview

Introduction & Goals

System Design Interview Framework

System Design Technical Fundamentals

System Design Interview Questions

Why Should You Care about System Design?

The Purpose of a System Design Interview

How do I Prepare for a System Design Interview?

Chapter 2: System Design Interview Framework

System Design Interview Framework Flow

Step 1: Gather Requirements

Purpose

Functional Requirements

Gathering Features

Clarify Terminologies

Non-Functional Requirements

Scale

How Many Active Users Are There in the System?

How Are the Users Distributed Across the World?

What Are Some Scenarios That Could Lead to High QPS?

What Are Some Scenarios That Could Lead to High Storage and Bandwidth?

Performance Constraints

What is the Availability and Consistency Requirement?

What Is the Accuracy Requirement?

What Is the Response Time and Latency Constraint?

What Is the Freshness Requirement?

What Is the Durability Requirement?

Step 2: Define API

Purpose

How to Define API

Distracting Input Parameters

Missing Input Parameter

Vague and Nonsensical Input Parameter

Ensure the Response Satisfies the Requirement

User Experience with Response Code

User Experience with Response Data Structure

Intuition on Inefficiencies

Vague and Nonsensical Output

Step 3: Define High-Level Diagram Design

Purpose

Position of High-Level Design

How to Approach High-Level Design

Where Do I Start?

Don't Be Distracted with Additional Features

Don't Prematurely Optimize

Table Interesting Discussion Points

Have Logical Separation Between Services

Vague and Nonsensical Design

How Much Detail Should I Provide?

Depth Oriented Questions

Go With the Flow

Step 4: Data Model and Schema

Purpose

How to Approach Data Model and Schema

Add More Detail into the Arrows

[Add More Detail into the Queue](#)
[Add More Detail into the Database](#)
[Add More Detail into Cache and App Servers](#)
[Don't Be Distracted with Additional Features](#)
[Vague and Nonsensical Design](#)

[Step 5: End to End Flow](#)

[Purpose](#)

[Step 6: Deep Dive Design](#)

[Purpose](#)

[How to Approach Deep Dive Design](#)

[How to Come Up with Discussion Points](#)

[API and Interprocess Calls](#)

[Micro Services, Queue, and Databases](#)

[Detailed Algorithm, Data Structure, and Schema](#)

[Concurrency](#)

[Operational Issues and Metrics](#)

[Security Considerations](#)

[How to Frame a Solution Discussion](#)

[Priority of Discussion](#)

[Chapter 3: Interviewer's Perspective](#)

[Rubric Leveling Explained](#)

[Rubric Examples for Each Section](#)

[Functional Requirement Gathering](#)

[Non Functional Requirement Gathering](#)

[API Design](#)

[High-Level Diagram](#)

[Data Structure and Schema](#)

[Deep Dives](#)

[Chapter 4: Communication Tactics and Common Mistakes](#)

Improve Public Speaking

Keep the Requirements Simple

Conclude and Make a Stance

Articulate Your Thoughts

Control the Interview

Listen to the Interviewer

Talk Over the Interviewer

Show Confidence with Your Justifications

Resolving Disagreement

Discussion Point With Trade-Offs

Quality Over Quantity

Math With a Purpose

Focus on the Right Things

Spewing Technical Details With an Intention

Don't Jump Into a Real-World Design

Chapter 5: Technical Toolbox

Back-of-the-Envelope Math

What is the Purpose?

Do I Need to Do Math?

Types of Back-of-the-Envelope Calculations

Back-of-the-Envelope Calculation Techniques

Common Mistakes

API Design

Idempotency

Client vs Server Generated Data

Efficiency

Correctness

Focus on the Core

Schema Design

[Defining Schemas](#)

[Indexing Strategies](#)

[What is Indexing?](#)

[Primary Index](#)

[Secondary Index](#)

[Index Use Cases](#)

[Key or Attribute Lookup](#)

[Range Lookup](#)

[Prefix Search](#)

[Composite Index](#)

[Geo Index](#)

[Databases](#)

[Definition](#)

[Purpose](#)

[How Do I Choose a Database?](#)

[Types of Databases](#)

[Relational Database](#)

[Document Store](#)

[Columnar Store](#)

[Object Store](#)

[Wide Column Store](#)

[Reverse Index Store](#)

[In-Memory Store](#)

[Geo-Spatial Databases](#)

[ZooKeeper](#)

[Real-Time Data Update](#)

[Use Case](#)

[Short Poll](#)

[Long Poll](#)

[Server-Sent Event](#)

[WebSocket](#)

[How to Pick a Protocol](#)

[How to Scale WebSockets](#)

[WebSocket Load Balancer](#)

[Is Connection Still Alive?](#)

[Concurrency Control and Transaction](#)

[Definition](#)

[Purpose and Examples](#)

[Single Thread](#)

[Single Thread Micro Batch](#)

[Partition Into Multiple Serial Processing](#)

[Pessimistic Locking](#)

[Optimistic Locking](#)

[Change Product Requirement to Simplify](#)

[How to Pick a Concurrency Strategy](#)

[Replication](#)

[Definition](#)

[Purpose and Examples](#)

[Leader-Follower Replication](#)

[Leader-Leader Replication](#)

[Leaderless Replication](#)

[How to Pick a Replication Strategy](#)

[What Should Be My Replication Factor?](#)

[Sharding Strategies](#)

[Definition](#)

[Purpose and Examples](#)

[Vertical Sharding](#)

[Horizontal Sharding](#)

[Hash Key](#)
[Range Key](#)
[Other Data Structures](#)
[Outlier Keys](#)
[Geo-Shards](#)
[Sharding Considerations](#)
[Making the Final Recommendation](#)

[Cache Strategies](#)

[Definition](#)
[Purpose and Examples](#)

[Improve Latency](#)
[Improve Throughput](#)
[Improve Bandwidth](#)

[Cache Considerations](#)

[Cache Hit Rate](#)
[What Are You Caching?](#)
[Write to Cache](#)
[Cache Invalidation](#)
[Cache Eviction](#)
[Failure Scenario and Redundancy](#)
[Data Structure](#)
[Thundering Herd](#)

[Asynchronous Processing](#)

[Definition](#)
[Purpose](#)
[Synchronous and Asynchronous](#)

[Reasons for Asynchronous Processing](#)

[The Nature of the Job is Asynchronous](#)
[The Processing Time is Indeterministic](#)

The Processing Time is Long-Running

Improve Perceived Latency

Batch Processing

Purpose

Build Reverse Index for Search

Word Count for Document

Considerations

Stream Processing

Purpose

System is Down

Late and Out of Order Events

Watermark

Checkpointing

Batch Size

Lambda Architecture

Batch vs Stream Processing

Data is Enormous

Compute Intensive

Complexity of Streaming

Use Case

Lambda Architecture

Queue

Message Queue

Publisher Subscriber

Delivery Semantics and Guarantees

Custom Queue

Conflict Resolution

Last Write Wins

Conflict-Free Replicated Data Type (CRDT)

[Keep Records of the Conflict](#)

[Custom Conflict Resolution](#)

[Security](#)

[API Security Considerations](#)

[Man in the Middle Attack](#)

[Authentication](#)

[Timeout](#)

[Exponential Backoff](#)

[Buffering](#)

[Sampling](#)

[ID Generator](#)

[UUID](#)

[Auto Increment](#)

[Auto Increment Multiple Machines](#)

[Strongly Consistent and Fault Tolerant](#)

[Distributed Roughly Sorted ID](#)

[Offline Generation](#)

[Custom ID](#)

[Compression](#)

[Lossless vs Lossy](#)

[Compression Efficiency](#)

[Quality of File](#)

[Computing Time to Compress a File](#)

[Compatibility of Devices](#)

[Compressed File Usage](#)

[Pass Only Needed Data](#)

[Filtering](#)

[Pass Chunk Delta with Rsync](#)

[Fail to Open](#)

Distributed Transaction

Money Transfer

Blob Storage and Metadata Storage

Third Party Data Source

Database and Queue

Cache and Storage Update

Abstract Design Choices

Cold Storage

Networking

IP and Port

Domain Name System (DNS)

How to Route to the Closest Region

OSI Model

API Gateway

Content Delivery Network

Improve Latency

Reduce Bandwidth

Better Availability with Redundancy

CDN Considerations

Monitoring

Latency

QPS

Error Rate

Storage

Metrics Count

Full-Text Search

Text to Token Path

Normalize Step

Tokenize Step

[Remove Stop Words Step](#)

[Stemming](#)

[Indexing](#)

[N-Gram](#)

[Search Query](#)

[Ranking](#)

[Data Source and Indexing Pipeline](#)

[Service Discovery and Request Routing](#)

[Load Balancing](#)

[Shard Discovery](#)

[Product Solutions](#)

[Chapter 6: System Design Questions](#)

[Ridesharing Service](#)

[Step 1: Gather Requirements](#)

[Functional Requirement](#)

[Non-Functional Requirement](#)

[Step 2: Define API](#)

[Step 3: Define High-Level Diagram](#)

[Step 4: Schema and Data Structures](#)

[Step 5: Summarize End to End Flow](#)

[Step 6: Deep Dives](#)

[Driver Location Update](#)

[Location Storage Failure Scenario](#)

[Location Storage Search](#)

[Match Making Concurrency](#)

[Bursty Requests](#)

[Top Watched YouTube Video](#)

[Step 1: Gather Requirements](#)

[Functional Requirement](#)

Non-Functional Requirement

Step 2: Define API

Step 3: Define High-Level Diagram

Step 4: Schema and Data Structures

Step 5: Summarize End to End Flow

Step 6: Deep Dives

Client versus Server Timestamp

Aggregation Service Data Structure

Aggregation Service Scaling Deep Dive

Metrics Storage Deep Dive

Aggregation Service Failures

Late Events

Post Watermark Processing

Emoji Broadcasting

Step 1: Gather Requirements

Functional Requirement

Non-Functional Requirement

Step 2: Define API

Step 3: Define High-Level Diagram

Step 4: Schema and Data Structures

Step 5: Summarize End to End Flow

Step 6: Deep Dives

Fan-out Factor

Connection Storage Complexity

Globally Distributed User Case

Connection Store Design

Load Balance WebSocket

Stream Replay

Instagram

[Step 1: Gather Requirements](#)

[Functional Requirement](#)

[Non-Functional Requirement](#)

[Step 2: Define API](#)

[Step 3: Define High-Level Diagram](#)

[Step 4: Schema and Data Structures](#)

[Step 5: Summarize End to End Flow](#)

[Step 6: Deep Dives](#)

[Optimize for the Feed Read Query](#)

[Duplicate Posts](#)

[Global User Base](#)

[Distributed Transaction](#)

[Feed Ads](#)

[Poor Bandwidth](#)

[Distributed Counter](#)

[Step 1: Gather Requirements](#)

[Functional Requirement](#)

[Non-Functional Requirement](#)

[Step 2: Define API](#)

[Step 3: Define High-Level Diagram](#)

[Step 4: Schema and Data Structures](#)

[Step 5: Summarize End to End Flow](#)

[Step 6: Deep Dives](#)

[Scale for the Write Throughput - Delay Data](#)

[Scale for the Write Throughput - Near Real-Time](#)

[Idempotency of Metrics](#)

[Number of Unique Users](#)

[Scaling for Read](#)

[Cloud File Storage](#)

[Step 1: Gather Requirements](#)

[Functional Requirement](#)

[Non-Functional Requirement](#)

[Step 2: Define API](#)

[Step 3: Define High-Level Diagram](#)

[Step 4: Schema and Data Structures](#)

[Step 5: Summarize End to End Flow](#)

[Step 6: Deep Dives](#)

[Normalized and Denormalized](#)

[Secondary Indexing](#)

[Database Choices](#)

[Out of Sync Files](#)

[File Upload](#)

[Scale It Up](#)

[Folder Deletion](#)

[Rate Limiter](#)

[Step 1: Gather Requirements](#)

[Functional Requirement](#)

[Non-Functional Requirement](#)

[Step 2: Define API](#)

[Step 3: Define High-Level Diagram](#)

[Step 4: Schema and Data Structures](#)

[Step 5: Summarize End to End Flow](#)

[Step 6: Deep Dives](#)

[Rate Limiter Product Design](#)

[Rate Limiter Algorithm](#)

[Rate Limiter Failure Scenario](#)

[Data Storage Long Time Horizon](#)

[Data Storage Short Time Horizon](#)

[Client Facing Latency Sensitive Rate Limiting](#) [Chat Application](#)

[Step 1: Gather Requirements](#)

[Functional Requirement](#)

[Non-Functional Requirement](#)

[Step 2: Define API](#)

[Step 3: Define High-Level Diagram](#)

[Step 4: Schema and Data Structures](#)

[Step 5: Summarize End to End Flow](#)

[Step 6: Deep Dives](#)

[Concurrency of Channel](#)

[Database Solution](#)

[Chat Architecture](#)

[Read Performance Consideration](#)

[Real-Time Protocol](#)

[Database Sharding Strategies](#)

[What's Next](#)

[Appendix](#)

[Appendix 1: Number Conversions](#)

[Appendix 2: Capacity Numbers](#)

[Appendix 3: Math Number Assumptions](#)

[Appendix 4: Product Math Assumptions](#)

[Appendix 5: Advanced Concepts](#)

[HyperLogLog](#)

[Count-Min Sketch](#)

[Collaborative Filtering](#)

[Operational Transform](#)

[Z-Score](#)

[EdgeRank](#)

[Bloom Filter](#)

[TF / IDF](#)

[Page Rank](#)

[Appendix 6: Security Considerations](#)

[Appendix 7: System Design Framework Cheat Sheet](#)

Chapter 1:

Understanding System Design Interview

Introduction & Goals

We are writing this book because we want to help people prepare for a system design interview. Studying for a system design interview is difficult because software engineering is deep and complex. There are a lot of engineering resources, but you only have so much time to prepare. Sometimes even if you understand one component deeply, the interviewer is more interested in something else.

We will not distract you with information that doesn't matter. For example, we will not copy and paste "Numbers Everyone Should Know" from Jeff Dean directly because, in a generalist interview, no one will ask about L1 cache reference being 0.5 ns, since that is on the microprocessor level. However, 150 ms for sending a packet halfway around the world is essential when you're designing a latency-sensitive globally distributed application.

In a system design interview, you're not looking to retell a design that you watched on a tech talk and read about on an engineering blog. Passing a system design interview is about having a solid understanding of the fundamentals of the building blocks and piecing them together using problem-solving skills.

When putting together the building blocks, you may have to dig deeper to address the critical bottlenecks. When you come up with a solution to a bottleneck, you may have made another component worse. Then you need to talk about options and trade-offs. And ultimately, you need to show leadership by making a final recommendation. System design is about understanding the problem you're trying to solve before coming up with a solution. It is not about coming up with a solution and finding a problem to fit into that solution.

Because of that, this book will focus intensely on the system design framework and the fundamental building blocks for system design interviews. We will present how to relate them to system interview questions for each of the fundamental building blocks. For example, instead of telling you an exact clarifying question you should ask the interviewer, we will provide information on the fundamentals of asking good questions. Instead of telling you about how a piece of technology works (you can just search it online), we will focus more on when, where, and why you can apply that technology. Knowing when and why to use the building block comes from experience and is harder to find online, and that's something we will be focusing on.

System Design Interview Framework

The system design interview framework will be long and detailed because it's essential to get it right in a short 45-minute interview. In a 45-minute interview, you spend 5 minutes on the introduction and the last 5 minutes on questions and answers. You have approximately 35 minutes to demonstrate your design ability. It is important to ask the right questions and focus on the right things in such a short amount of time.

System Design Technical Fundamentals

Instead of giving you a bunch of cookie-cutter responses to a system design interview question, we will focus more on the building blocks of the system design. So, instead of telling you how CDNs work, we will try to discuss what situations you should mention and what follow-ups you can talk about with CDNs. That way, you have a better fundamental understanding of the components. In addition, you will be more flexible in applying the fundamentals in similar situations for different problems.

System Design Interview Questions

On top of the fundamentals, we understand it is helpful to provide some guidance on what you can talk about for a given question. Please remember that this isn't the only path you can take in a system design interview. The more important takeaway is discovering the vital discussion points and structuring your answer to impress the interviewer. You should understand

how the problems are solved instead of taking the solution to the perfect answer.

It is self-evident when a candidate has memorized a cookie-cutter solution without even identifying the problem. Whenever a candidate proposes a solution before the problem, the interviewers know they've memorized the solution rather than thinking about the problem, and it doesn't help the interviewers to provide positive data points. When the interviewers dig deeper into the problem they're trying to solve, the candidates fumble to find the problem and discover there wasn't a problem to be solved after all.

Why Should You Care about System Design?

You are probably reading this book because you have a system design interview coming up, and you want to get the job. Before getting discouraged about the amount of information necessary to do well in a system design interview, it is important to keep in mind why you should care about system design.

The more senior you are in software engineering, the more system design matters for an interview and for your job. Even if you are an efficient coding ninja, coding to the wrong architecture will cost the company more money than if you hadn't code. System design interviews are weighted much more heavily the more senior you become. For most well-known companies, staff-level positions require multiple rounds of system designs.

Being good at system design helps you continue to climb the ladder as you're making a more significant impact on the overall business. Being more senior in an organization has a considerable bump in compensation as well. The annual total compensation difference between a senior engineer and a staff engineer can be between 150k and 200k US dollars. That requires you to do well in a system design interview.

Being good at system design allows you to see the overall picture of your company's architecture better. Remember when you were an entry-level engineer confused about some of the technical terms and wondering about

the whys of technical decisions? It can be pretty frustrating. Having the proper fundamentals allows you to better connect the dots and gives you a sense of satisfaction. So invest in yourself now to pass the interview and become a better architect.

The Purpose of a System Design Interview

Before we jump into the perspective of an interviewer, we would like to step back and think about what the purpose of a system design interview is. Companies need to hire experienced engineers and experienced engineers provide value by coming up with reliable architectures.

To come up with reliable architectures, interviewers need to judge if you're able to take a well-defined set of requirements, find solutions, and make difficult decisions when faced with challenging trade-offs. This corresponds to a real-life situation at work where, for example, the product manager may give you some requirements and expect you to clarify the requirements and come up with an architecture to satisfy that product proposal.

The interviewer isn't looking for your ability to recall an architecture you read in a blog. What if the current company wants to offer an Instagram-like feature that is different from Facebook's Instagram with a different user base and query patterns? Doing well requires strong problem-solving skills, technical knowledge, and communication skills.

We will start off by going through our recommended system design interview framework and providing some guidance on what the interviewers might be looking for.

How do I Prepare for a System Design Interview?

Being good at system design is hard. It's a combination of knowledge, experience, creativity, communication, and problem-solving. As you will see throughout the book, a given question may have many possible solutions based on the requirements and assumptions. Even with the same assumptions, it may still be challenging to decide which design option is the best. That's why software engineers debate designs every day, and it's for good reasons.

Now, there are so many topics to learn about in preparation for a system design interview. You might be thinking, *Learning the ins and outs of Paxos already took me two weeks, and I've already forgotten about it*. Don't worry! Unless you're interviewing for a deep distributed system where you may need to implement a consensus protocol, it's unlikely the interviewer will ask you about the ins and outs of Paxos.

Researchers and architects created technologies for a reason. Nobody decided to wake up one day and make Cassandra out of thin air. When you read about a piece of technology, try to understand the big picture of the problem they're trying to solve and why other options were not good enough. This requires some level of critical thinking. Sometimes knowing the internals helps solidify the understanding. However, it's unlikely you will need to talk step by step about how each piece of technology works.

This book tries to strike a balance between too shallow and too deep. We will provide the reasoning behind our design choices for the problem at hand. At times, you may disagree, but that's ok! If a design decision is easy to conclude, it's probably not that interesting. But the value we would like you to get out of this book is to warn you not to look at another blog and try to memorize the solutions, and instead to understand the fundamentals. Know when, where, and why you would apply a solution for a given problem.

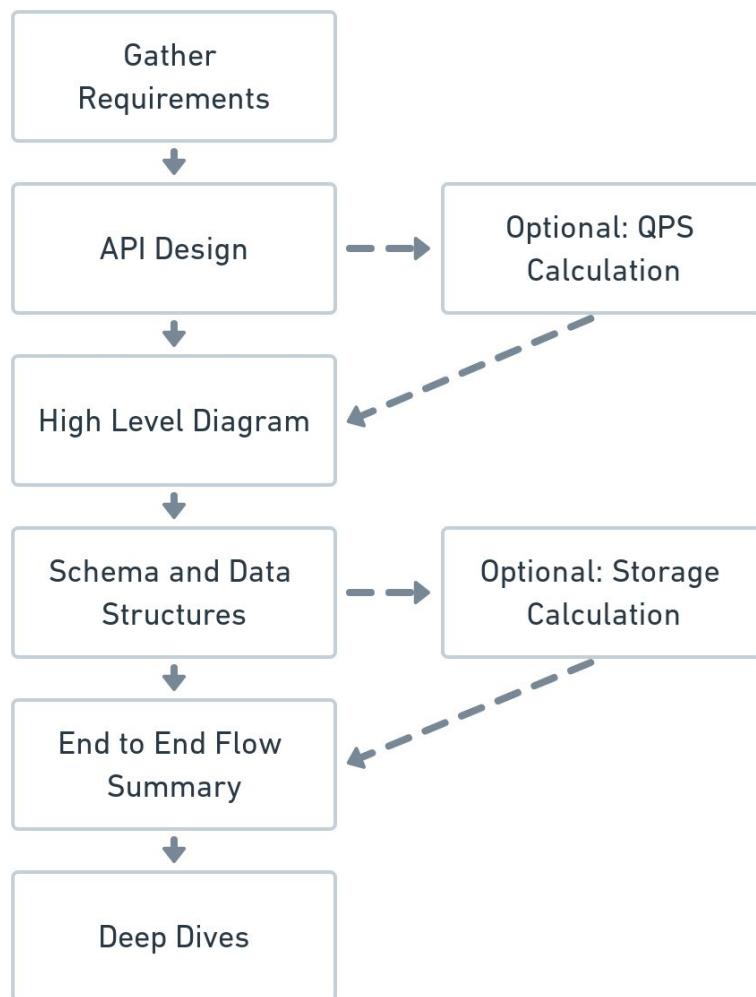
You should go through this book and search on Google if there are concepts and vocabularies that you don't understand. Then, join the discord community to provide feedback and learn from others. Most importantly, schedule a lot of live system design interview practices with experienced and qualified engineers. We can't stress enough the importance of realistic mock interviews since so much of it is based on communication and how you present your arguments. Finally, enjoy and process, since you'll be leveling your design skill outside of the interview context as well. Good luck!

Chapter 2:

System Design Interview Framework

System Design Interview Framework Flow

Having a framework for system design is more important than often believed. An unstructured interview will make the flow difficult for you and the interviewer to follow.



We will dig much deeper into each section but take this chapter seriously as this is what makes system interviews unique from your day-to-day work. Also, you do not want to trivialize sections like requirement gathering, as

they are way more important than you think. For the suggested timing, we will assume it is a 40-minute interview. Here's the reasoning for the flow:

Step 1: Gather Requirements

This flow makes sense because when the interviewer asks a question, you should clarify the requirements and the question to establish the baseline and assumptions for the interview question. Afterward, you want to have a top-down approach starting from the API and working to the bottom to the schema and data structures. If you start from the middle, it's hard to conceptualize the overall big picture.

Functional Requirement Timing

Timing: 2 minutes

The functional requirement gathering phase is just feature gathering. Really try to understand what you're designing for, but don't spend an exorbitant amount of time here.

Non-Functional Requirement Timing

Timing: 3-5 minutes

The non-functional gathering phase is critically important, and these are things that make your design unique and very engineering-focused. You should spend more time here.

Step 2: Design API

Timing: 3-5 minutes

API should be after requirement gathering because APIs are the entry point to your system. After you have the requirements, it should be clear what APIs you will need. If you try to jump into high-level diagrams or schemas right away, it can be confusing because your design might have multiple APIs.

Warning

A lot of candidates have been doing back-of-the-envelope calculations right away after requirements gathering. This is a mistake.

You should do the back-of-the-envelope math after you finalize the API and schemas. Before you start on the math, ask the interviewer if they are interested in the math; maybe they will just tell you the QPS number. Some interviewers are disinterested in spending 5 minutes seeing you do basic algebra.

If you're doing back of the envelope math, you're already implicitly designing for the API and the schema. How can you calculate the QPS without knowing which API the QPS is for? How do you know what storage capacity you're calculating for without knowing the schema?

If you feel like the result QPS calculation will impact your later designs, feel free to do the back of the envelope math after you finalize the APIs. Otherwise, it's preferred to do the math during deep dive, where you can use your math result to *justify* a need to discuss scalability.

Step 3: Define High-Level Diagram

Timing: 5-7 minutes

After you finish designing the API, it makes sense to start fulfilling the APIs by connecting the components.

Step 4: Define Schema and Data Structure

Timing: 5-7 minutes

Now that it is clear which databases, queue, in-memory, and apps you have from the high-level diagram, you can naturally progress to providing more details on the schema and data structures for those components. If you feel like the storage calculation will impact your later designs, feel free to do the math after the schema.

Step 5: End to End Flow

Timing: 2-3 minutes

Before going into deep dives, just take a moment to summarize the flow you have to ensure there is at least a working solution that satisfies the requirements.

Step 6: Discuss Deep Dives

Timing: 15-20 minutes

By this point, you should have a working design. Your job now is to identify bottlenecks or dig deeper into components. What you choose to talk about is very telling about your design sense. You can use back-of-the-envelope math to justify a scalability need. Doing well in the deep dive section will set you apart from other candidates and could lead to a more senior-level position. Ideally, you should find 2 to 3 good solid deep dive discussions, although it's not a hard rule, depending on the context.

Step 1: Gather Requirements

Purpose

The purpose of requirement gathering is to test your ability to clarify an open-ended and ambiguous problem statement. Sometimes the problem statement could be as vague as “Design a ridesharing service.” The interviewers want to see if you can organize your thoughts and focus on a set of requirements. Then you should finalize the system’s assumptions with the interviewers, so you can focus on them for the rest of the session.

Functional Requirements

Gathering Features

During this section, act as a product manager and develop user stories to solve the users’ problems. Even though we’re software engineers, interviewers do look for solid product sense and strong user empathy. When given a question, think about:

1. Who is it for, and why do we need to build it?
2. What are the features we need to solve the users’ problem?

For example, when asked to design a photo storage cloud service, most candidates immediately jump into the functional requirements by coming up with the need to store and view photos. There’s no reasoning behind why the users need to store and view the photos.

Situation
Interviewer: “Please design a file storage system”
Don't Do This
Candidate: “Can I assume the user can save the file into a cloud storage and they should also be able to view the files in a dashboard”?

Interviewer:

“Yeah sure, and the users would like to view their files in a file-system-like dashboard.”

Candidate:

“Ok, we will need to create folders and each folder should have files and folders”

Do This**Candidate:**

“Why are we building file storage? What is the customer problem we’re trying to solve?”

Interviewer:

“The users would like to organize their local files into the cloud with the same folder-like structure. They want to ensure they don’t lose the photos and would like to occasionally browse their files.”

Candidate:

“Ok, to solve that customer pain point, we can have a feature where the user can click and drop a folder and we will recursively upload the folders and files within it. And here’s a wireframe of what I think the end-user should see. We will ensure the system has strong durability”

Analysis

Even though both approaches lead to similar APIs that create and view files, the latter approach demonstrates much stronger user empathy. You’re suggesting features to solve customer problems rather than just coming up with random features that seem related. Because you understand the product, you can leverage it by claiming the importance of the durability of your system since users won’t like to lose their life memories.

For more user-facing products, it's worth spending a little time hashing out the wireframe design the end-users see. The dashboard design will impact your API design.

For more infrastructure and backend problems, still take a moment to think about the end-users' experience. For example, if you're designing a web crawler, you want to ask how important the freshness is to the user. If the crawler is supposed to support a breaking news website, the freshness needs to be a lot more urgent.

After you've come up with a couple of features, narrow them down to just 1 or 2 and get the interviewers' buy-in. In real life, each feature could take months, if not years, to build. If you focus on too many features, the session will just be breadth focused. You should ask which features the interviewer is interested in. Typically, the interviewer will narrow down to a selected few. Don't be distracted!

Do This

"Here are the features I think we can go with to solve the customers' problem:

1. Users can request a ride and we will match them with a nearby driver.
2. Users can view all the nearby drivers.
3. Users can receive estimated payments.
4. Users can pool with other riders.

For now, I will focus on feature #1 and expand further if needed, are you ok with that?"

Clarify Terminologies

After you've come up with 1–2 features to focus on, just take a moment to clarify a level deeper into what it means. For example, if the interviewer asks you to design a feature to show a list of Netflix movie recommendations, take a moment to ask what should be in a

recommendation and the ranking algorithm. Since a recommendation system can take in several signals, you want to make sure you know what signals you need to build the data pipeline for the recommendation system.

If the interviewer asks you to develop the recommendation system, try not to over-complicate the algorithm and model since interviewers are focused on the system, hence the system design interview. Unless this is a machine-learning focused interview where the model and algorithm are the focus, you should understand the interview context and proceed accordingly.

Non-Functional Requirements

After you agree on the features, identify areas that make your design unique and challenging. You should ask questions to build preliminary intuition on what could break your system and, when it does, some areas that can be compromised to scale for those bottleneck challenges. Also, asking good questions gives interviewers good data points about how you think about the design by focusing on what matters.

In this section, you should focus on scale and performance constraints.

Scale

How Many Active Users Are There in the System?

The goal of this question is to figure out the scale of the system. Sometimes you might need to adjust the question to fit the context of the interview design question. For example, if the interviewer asks you to design a web crawler, you might want to ask how many URLs and pages you have to crawl. You can use this number to build the intuition on the scale and calculate the math.

How Are the Users Distributed Across the World?

Globally, different users access applications as the world becomes closer together through the internet. Unfortunately, the latency for cross-globe communication may significantly impact the user experience. Therefore, it's crucial to figure out the distribution to discuss geo locality design in the deep dive section.

What Are Some Scenarios That Could Lead to High QPS?

The goal here is to demonstrate your ability to think of scenarios that can break your system. One example is bursty situations that cause a thundering herd problem. Just as in real life, when there's a design proposal, you look for real-life scenarios that may cause headaches for your system. Not only do you demonstrate experience by considering the worst case, but it also brings up deeper design considerations with interesting trade-offs that will leave the interviewers with stronger impressions. A design without bottlenecks isn't challenging and interesting. You should be the candidate who proactively comes up with difficult cases instead of waiting for the interviewer to provide them.

Examples Questions of QPS Bottleneck

Design a Ridesharing Service

“Should I consider cases where users get out of a concert event and a significant number of people are trying to book for rides?”

Design Facebook Live Streaming

“Should I worry about events like royal wedding where there could be tens of millions of people trying to enter the stream at the same time?”

Design Slack

“Should I worry about supporting a company level chat room that has tens of thousands of employees where there may be a lot of chats?”

Design E-Commerce

“Should I worry about the situation where many buyers try to buy an item at the same time leading to inventory problems?”

What Are Some Scenarios That Could Lead to High Storage and Bandwidth?

The goal here is to figure out what kind of pattern could cause the system to have high storage and bandwidth such that you need to scale the storage system. One factor is the amount of memory passed through a request and how frequently the servers handle the requests.

Examples Questions of Storage Bottleneck

Design a Photo Storage

“Are there super users who upload more photos than an average user?”

Design a Cloud File Storage System

“Are there users who upload large files in the magnitude of GB?”

Design Slack with Image and Video Sharing

“Since slack is used during work hours, can I assume that there will be a lot more images and videos sent during those hours and scale for that?”

Design Netflix

“Can I assume users like to watch movies at night and anticipate bigger traffic during those hours? Can I assume there is only a small subset of popular movies most people like to watch?”

Performance Constraints

Remember, if the world ran on supercomputers with no network latency, computers would never go down, and with unlimited storage, we would not have to worry about distributed systems. However, things do break, and networks do get congested. To achieve the scale that we need, we may have to make sacrifices to our system. You should discuss and agree with the interviewer on the constraints before proceeding further. Here are some high-level idea aspects to consider:

What is the Availability and Consistency Requirement?

The goal here is to discuss whether the system can tune the user product experience’s consistency to meet the availability demand better. A common mistake candidates make is to be too solution-focused, where candidates begin talking about choosing an AP system (CAP Theorem) over the CP system. Focus on the user experience instead.

Situation

Interviewer:

“Please design Facebook Newsfeed”

Don’t Do This

“I will choose AP system over CP system because we need the newsfeed product to be highly available and it is ok to be inconsistent.”

Do This

“When a user posts a feedback, can I assume that it is ok to have some delay before the user’s post shows up in other users’ feed in case I need to optimize for availability?”

Analysis

In the Don’t example, it’s unclear which part of the system should favor availability over consistency. In a system, there could be components that are “CP” and other components that are “AP.” You need to be clear on the use cases. Even if every feature within it should be “AP,” there is much more clarity if you describe the user experience.

What Is the Accuracy Requirement?

Similar to consistency, the goal here is to discuss whether the system can sacrifice accuracy to meet the design constraint better. Accuracy is different from consistency. For example, eventual consistency is eventually accurate where inaccuracy implies the processed data doesn't have to be the same as what the users provided. Yet, the design still offers a good user experience.

For example, you should ask whether or not it is ok for a notification service to drop some messages. Is it ok for rate limiters to be approximately correct? Is it ok for stream processing to consider most of the events but drop the late events? There can be room for creativity here, and coming up with a clever idea will give you bonus points.

What Is the Response Time and Latency Constraint?

The goal here is to figure out how long the users have to wait before receiving the expected data and still deliver a good user experience. Different applications have different response time requirements.

For example, it might be acceptable for an airline booking system to take more than 5–10 seconds, but it is not acceptable to render the Facebook News Feed to take more than 500 ms in p99 of the use cases.

Reminder

People often confuse latency with response time. However, from the end-users' experience, we want the response time, not latency, because that's how long the users have to wait before they get a response.

$$\text{Response Time} = \text{Latency} + \text{Processing Time}$$

Response Time: The time difference between the client sending the request and receiving the response.

Latency: The time the request has to wait before it is processed.

Processing Time: The time it takes to process the request.

Response Time is always more than Latency. However, most APIs are just simple pass-throughs to fetch something light with little processing time and thus Response Time \approx Latency.

What Is the Freshness Requirement?

The goal here is to figure out how data staleness will impact the user experience. Data freshness categories are real-time, near-real-time, and batch processing. Of course, there isn't an exact definition of each category, and the point is to articulate how each impacts the system you're designing. Here are some examples of how freshness is relevant to interview scenarios:

Real-Time

A stock trader sitting at home needs the data to be real-time because a delay in the price feed may cause the trader to make the wrong trading decision.

Near Real-Time

When a celebrity is streaming their experience at a baseball game on Facebook Live, the stream itself should be near real-time, but a couple of seconds delay is acceptable. The users won't feel a difference with a couple of seconds delay, but if the stream is delayed by a couple of hours, some users will likely find out because the game is already over.

Batch Process

Because real-time and near real-time systems are challenging and expensive to build, some applications can still have a good user experience even if it takes a couple of hours or days to run. For example, a static website's web crawler for search does not need to be updated often.

What Is the Durability Requirement?

Here the goal is to figure out how data durability will impact the user experience. In the event of a data loss, how will the user feel? Service level agreement measures durability in 9s. For example, nine 9s will mean 99.99999999%, which means in a year, the storage will lose 0.000000001% of the data. Don't worry too much about the exact number of 9s in the interview, you can just stress the importance of data loss with respect to the end-user.

High Durability

When you're storing users' life photos, it is extremely important to be highly durable. Losing a photo would mean never being able to see that moment of their life again. So, in the interview, you should emphasize the importance of durability design to prevent correlated failures.

Medium Durability

For casual chats that happened years ago, you might argue that the users will rarely look at the ancient chat history again. While the chat history is still important to be durable, losing a message that happened years ago wouldn't get a user super angry.

Low Durability

For a rich sharing service to capture drivers' location, it is acceptable if we lose a driver's location for a moment in time because we will get their location for their next update in the next few seconds. So losing location data wouldn't result in many impacts on the underlying system.

Step 2: Define API

Purpose

The purpose of the API is to have a detailed agreement between the end-user and the backend system. Having a detailed API contract gives you and the interviewer confidence that you are both on the same page with the functional requirements.

How to Define API

API is a very broad term since it could be from a mobile client, web client, internal system, single machine inter-process call, etc. So what you have to define is contextual to the functional requirements.

If the interviewer asks you to design a ride-matching experience for a ridesharing product, you will want to define an API from the rider to your backend system. Likewise, if the interviewer asks you to design a Google Drive, you need to develop an API to render the initial dashboard and the subsequent pages when the user clicks into a folder. So you should make sure you define the wireframe well during the requirement gathering phase.

You will need an API signature, input parameters, and output response to define an API fully. Unfortunately, due to the time pressure of an interview, a lot of candidates forget and miss some input parameters and most forget to define a response. Incorrect and ill-defined APIs make the discussion very hand-wavy and incomplete.



Some of the examples below may look trivial and easy on the surface, but during an interview when under time pressure, many candidates make mistakes. Most candidates don't take a second look to double-check if their API design makes sense. Making mistakes on designs that should be straightforward will raise an interviewer's eyebrow.

API Signature

Once you've identified the scope of the API that you need to define, you need to come up with an API signature that clearly defines what it does. Naming is probably the most important aspect to define clearly in this step. For example, if the interviewer asked you to come up with an API to request for Uber rides, name it “request_ride” instead of “rides” or “fetch,” which don’t articulate what the API does.

Our recommendation is to define an easy-to-read API signature like “request_ride” instead of a specific API technology like REST and protobuf. The interviewer usually isn’t interested in RESTful design so avoid elaborating on this unless they explicitly ask for it. Sometimes we’ve seen candidates spend 7+ minutes on just a RESTful design. Of course, if you sense the interviewer wants a specific technology, go with the flow, but in practice, not too many interviewers care for that.

Input Parameter

The biggest warning we have for this section is that everything counts. If you decide to introduce an input parameter because it kind of makes sense, you need to be able to justify it if the interviewer asks you about it. The converse is also true; if you forget an input parameter, the interviewer might come back and question how you’re going to achieve a functional requirement without certain information from the client. Of course, if the input doesn’t make sense, you’re also going to be questioned about it.

Distracting Input Parameters

You should not introduce input parameters that are not important to the functional requirements. If you give the interviewer more input parameters than needed, it doesn’t mean it is better; it is harmful in an interview since it’s unfocused and distracting.

Situation
After the functional requirement gathering phase, the interviewer just wants you to focus on matching a single rider request with a driver, and

that's it.

Don't Do This

request_ride(user_id, pickup_location, destination, car_type, is_carpool, payment_method)

Do This

request_ride(user_id, pickup_location, destination)

Analysis

Since the agreement with the interviewer is to focus purely on the core functionality of matching the ride and the driver, don't start introducing a bunch of functional features like different levels of car type while you're designing the API. Introducing a bunch of random inputs can be very distracting to you and the interviewer.

If you feel like the feature is essential to talk about and feel like you forgot to bring it up during the functional requirement gathering phase, feel free to ask about it. Ask, "Should I worry about different levels of car types" before directly putting the parameter in the API.

However, you should try to minimize the feature clarification questions past the requirement gathering phase. If you ask functional requirement questions during design can be distracting for the interviewer and yourself.

Missing Input Parameter

Your API input parameters need to capture the functional requirements you're trying to design. Missing input parameters will make the interviewer think you're careless. Candidates miss inputs more often than they should. After you've defined the API, take a moment to ensure it satisfies your functional requirements.

Here are some example mistakes:

API Design Goal	API	Issue
Upload photo to the server	upload(user_id, folder_id)	You're missing the photo bytes to be stored.
Request ride for Uber	request_ride(user_id, destination_address)	The system wouldn't know where to pick the user up.
Get newsfeed for a user	get_feeds()	You forgot the user you're fetching the feeds for.

Vague and Nonsensical Input Parameter

This section is beginning to test your technical skill and is not a situation where you can simply slip through the cracks without a solid fundamental understanding of design. Hopefully, through examples, you will develop a stronger intuition.

Vague Input

When you design an API input, it's necessary to clarify inputs that the interviewer can interpret in many ways. For example, consider book_calendar(user_id, starting_time, ending_time)—the granularity of time is critical for the end design. If you forgot to discuss the time granularity during functional requirements, take a moment to clarify with the interviewer.

Nonsensical Input

Similar to missing input, you need to make sure the input works and makes sense. Just because you put something in the input parameter doesn't mean the interviewer won't look at it.

API Design Goal	API	Issue
Upload photo to	upload(user_id, photo_bytes,	Usually, the client shouldn't be the one defining the photo_id, the server should

the server	photo_id, folder_id)	be the one doing so.
Add a friend on Facebook	add_friend(user_id, friend_name)	The friend's name is ambiguous since people can have the same name. friend_user_id makes more sense here.
Send an email	send_email(user_id, to_user_id, message, timestamp)	Instead of the client passing in the timestamp, the server can just generate the timestamp for you.

Output Response

Similar to the input parameters, the level of detail for the response is just as important, and everything counts. Unfortunately, from our interview experiences, most candidates do not define the output. The missing output makes the contract very confusing and open for interpretation. Like the input parameters, you shouldn't include extraneous details irrelevant to the functional requirement or miss important responses necessary for the functional requirement.

Ensure the Response Satisfies the Requirement

Make sure your response includes enough data for the requirement you're designing. For example, if you're designing Google Drive and agree with the interviewer that folders should show up first, followed by the files, each list is sorted alphabetically. You need to ensure the response includes both the folders and files sorted alphabetically.

User Experience with Response Code

Usually, APIs will return some sort of response code depending on whether the server can handle the request. It is worth using the HTTP response code as a reference, but it's unnecessary to map your design to a specific HTTP response code.

However, it is important to clearly define the user expectation for a response code you have specified. For example, if you are designing a request_ride API and the server returns a SUCCESS code, what does

SUCCESS mean? For instance, it might be a success if the server has received your request and is processing the request, or if the server has fully processed the request and is letting the user know which driver will pick them up.

User Experience with Response Data Structure

You should provide enough details about the data structure. For example, the following two APIs look very similar but can be a very different experience and lead to significantly different designs:

```
request_ride(user_id, pickup_location, destination) → { driver_ids: [1, 2] }  
request_ride(user_id, pickup_location, destination) → { driver_id: 1 }
```

The first one allows the user to choose a driver, whereas the second assigns the driver to the rider. For the first one, you may have to worry about the concurrency when multiple users pick the exact driver simultaneously. Take a moment to look at your data structure and ask yourself why you need the data and how you can use the data and make sense of the requirements.

Intuition on Inefficiencies

When you think of the output response, try to identify inefficiencies of the data structure, especially collections. For example, if you're designing for News Feed, the response will usually be a list of feeds. You should develop the intuition that a list can be long and lead to a slow and impossible query when a user has a long feed list. If a user is fetching a list of photos back, try to think about pagination and thumbnails to identify the inefficiencies of the query.

Vague and Nonsensical Output

Similar to the input discussion, detail matters here. For example, if you're designing Yelp and are querying for a list of points of interest, if the requirement is that the user can click into a point-of-interest for more details, you will need to return the point-of-interest ID such that you can send the server using that ID to fetch for more information. If you had designed it by returning a list of point-of-interest string names, it wouldn't make sense.

Step 3: Define High-Level Diagram Design

Purpose

The high-level diagram design aims to set up the foundations for the design and give clarity to the interviewer on which parts are important to have to achieve the requirements. In addition, having a high-level diagram gives you and the interviewer confidence that there is at least an end-to-end flow that satisfies the requirements.

Position of High-Level Design

We believe high-level diagram design should go after the API design because the API is the beginning of the high-level diagram. We don't recommend schema design after API because you don't know how many data stores you potentially need. For example, you may need to store the ride record and driver location to design a ridesharing service. It will be a lot more apparent once you figure out the flow diagram.

How to Approach High-Level Design

Where Do I Start?

You need to introduce a couple of microservices for a given system design question, and the fear is missing some of the components. Therefore, we recommend starting from the top with the API and working yourself down to the last component.

Step 1: Define the Client for an API

For a given API, think about who is making that API call and draw a representation of the caller. The caller could be a box, computer, or human figure, whichever is your favorite choice.

Step 2: Define the Next Sets of Logical Blocks

From the end-user, think about what the next logical block worth discussing is. Most of the time, that may be an API gateway to take in the user request. Then the API gateway forwards the request to an App server, and the App server sends a write query to the database. API gateway isn't always

required, however put it in if you feel more comfortable. Continue down the flow until the architecture thoroughly handles the API request.

Step 3: Repeat Step 1 for the Next API

After you complete the first API, you should move on to the next API. The next API should interact with the system you've designed from step 1 and step 2.

Example

In this example, we will assume you're designing for a ridesharing service to match riders with the drivers. The agreed-upon APIs in the API section are:

```
request_ride(user_id, pickup_location, destination)  
update_driver_location(user_id, current_location)
```

Step 1: Define the Client for an API

Rider

Step 2: Define the Next Sets of Logical Blocks

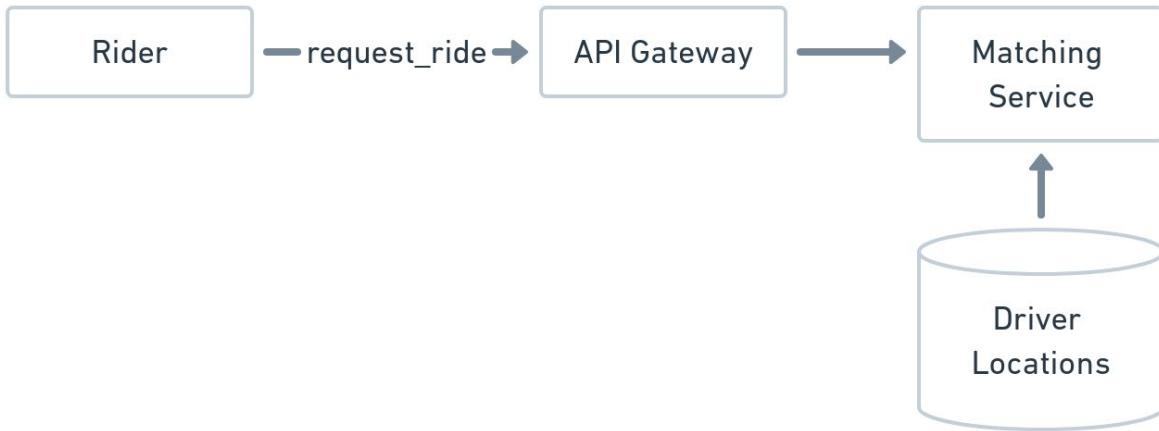
2.1: Request_ride API is fronted by the API Gateway

Rider

—request_ride→

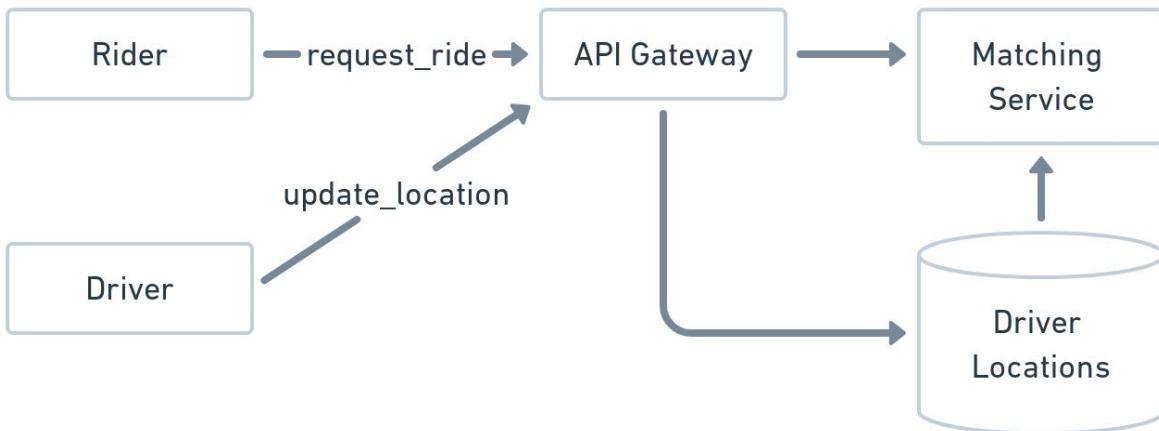
API Gateway

2.2: API Gateway proceeds to call the ride matching service which gets the driver location data from the driver location store.



Step 3: Repeat Step 1 for the Next API

Handle update_location for the driver and connect it with the previous diagram.



This API-driven high-level diagram design may seem obvious, but many candidates still make mistakes because they don't have a systematic approach. Some candidates simply list out the microservices. It is easy to forget a microservice if you are listing them out.

Other candidates mistake jumping into the “main” microservice and try to connect the diagrams. Sometimes they fail to identify the end-users and describe the end-to-end flow.

In the ridesharing example, you might also have many concerns, thinking this isn't enough information and is not deep enough. That is ok. The goal is to keep it simple and to have a holistic view of the system first. After you

accomplish the high-level diagram quickly and accurately, the next step is to proceed deeper with schema design and the deep dive.

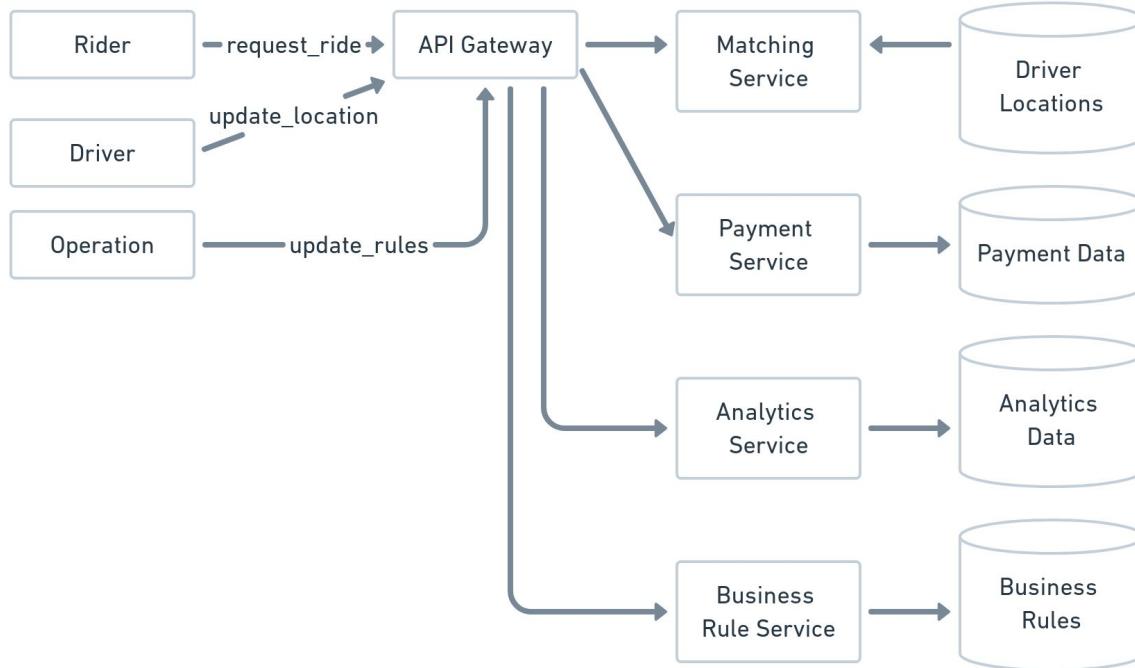
Don't Be Distracted with Additional Features

Similar to API design, don't start adding in a bunch of boxes that are unrelated to the core of the problem. There are two types of irrelevant information. One is designs that are not unique to the specific situation. For example, drawing a DNS server and authentication server is distracting since it's not the main focus of the question. If you think adding those components is ok, then the question is, how far should you go? Should you also design a logging system, deployment system, and rate limiter as well? Again, you should keep the discussion focused on the core of the question.

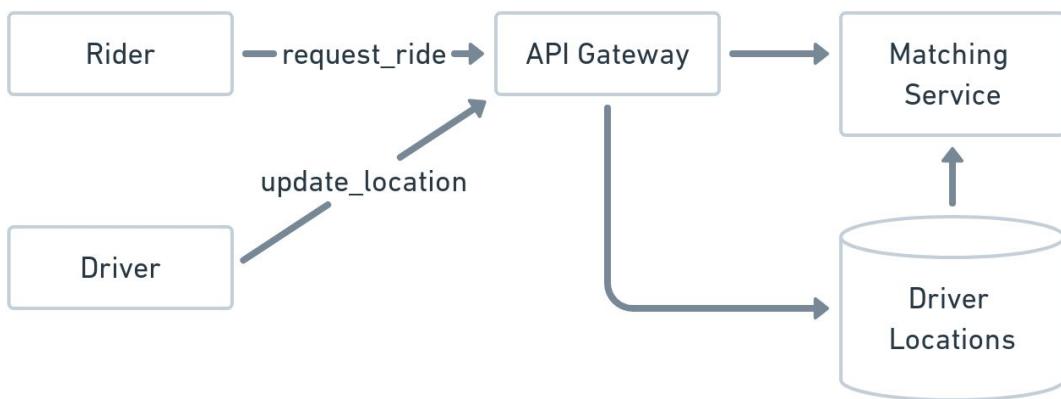
The other type of distraction is introducing boxes and arrows for additional features. For example, in the ridesharing design example, if the only requirement is to match a rider and a driver, don't start adding boxes like payment service, estimated time service, and fraud service, even though they are necessary to "the ridesharing design."

If you are unsure, you can ask the interviewer, "Should I worry about the payment system?" But just a warning, it can be distracting and annoying if you do this too often since you already agreed to the feature set only minutes ago.

Situation
You agreed with the interviewer to focus on just matching a rider with a driver.
Don't Do This



Do This



Analysis

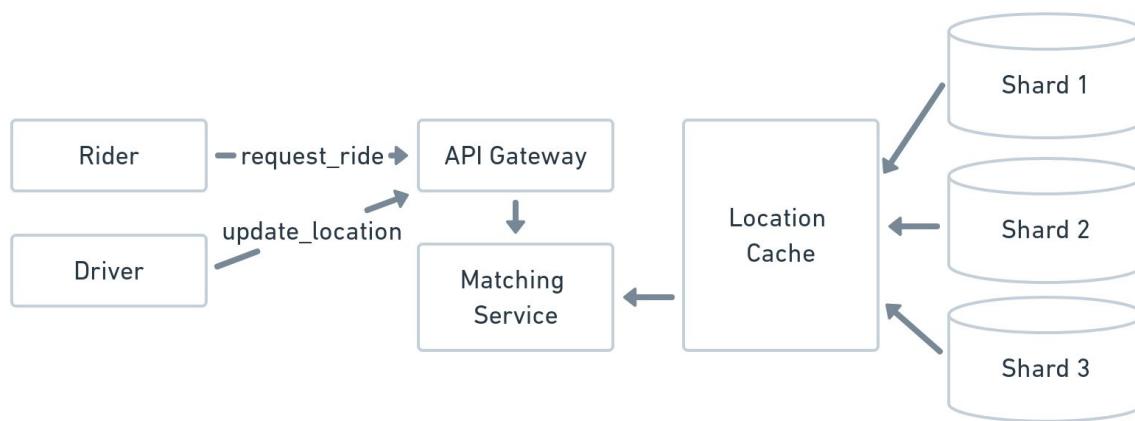
Don't introduce a bunch of boxes unrelated to the main feature. Focus only on the components that matter. Put yourself in the interviewer's shoes and think about what kind of signals they will be writing for you in the Don't section. Do you think they will say, "wow, they surely know a ton about designing a rider sharing service with all these services?" or "the candidate is asked to match riders and drivers, but they kept adding new boxes unrelated to what we agreed on"?

Don't Prematurely Optimize

The objective of the high-level diagram design is to get a flow diagram out in the interview as the baseline for further discussions. Many candidates feel obligated to include optimizations because they are something they've seen on a blog or they're trying hard to impress the interviewer.

For example, some candidates like to introduce cache layers and sharding while going over the high-level diagram. Because the goal is to finish the high-level diagram, candidates who do this tend to be very hand-wavy with the justifications. To introduce cache layers or to shard requires solid justifications for the additional complexity. If you're going to introduce complexity, please take a moment to think about the problem you're trying to solve. Otherwise you're at risk of digging yourself a hole without the complete high-level diagram picture and schema. To avoid digging yourself a hole, we suggest keeping things simple first, then optimizing later during the deep dive section.

Situation
The candidate is drawing out the high-level diagram.
Don't Do This



Candidate:
"I would introduce a cache between the database and the matching service and I would shard the database because we will have a ton of traffic."
Analysis

It is unclear why the candidate even needs the cache and shard. The candidate didn't provide any justification other than "A ton of traffic." The candidate hasn't hashed out the database schema yet, then how would the candidate even know how to shard the database? If the candidate introduces a cache, why are the cache in-between ride-matching service and the database cluster, are there other options? What is in the cache? Why do we even need a cache?

We don't just throw in a box on the cache cluster at work because it's a generic solution. Typically we identify there's a problem first, then come up with detailed solutions and trade-offs. Unfortunately, premature optimization slows down the rest of the high-level diagram and schema design and makes the whole discussion very hand-wavy.

Table Interesting Discussion Points

To continue with "Don't Prematurely Optimize," you might be tempted to dig into a section and prematurely optimize without completing the holistic picture first. One tactic you can do is to maintain a list of discussion points. If it's on a physical whiteboard, you can write it to the side. If it is a virtual interview, you can maintain that list somewhere on the document. Keeping a list of discussion points helps the interviewer perceive you as someone who stays focused and is technically aware of bringing up the important discussion points. Coming up with important discussion points is a critical part of doing well in a system design interview.

Caveat: Sometimes the discussion may just be a short one, and it's ok to address it. The purpose is to not dig yourself a hole when you don't have the full picture yet. Also, if the interviewer seems interested in exploring deeper right away, you should go with the flow.

Situation
The candidate is drawing up the high-level diagrams and is tempted to dig into a section
Do This

Candidate:

“I feel like we might need a queue to handle the rider request because the traffic can be bursty, let me come back to this after I’m done with the overall”

Candidate:

“I feel like the driver location update is very frequent, I’m not sure if I should hit the disk first or need a persistent store at all.”

Candidate:

“I feel like there are various algorithms for ride matching services that would impact the overall design.”

Candidate:

“I need to think about how to efficiently query for a list of drivers in a given geo region.”

Candidate:

“I feel like there might be some concurrency issues with the driver and rider matching, let’s dig this deeper.”

To Be Discussed Later:

1. Queue to handle bursty traffic
2. Data store for driver location
3. Ride matching algorithm
4. Think about indexing solution for driver geo locations
5. Concurrency problem with ride matching

Candidate:

“So, here’s my high-level diagram with the happy use cases. Is there one area you would like me to dig deeper into in this list?”

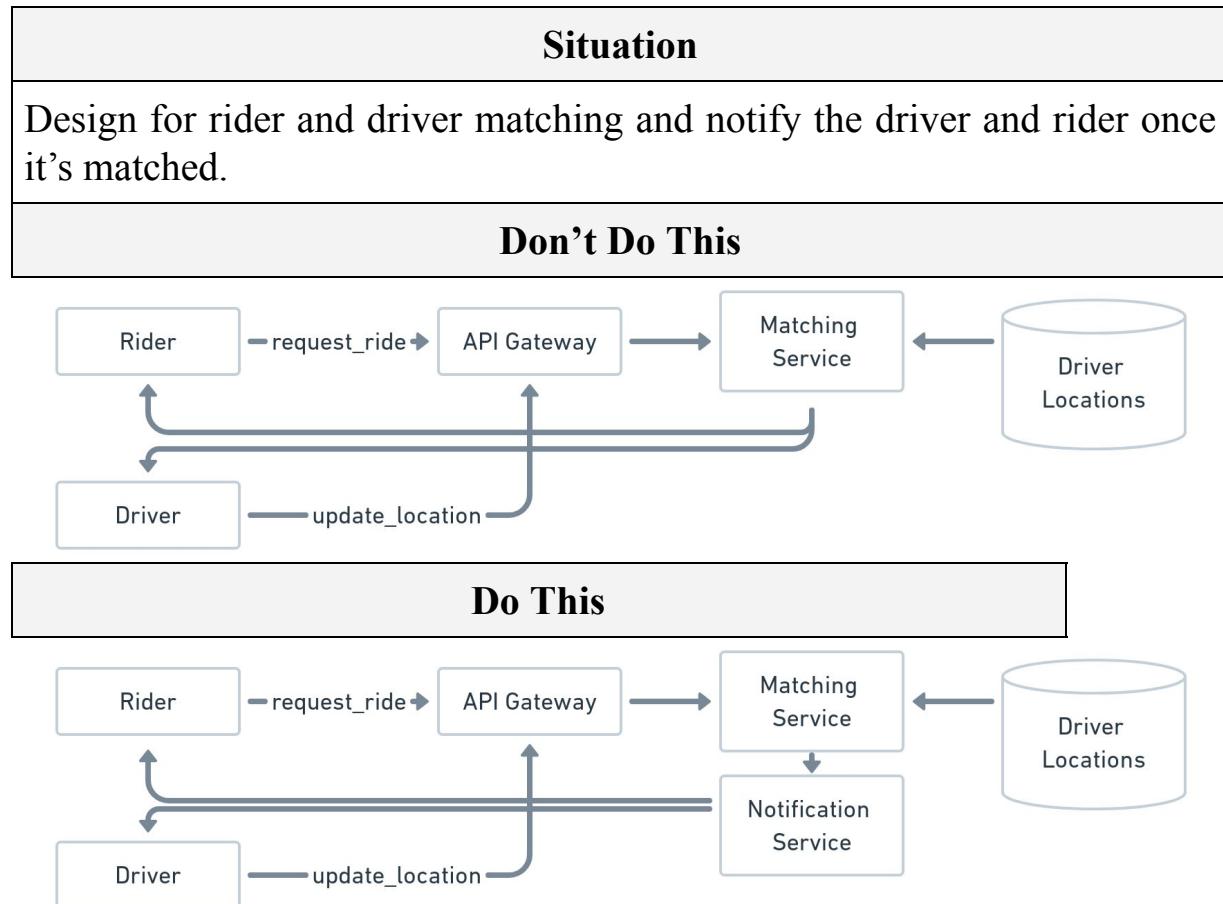
Analysis

The candidate doesn’t get distracted from completing the high-level diagram and maintains a list of discussion points by staying focused. As a

result, the candidate was able to make progress and identify a list of important discussion points to impress the interviewer.

Have Logical Separation Between Services

The purpose of the high-level diagram is to bring clarity to the discussion, and you want to be clear on the responsibility of microservice boxes. Continuing to use the ridesharing example, let's say the system matched a rider with a driver, then the system should notify the rider and the driver about the match. Thus, one responsibility is to match the rider and the driver, and the other responsibility is to notify them, therefore, you should separate them as two separate logical boxes for clarity. Even though in practice, you may decide to host both logical services on the same physical machine.



The suggestion might sound contradictory to “*Don’t Be Distracted with Additional Features*,” but it’s not because the interviewer asked you to

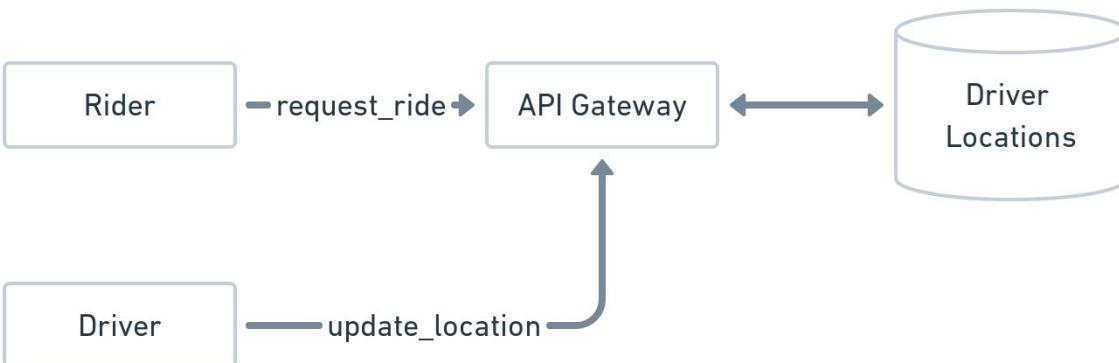
design a notification system here. Even though the Don't Do This example could technically work in an interview, as you add more components, it becomes more difficult to follow the API flow when services have multiple responsibilities.

Vague and Nonsensical Design

As obvious as it sounds, your design needs to work and make sense. If there are things that don't make sense, there are most likely technical gaps with your understanding. Hopefully, the fundamental chapter will provide the right foundations there.

If there's a flow that doesn't make sense, it won't look good on you during the interview. After you've designed the high-level diagram, take a moment to walk through the flow of the defined APIs to ensure the requirement is satisfied. Unfortunately, this happens more often than it should, and most of the time, it's because candidates don't take a moment to check their work. Hence the feedback of, "I feel like the candidate required some hints and guidance to discover there were issues with their design."

Situation
You agreed with the interviewer to focus on just matching a rider with a driver.
Don't Do This



Analysis
To match the rider and the driver, you need another service to take in a list

of drivers with the rider quest to come up with a recommendation. Here, there's just a database with locations. The database isn't going to make that decision for you unless you have some database that handles the matching business logic. Nonsensical design is usually a result of not thinking through step by step thoroughly. Take a moment to make sure your diagram provides the right level of details for clarity.

How Much Detail Should I Provide?

We often get the question of, "How much detail should I provide?" even if it's focused on the core of the question. For example, in a ridesharing design, you will most likely need DNS, load balancer, fiber cables, virtual machines, logging, monitor, and so on. We recommend not worrying about those levels of detail since they are generic across all design questions. We suggest only bringing it up if it's unique to the question. For example, a notification system may require some sort of open connection like WebSocket, and there are challenges around maintaining that. In that case, it's worth investing a bit more time into that. If it makes you feel more comfortable to mention some of the details, you can lightly touch on them.

Situation
You want to mention a service cluster must be fronted behind a load balancer.
Don't Do This
Candidate: "I would have redundancy for my load balancer behind a cluster of app servers so if the first load balancer fails, load balancer 2 will recognize that and begin taking in traffic. For my load balancer, I can do this in a round robin fashion or do it in a stateful way where we map a user_id to a given set of app servers. The load balancers will heartbeat each other and check if the other load balancers are still alive."
Do This



Candidate:

"I would have a load balancer behind my app servers and the app server will persist that information into disk."

Analysis

Similar to "*Don't Be Distracted with Additional Features*," you just invested a bunch of time into describing a generic load balancer that has nothing to do with the core of the problem. Unless the load-balancer design is critical to the feature you're designing for, don't invest so much time into something unrelated. Time is valuable.

Depth Oriented Questions

Sometimes there will be questions that are more focused on the depth of a particular requirement. For example, an API rate limiter wouldn't have too many high-level diagrams to talk about right from the start. That is fine, finish up the bare high-level diagram and then go deep into some areas worth discussing.

Go With the Flow

As you're drawing out the high-level diagram during the interview, the interviewer may interject and have you explain further. In that case, try to understand the context of the interjection and if they're just looking for a quick answer to a question or if they want you to dig deeper. Generally, you should go with the interviewer's flow, but in most system design sessions, especially if you're technically strong, you will be driving most of the discussion, and staying focused and organized will strengthen your presentation.

Step 4: Data Model and Schema

Purpose

The data model and schema can potentially significantly impact your performance and end design. If you don't provide detailed data models, the interviewer can perceive the discussions as too hand-wavy.

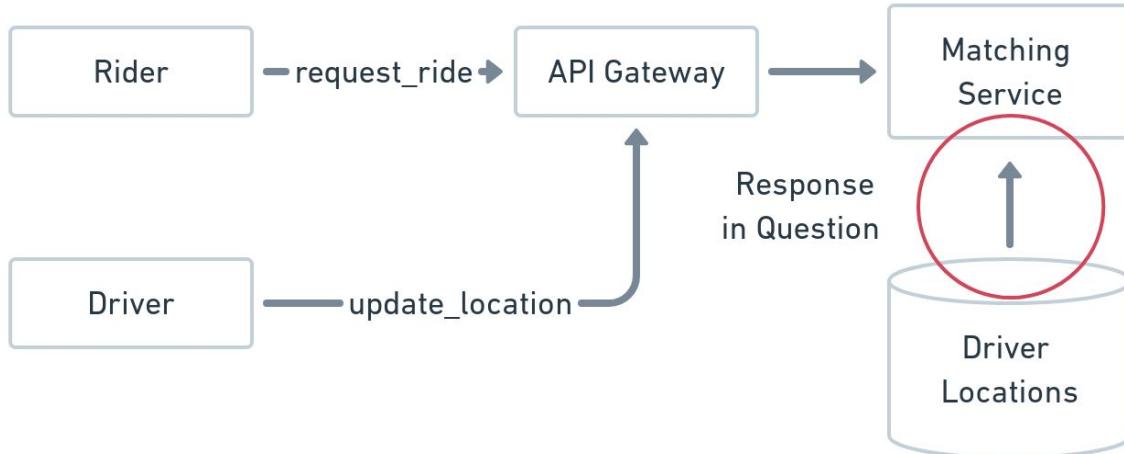
How to Approach Data Model and Schema

Hopefully, you would already have the high-level diagram thoroughly hashed out at this stage, and you can take a look at the diagram and see where you can add clarity. Most candidates will look at their diagram with boxes and arrows and avoid details because they don't want to dig themselves a hole. Strong candidates will try to find opportunities to provide more color into important areas. Here are a couple of ideas to add clarity.

Add More Detail into the Arrows

When you look at your high-level diagram, there will be a bunch of boxes and arrows. Some arrows are pretty evident as to what they do. For example, if a blob store has a key of video_id and a value of bytes, the arrow is most likely just a key-value lookup for video bytes. What if the arrow is from a ridesharing match service to a location service? Let's look at an example:

Situation
You've just finished the high-level diagram for ride-matching service where the primary requirement to design for is matching riders with drivers.



Options
<i>Option 1: A unsorted list of driver IDs</i> <i>Option 2: A list of driver IDs with score attached to each driver</i> <i>Option 3: A single driver ID</i>
Analysis
<p>All three options are equally reasonable. If it's Option 1, you'll need to discuss how you will rank the drivers in the ride-matching service. If it's Option 2, you'll need to discuss how the ranking score data gets persisted to the driver location database. For Option 1 and 2, you need to discuss why you need a list and some concurrency issues if multiple riders are getting the same list of drivers.</p> <p>As you can see, detail matters and significantly changes how the system handles the design. If you don't provide clarity in areas that matter, the interviewer will perceive you as hand-wavy. By proactively bringing them up, it gives the interviewer you care about detail and, if you explain well, provides the interviewer with solid signals.</p>

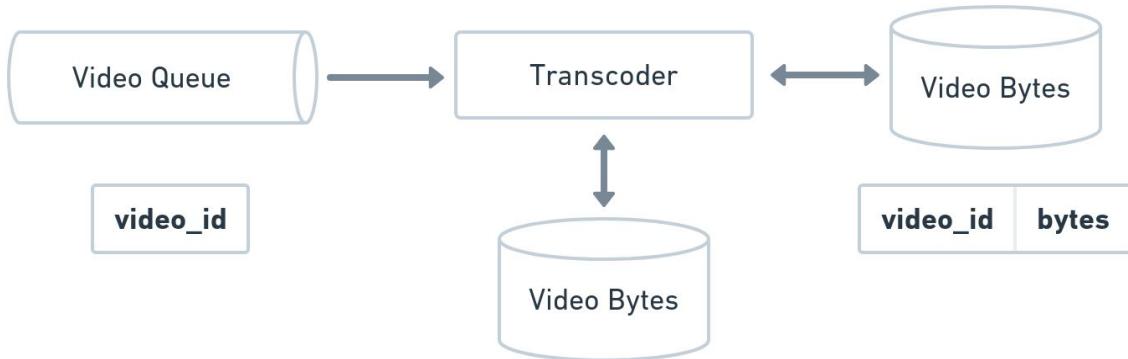
Add More Detail into the Queue

If there's a queue, it might be worth talking about what is inside of the queue. For example, if you are designing a system to transcode a video into various formats through a data pipeline, it's important to talk about what goes into the queue.

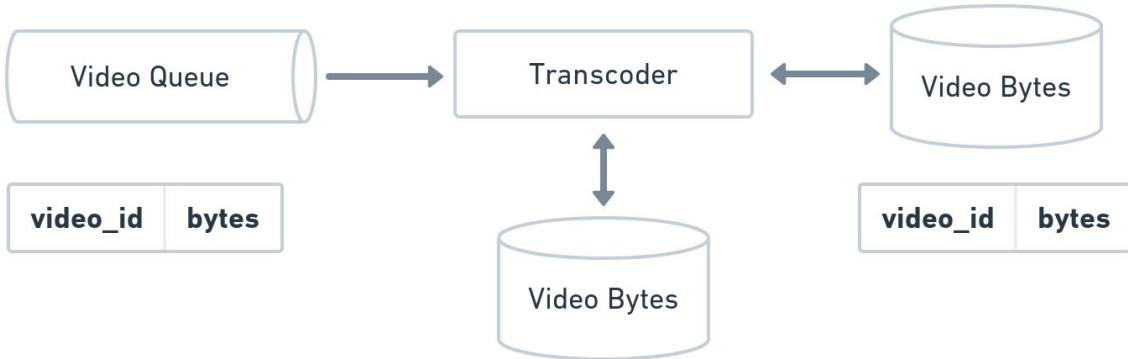
Situation

The interviewer asked you to design the data pipeline for transcoding RAW video formats for uploaded videos.

Option 1



Option 2



Analysis

Most of the candidates will simply say, “I will pass in the video into the queue, and the transcoder service will take the request and start to transcode,” without really providing detail to what is inside of the queue. As you can see, Option 1 and Option differ significantly in performance, where Option 2 may quickly exhaust the memory by unnecessarily enqueueing the video bytes into the queue.

By saying, “I will only insert the raw video id, and the transcoder can pick up the bytes from the Video Bytes blob store and transcode to save the

transcoded videos back into the video bytes store with the new video id of another transcode. Designs like this give the interviewer a positive signal into how you're looking to optimize.

Add More Detail into the Database

When you draw a database diagram, don't just leave it there and assume the database stores everything you need efficiently. You should take a moment and design the database schema you need to satisfy the main requirements. It sets up the foundation for future discussions like sharding, indexing, and caching. Without the data model and schema, the discussion will become very hand-wavy and unproductive.

Situation
You're asked to design a database schema to store driver location.
Option 1
Driver Location Table driver_id x-coordinate y-coordinate
Option 2
Driver Location Table driver_id x-coordinate y-coordinate
Driver Location Index position_id -> driver_ids
Analysis
A lot of candidates just give Option 1 without discussing the efficiency of the query. By specifying the schema, the interviewer will know how to think about the query. For example, if the candidate doesn't recognize they need a full table to find a matching driver, it doesn't look great. Without the schema, the efficiency discussion will not happen, so take a moment to discuss the schema and its efficiency.

Add More Detail into Cache and App Servers

Similar to databases, queues, cache, or any data stores, it is important to discuss the data structure and data model because it will affect the ultimate design. So just make sure you check any app servers you introduced that hold temporary data or any distributed cache you introduced in your high-level design. Make sure you present the necessary detail if it's essential to the efficiency of your design.

Don't Be Distracted with Additional Features

Similar to API and high-level diagrams, don't start adding details unrelated to the core of the question. Adding unnecessary details commonly happens during schema design where candidates start adding in many generic columns that aren't helpful to designing the central part of the question. Here is one example:

Situation
Design the schema for the ride record to display the status of the ride to the rider and driver.
Don't Do This
User Table User ID Name Birthdate Birthplace Address Joined Date
Rider Table User ID Payment Method Number of Rides Taken
Driver Table User ID Rating
Ride Table Ride ID Rider ID Driver ID Status Initiated Time Estimated Cost Starting Location Ending Location Fraud Status
Do This
Ride Table Rider ID Rider ID Driver ID Status

Analysis

To determine a given ride's status, you don't need the estimated cost and number of rides taken for a rider. Those are just superfluous information that has nothing to do with the central requirements agreed with the interviewer. Focus on columns and schema that are necessary for the problem at hand.

Table Interesting Discussion Points

Similar to high-level design, while you're designing for the data models and schema, you might have some intuition to optimize for the design at hand. If you can convert the discussion quickly, it makes sense to get it out of the way. If the discussion is lengthy, include the points in the discussion points list for the deep dive section. The point is not to unnecessarily complicate things.

For example, if you have two tables—Ride Table and User Table—during your design, if you feel that the driver and rider name are often queried together with the ride, it might make sense to denormalize the tables together to prevent joins. This discussion should be relatively quick.

However, you might look at a database schema and have all sorts of ideas on sharding the database. You should table lengthy discussions like sharding the database for later if you haven't finished the end-to-end design yet.

Another common mistake is digging into the way too common MySQL vs. NoSQL debate. The discussion is more involved than just "NoSQL is more scalable." Since most candidates don't know the internals of databases well, they usually spend a quick second claiming they will use NoSQL while designing for database schema. You can also table it by saying, "I will lay out the logical schema but come back to the internal later." If you're confident, it's also reasonable to deep dive into a database choice here—just make sure all the write and read queries into that database are fully

designed. Otherwise, you can't justify well without knowing the query patterns of the database.

Vague and Nonsensical Design

The same thing as with API and high-level design, your data model and schema need to make sense and include all the details required to satisfy the requirements. Ensuring correct and efficient schema does require technical fundamentals. Once you have finished with the data model and schema design, take a moment to double-check the work to ensure the schemas and data structures work.

Situation
Design the data model for group chat like Slack.
Don't Do This
Message Table
User ID Receiver ID Message Timestamp
Analysis
The schema is a real-life example where the candidate makes the mistake of not realizing a user and a receiver can belong in multiple chat rooms. Mistakes like this happen more often than they should but will often give a poor impression to the interviewer. Take a moment to double-check your work before moving on.

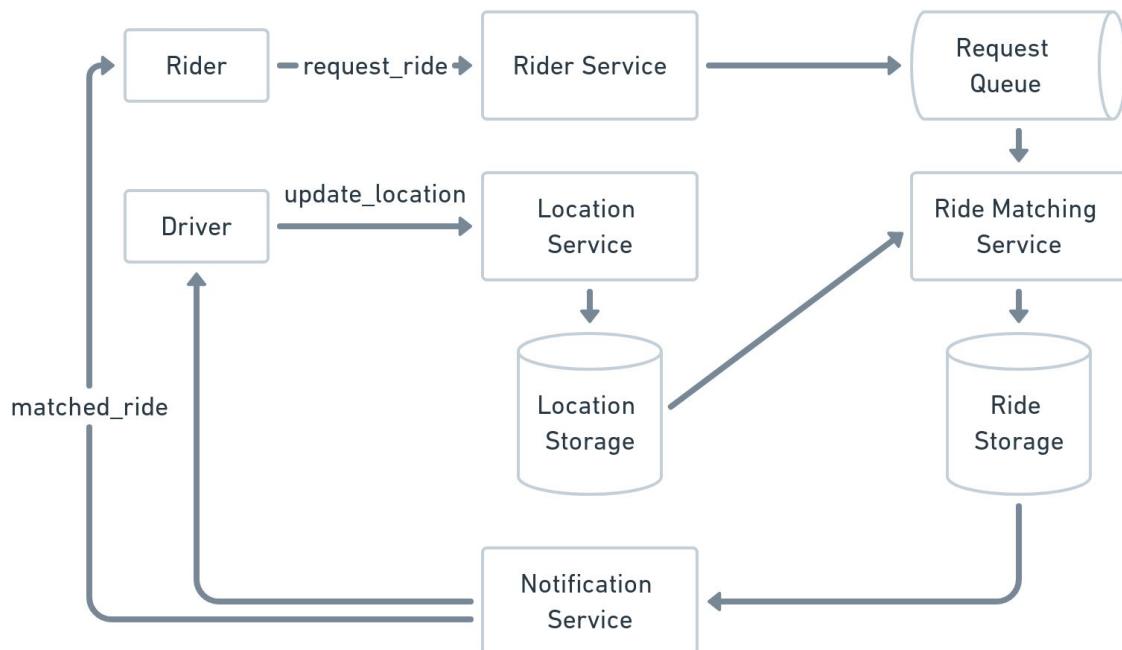
Step 5: End to End Flow

Purpose

After your requirement, API, high-level diagram, data structure, and schema design, the purpose of the pre-deep dive design is to take a deep breath and walk through how each API flows through your system and whether or not they satisfy the original requirements. It's also a moment to begin identifying any important talking points in the deep dive section.

We want to call this section out because almost all candidates immediately jump into scalability without making sure their system works and without identifying other important discussion points. Walking through the APIs and the flows also gives the interviewer some confidence that you and the interviewer have something that is working, which is a positive signal.

Situation
You've just finished the API, high-level diagram, data structure, and schema to satisfy the requirement to match riders with drivers.



For each API:

Step 1: Walk Through Each API and its Flow and Mark Any Interesting Points

“When the rider requests a ride, the API will pass its user_id and the location. The location will go into the queue with (rider_id, rider_location), the ride-matching service will take the rider request event. *Oh wait, maybe I can do micro-batches here. Let me come back to this.* Then it will fetch from the driver table a list of driver_ids and their corresponding score. *Hmmm, there might be some concurrency problems here but let me come back to this.* Once the system matches the rider with a driver, the matched event will get broadcasted out to them.”

“The driver will update their location with user_id and their current location. We may want to talk about the push frequency, but generally, if the system didn’t match the driver, we don’t need the frequency of a push because accuracy isn’t as important. We can dig deeper into QPS calculation if needed. After the location update, it will go into the driver location table, and the index will be updated.”

Step 2: Ask the Interviewer For Next Step

“So I believe the current design will work for most of the happy cases. I have a lot of discussion points I’ve captured. Is there any point you would like me to dig deeper into?”

Step 6: Deep Dive Design

Purpose

The purpose of the deep dive section is to demonstrate to the interviewer that you can identify areas that could be problematic—known as bottlenecks—and develop potential solutions and trade-offs to address those issues. In addition, the topics you bring up and how you frame your proposed solution will be telling about your maturity as an engineer.

How to Approach Deep Dive Design

The System Design Interview Golden Question

“What is the problem I am trying to solve?”

If there's one piece of advice we wish people would take, it would be to consistently ask yourself the question, “What is the problem I am trying to solve?” Unfortunately, most candidates are too solution-oriented, and they jump into solutions instead of thinking about the problem they're trying to solve. Being solution-oriented is most likely due to people reading blogs and tech talks from well-known companies and feeling obligated to retell the same level of difficulty of design in an interview to impress the interviewer. But instead, most seasoned interviewers are sitting there wondering why you're proposing a particular solution without any basis.

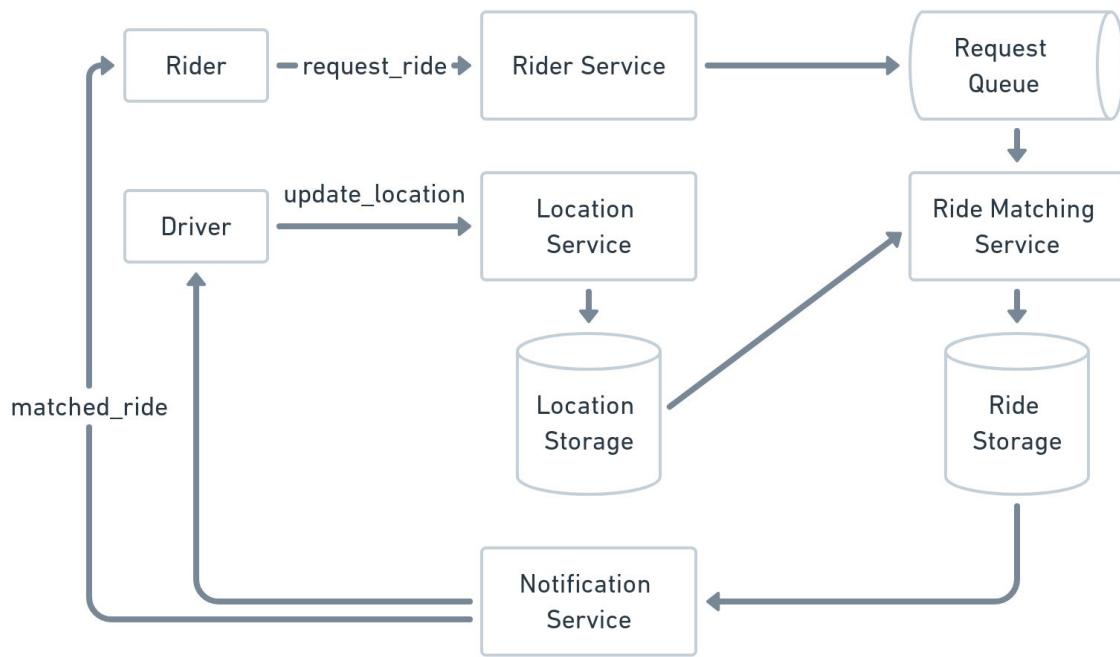
This advice applies to real-life work as well. You're not going to propose adding some cache cluster in front of your source of truth database to the senior engineer because you read it from a blog. It needs to make sense, and the trade-offs need to be well articulated, so the team knows the consequences of implementing the design. This approach should apply to system design interviews as well.

It's also worth pointing out that digging deep and realizing that a problem doesn't need to be solved is also important. Claiming it's a non-problem and concluding you don't need to solve it is a great data point for the interviewer. Why solve something that isn't a problem?

Candidates are nervous that they must exhibit distributed system expertise and try to solve something that isn't a problem, usually coming across poorly as a result. Imagine a software engineer joining a ten-person startup claiming the team needs to start sharding the database when there are less than 1 QPS in the system. Would you want to hire that person?

Situation

You've just finished the API, high-level diagram, data structure, and schema to satisfy the requirement to match riders with drivers and have finished walking through the APIs with the interviewer.



Don't Do This

"I would add a cache cluster in front of the driver location table, and I would shard the queue and the driver locations table because we have hundreds of millions of users, and we need to scale."

Do This

"Ok, let me think about areas that could potentially need to scale. Let's start with the drivers' location table. Let me calculate the QPS to see if it needs to scale."

The candidate does some back-of-the-envelope math calculations and comes up with 500k QPS to update_location API call.

“Now, we have a problem with 500k QPS, and each database from my experience can handle 20k-30k QPS at best. We have a bottleneck problem.”

Proceed to talk about the solutions, trade-offs, and recommend a final solution.

Analysis

In the Don’t section, it’s not even clear what the candidate is trying to solve. What if QPS isn’t an issue? Why are we assuming it to be an issue without a justification? In the Do section, the candidate identified a specific problem and proposed a solution. Solutions like telling the client to lower the update frequency can be an excellent scalability solution, even though it’s not “a hard distributed systems concept.”

How to Come Up with Discussion Points

Hopefully, you would’ve already made some discussion point lists by this time from the API, high-level diagram, and schema section. For each discussion point, remember to think about the golden question, “What is the problem I am trying to solve?”

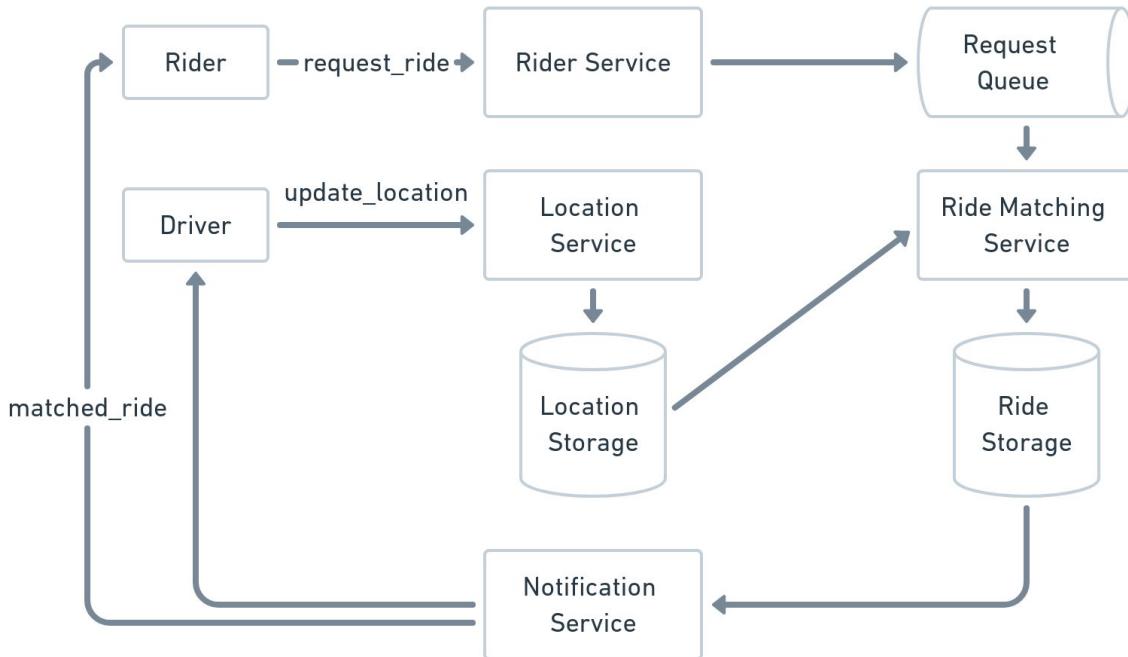
The good news is that you are the one coming up with the bottlenecks for the most part! In a system design interview, unless the interviewer tells you so, there’s no monitoring dashboard, product manager, or real customers to tell you what the query patterns are. So you can take advantage of this situation by coming up with problems and bottlenecks where you will shine. Just remember, it still needs to be relevant to the core of the problem.

Let’s say you are an expert in dealing with thundering herd problems and you’re working on a ridesharing service; you can set yourself for success by coming up with real-life scenarios where a thundering herd happens for

ridesharing service. Then you can easily proceed to propose solutions for the thundering problem.

Advice
Come up with problem scenarios and bottlenecks that will set you up for success. The magic formula for success is:
<p>Step 1: Identify a bottleneck Step 2: Come up with options Step 3: Talk about the trade offs Step 4: Pick one solution Step 5: Active discussion with interviewer Step 6: Go to step 1</p> <p>The more critical bottleneck you identify, the better it will look. Of course, your justifications from step 2 to step 4 will matter too, but if you identify irrelevant or non-critical bottlenecks, it won't be as impressive.</p>

Situation
You've just finished the API, high-level diagram, data structure, and schema to satisfy the requirement to match riders with drivers, and have finished walking through the APIs with the interviewer. You're looking for discussion points.



Do This

Take a look at the diagram you have so far. As an example, you suspect you might need a queue to take in the rider requests. You can set yourself up for success by claiming:

“Well, I am worried about thundering herd situations where there will be bursts of requests during rush hours and one-off events such as celebrity concerts where everybody leaves the venue at the same time. So I want to ensure the service won’t go down during such events.”

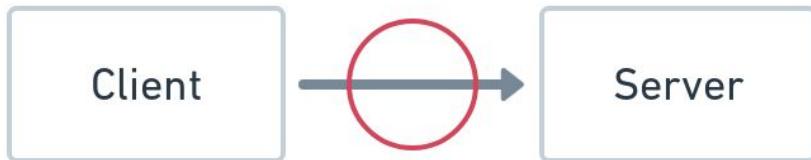
“Let me calculate what kind of QPS we need to support the driver location store. Perhaps we can consider turning it into a cache since we discussed that durability for location isn’t that important. Let me go over the options. Does that sound good?”

Sometimes candidates still have difficulty coming up with interesting discussion points because they’re nervous or just don’t have the relevant industry experience to have that kind of intuition. So below are some frameworks that may help trigger some ideas. In the following list, we

won't talk about the solutions to the problem yet. Instead, the focus is on identifying important problems and discussion points.

API and Interprocess Calls

When you have a high-level diagram, there will be boxes and arrows. For the arrows, it's usually an API or interprocess call. For each interprocess call, there could potentially be interesting discussion points.



Latency

With your high-level diagram, look at your end-user API and think to yourself if there's a query pattern where a discussion about latency is interesting. For example, in system design questions like typeahead, latency is critical. If you want to achieve 10 ms for p95, the network round trip in addition to a disk seek is likely not going to cut it. On the other hand, some questions like placing an airline ticket where the latency isn't as big of a deal. Not everything needs to have low latency. You should build a product intuition around that and confirm with the interviewer.

High QPS

Any components like servers, queue, cache, database, etc., can break if the QPS is too high. Discussing QPS is the right time to do back-of-the-envelope calculations to identify if QPS is an actual problem. There are some ballpark numbers in Appendix 2: Capacity Number to help define that. After you identify QPS as a problem, you can think of possible solutions to resolve the issue, and there are many creative ways to tackle it.

Bursty of the API / Thundering Herd

If the world runs predictably and uniformly, it should be much easier to predict capacity and design. However, there are always user stories in real-life patterns that may lead to a sudden influx of requests and break the system down. For example, when a celebrity starts a live Facebook feed, people rush to watch the stream. When people finish a concert, people are

requesting rides at the same time at the same place. When breaking news happens, tweeters are tweeting about the breaking news event. So, for the system design question you're given, try to come up with a user story as an avenue to talk about scalability when such a thundering herd event happens.

Slow, Low Bandwidth, Congested Network

When you look at the arrow between the end-user and the system, think about the network. Companies like Facebook are trying to target the whole earth to every user. The end-users may have poor bandwidth due to various reasons in areas with poor services. For example, poor bandwidth can make photo uploading for Instagram challenging due to insufficient bandwidth. Think about some technical solutions and product requirement compromises that you can make to improve the end-user experience.

Query Optimization

Until you finish API, high-level diagram, and schema, you probably only came up with a happy case API that satisfies the requirement, and for most users, it will most likely work. However, we would like to bring a good user experience to all the users. So, take a look at your end-user API and internal RPC calls for the input, the output, and the API calling patterns.

For input, can you reduce the size the client passes to the server? For example, if it's a huge file, can you just pass a chunk instead? For the output, are you able to reduce the amount of data passed back? For News Feed, instead of fetching for all the feeds, are you able to paginate it? For the end-user queries, are you able to reduce the number of API calls? Even if your design is already sound, letting the interviewer know that you're thinking about query optimization will give a positive signal.

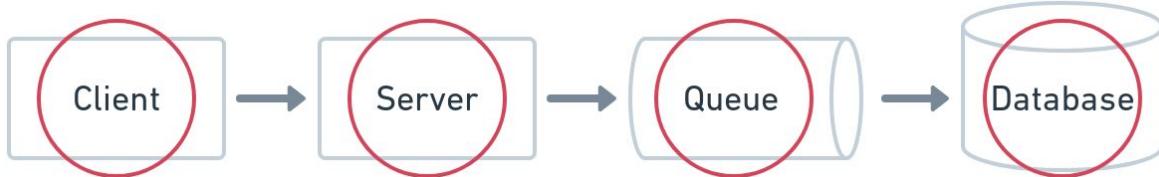
Further Technical Detail

Take a look at the arrow and think to yourself if it's worth digging deeper into that abstract arrow. For example, an arrow could be a TCP or UDP connection, short polling vs. WebSocket vs. server sent event, etc. If it's relevant to the question, it's an opportunity to dig deeper, and you may find opportunities for discussion points with the interviewer. For example, to

stream YouTube videos, you can explore networking to optimize the user's experience by talking about TCP versus UDP.

Micro Services, Queue, and Databases

On top of your high-level diagram, your arrows are boxes and diagrams that represent microservices, queue, load balancer, databases, etc. Take a look at each of those components and ask yourself some of the following questions.



Failure Scenario

When considering a component to discuss, think about what happens if that component fails and how the failure impacts your non-functional requirements. Think about partial failures and complete failures and the implications and consequences. Finally, think of creative and technical solutions to solve the failure scenarios.

For example, to design a ridesharing service, you need to store the driver's location, but what if the driver location storage is down? What would be the impact on customers? Are they still able to retrieve for a ride, or will they be matched with a less optimal ride?

Another example is what happens if a queue worker processes and commits a finished job but fails to acknowledge the queue. Another worker may end up picking that job again, leading to a double run. It's essential to think about the implication of double run here and the impact on the end-user.

High Amount of Data

For storage-related components like in-memory stores on an app server, cache, and databases, think about the current query pattern to the store and what the store is storing. Think about whether the current query pattern will lead to too much data and inefficiency for future queries. Will the current pattern lead to out-of-memory issues to the app servers and the cache?

Storage bottleneck is another opportunity for back-of-the-envelope calculations to see if the current storage pattern is sustainable and what kind of optimization solutions you can come up with.

Dig Deeper to Identify More Areas of Interest

Take a look at the high-level diagram and think about an area that you can deep dive into and discuss ever further. As you dig into an area, it may lead to other opportunities for problem discussions relevant to the question.

For example, if you were to design a notification service, you will likely have a bi-directional WebSocket connection to push from the server to the client. Most candidates will draw a bi-directional arrow between the client and the server and conclude there. WebSocket is an opportunity to dig deeper into how managing a WebSocket works and how some users may not get notified if a WebSocket server goes down. You should discuss with the interviewer if that's an acceptable user experience.

Design Choices

As you look at the high-level diagram, you might run into situations where you have to provide more details on top of the box in the flow diagram. As you explain, you might find yourself having to choose between design choices. For example, if you're looking at the database, you can think of the options and which one is the best for this question. If you're looking at a queue, you can talk about what kind of queue you would like to use for this question. When you dig into possible solutions, focus on the fundamentals of the technologies, not just buzzwords. Focus on how the fundamentals impact the user experience. This requires some technical knowledge which we will cover in the toolbox chapter.

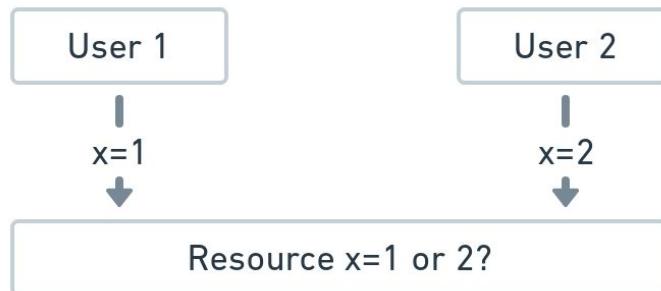
Detailed Algorithm, Data Structure, and Schema

Sometimes as you're going through the high-level diagram, you may have only one algorithm, one data structure, and one schema. Here's an opportunity to optimize the algorithm, data structure, and schema and develop more design choices. The question you are asking yourself is, "Do I have alternative solutions that would make the current system even better?"

For example, to design a ridesharing service and store the driver location, you might have a table that keeps on appending a new driver location where you insert one record every time there's a location update. Take a look at that and think if there is optimization you can make to that table. Are you able to store less? Are you able to optimize the lookup query?

Concurrency

Concurrency is a topic that most candidates fear, and mastery in this area will set you apart from most of the candidates. Usually, when you draw the high-level diagram, there will be shared resources, but what if the requests are accessing the same resources simultaneously? What would be the implication to the end-user?



You will get some credit from identifying the concurrency issue of your design, and it's even better if you can come up with a reasonable solution to the problem. So take a look at your high-level diagram and see if there are resources that are accessed at the same time by different clients. And even if the resource belongs to the same user, there can still be concurrency problems when there are multiple sessions.

Operational Issues and Metrics

Systems and users don't always behave the way you expect, and unexpected behaviors may lead to system failures, resulting in poor user experience. Sometimes, you'll want to monitor your system to ensure they are still working the way you expect.

For example, if you're designing a ridesharing service and introducing a queue to handle ride requests, what if the system isn't fast enough to take the requests? And even worse, what if there aren't enough drivers to pick up

the rides? You'll most likely want to keep track of the spillover metrics of the queue and get notified when it becomes a problem, since failing to match riders to drivers is a poor experience for the driver and the rider.

Another example is designing an hourly batch processing of building indexes for new articles for news search engines. Finally, you might want some metrics regarding whether the job has run and if they are successful. Failure to run will lead to outdated news and lead to a poor user experience.

The point of this section isn't to retell all metrics the big companies have in place. The point is to demonstrate to the interviewer that you can build a reliable system by identifying important areas to track.

Security Considerations

Candidates often overlook security discussions since most engineers think of them as something only for security engineers. In a system design interview, while the interviewer may ask you about basic concepts such as Transport Layer Security (TLS) and tokens as trivia questions, they shouldn't focus on the security discussion unless they're significant to the requirements.

For example, you should discuss the implications of a man-in-the-middle or malicious user attack for your design. For example, you designed an API for the ticket booking system with `book_ticket(user_id, ticket_id)`. What if the end-user tries to book many tickets and cancels them later on, to lock up the tickets maliciously? What would be the implication, and more importantly, how would you deal with it?

For a payment system like Venmo, if you came up with the API `transfer_money(from_user_id, to_user_id, amount)`, what if the user specifies an amount they don't have? What if the user specifies `to_user_id` as themselves and `from_user_id` as another person?

The answer may just be having some sort of token verification and business validation, but the point is to let the interviewer know that you have the awareness to bring up related security topics, and you get credit for doing

so. You can expect feedback like, “The candidate brings up relevant security topics and provides reasonable solutions to them. Most candidates don’t even think about security. I can trust the candidate to design safe and reliable APIs.”

How to Frame a Solution Discussion

When you’ve identified a problem to solve, you need to come up with multiple solutions, and for each solution, you need to discuss the trade-off. Then, after you have fully discussed the options and trade-offs, you should try to decide which solution you would prefer and why. After that, there may or may not be an active discussion with the interviewer, depending on the interviewer’s style.

Step 1: Identify and Articulate the Problem

You should clearly articulate the problem you’re trying to solve and why that problem is important and interesting for the system design interview question.

Step 2: Come up With Potential Solutions

Once you’ve identified the problem, you should spend enough time coming up with reasonable potential solutions. Since the system design interview is relatively short, usually, two solutions are good enough. Just make sure they are important. Unfortunately, most candidates just come up with one solution that doesn’t tell the interviewer about problem-solving skills.

Step 3: Discuss the Trade-Off

For each of the solutions, try to come up with pros and cons about the solution. For each trade-off point, think about whether or not it is that good or bad in the interview context. Try to tie this discussion back to the functional and non-functional requirements you defined with the interviewer.

For example, if your concern is about the durability of a solution because you’re using an in-memory store, you can reference how the non-functional requirement specifies the accuracy of the data isn’t essential. Then you

conclude that as long as you asynchronously replicate the in-memory store, replication-lag leading to out-of-date data is still acceptable.

Step 4: Take a Stance on Your Preference

After you've discussed the trade-off of the solutions, you want to take a stance on what you think the solution should be, based on the information you have and the assumptions you've made. Don't worry about right or wrong. There's no perfect answer here as long as the technical basis for your supporting argument is sound.

Step 5: Active Discussion With the Interviewer

The interviewer may disagree with your conclusion deep down inside them, but that's ok. An interviewer should judge based on how you justify your final recommendation and not on their personal preference. However, your fundamentals need to be logically correct and make sense. If they disagree with you because you are technically wrong, then, of course, that won't look good on you.

Usually, disagreement happens when two parties have different interpretations of the assumptions, and your job as a candidate is to try to resolve that. Unfortunately, this requires strong communication skills, but having the right mindset will drive the discussion forward.

For example, let's say you both agree that driver location doesn't need to be accurate. You propose you can update every 60 seconds, whereas the interviewer thinks that is too infrequent and it should be at least every 20 seconds. Now, both are not wrong because the definition of *accurate* is loosely defined here. You have to neutralize the discussion and address the root of the problem. In this case, the problem boils down to: how frequently should the location be updated such that we still provide a good user experience? In this case, you can neutralize by saying, "To find the right frequency for the update, we can try to perform testing on the frequency and monitor customer feedback." It shows maturity, and you've just proposed another solution to another problem!

Warning

Even though the interviewers are supposed to judge based on your justifications, unfortunately, sometimes inexperienced interviewers may be headstrong with a specific solution they are hoping to see. You will have to read the room and get a pulse check to make sure you don't leave it unaddressed.

For example, if you sense your solution doesn't convince the interviewer, you can communicate the following, "Here are the options and the trade-offs. I chose this solution because of this reason. Do you have any questions or concerns?"

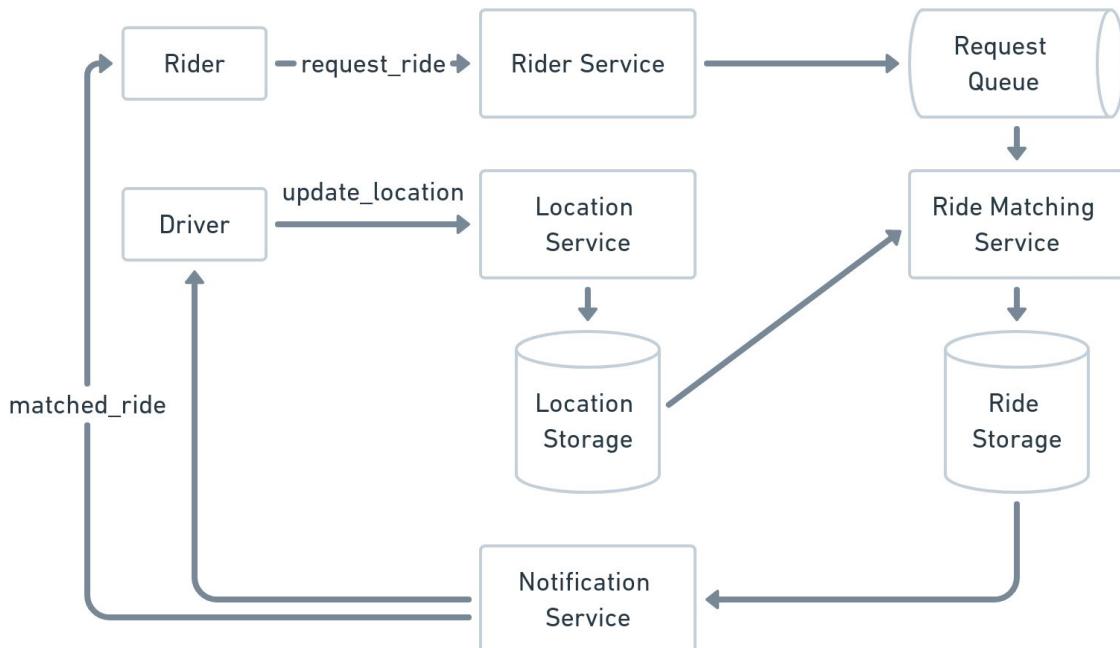
Remember, an interview is a two-way street, and sometimes you may be more experienced than the interviewer. You don't want to make the interviewer look bad. Be neutral and respectful while focusing on the problem at hand.

Step 6: Return to Step 1

After you've finished the discussion, identify the next important discussion topic and impress the interviewer.

Situation

You've just finished the API, high-level diagram, data structure, and schema to satisfy the requirement to match riders with drivers, and have finished walking through the APIs with the interviewer. You've identified the QPS to the driver table to be a problem.



Example Flow of a Solution Discussion

Step 1: Identify and Articulate the Problem

“Now, we have a problem with 500k QPS, and each database from my experience can handle 20k - 30k QPS at best. So we have a bottleneck problem. If we don’t scale up, the database might be unavailable for an update. Even worse, the database can crash and the users won’t be able to get matched with rides, and availability is important for our design.”

Step 2: Come up With Potential Solutions

“I can think of a couple of solutions:

1. Ask the driver client to update less frequently.
2. Instead of going to disk, we can store it in memory.
3. We can shard the database.

Let me think about the trade-offs for each.”

Step 3: Discuss the Trade-Off

“Here are the trade-offs:

1. By asking the client drivers to lessen their frequency, we will achieve lower QPS. However, the location will be less accurate. The pro is that it is simple. We probably just need to change a configuration file, and changing the configuration file can be quickly done. Instead of updating every second, intuitively, I feel like 10s would be just as good since how far can a car travel in 10 seconds?
2. We can store the location in-memory first instead of on the disk, but the problem is if the in-memory store goes down, we won't have their latest information. The advantage is that an in-memory store would allow for higher QPS. Also, since cars are moving around, as long as we pick up another leader to take rides, we don't need to worry about replication lag since we will get a ride update every 10 seconds anyway.
3. By sharding the database, the advantage is there will be more databases to handle the throughput. The con is additional complexity. Depending on how we shard, we may need to conduct scatter-gather, and there'll be operational overhead to maintain the shard coordinator.

So here are my thoughts, and let me think about which one I would choose.”

Step 4: Make a Stance on Your Preference

“Option 1 almost seems like a no-brainer; we should do that one for sure since there's a lot to gain with very little con. For option 2, since we're building this from scratch, we don't need to worry about migration from the database to the in-memory store. In-memory storage sounds like a good solution as well. For option 3, we might need to shard in the future, but for now, I believe option 1 and option 2 will solve our issue.”

Step 5: Active Discussion With the Interviewer

“So here's my preference. Do you have any further questions?”

[Proceed to discuss if the interviewer has questions and follow ups.]

Step 6: Go to Step 1

“If not, we can talk about the next discussion point I have on my list.”

Priority of Discussion

Now that we've listed many ideas for discussion points, you might be wondering which of the discussion points you should be talking about because, after all, the interview is only so long. You should try to read the room on what the interviewer is interested in. If the interviewer wants to know what you think is important, try to identify scenarios that will likely happen and are hard to solve.

Throughout the book, we've been focusing on solving a problem instead of coming up with solutions. The way you should think about it is that the impact of the problem is left untouched. Will the users leave your application, or will your company go out of business because you're spending way too much money on the infrastructure? Think about the severity of the impact. If it's not a problem, don't invest too much time into it. Like other pieces of advice, focus on points that are unique to the situation, not generic technologies that are irrelevant to the question.

Here are some examples:

Important

During ridesharing design, one discussion point you might consider is concurrency, where multiple riders may get assigned to the same driver. What would be the implication of that? The implication is that it will be a poor user experience when they fight over the driver at the pick up spot.

Important

If person A transfers money to person B, what if part of the transaction fails? The result would be inconsistent ledgers and lead to angry customers.

Important

If you're designing for YouTube and Netflix and you're serving some of the most popular videos from a data center that's somewhat far away, you might bring the internet down by occupying too much bandwidth through the routers.

Not As Important

You're designing for a notification system with very low payload , and you're thinking about introducing compression or bringing the servers closer to the user. However, it's not clear if solving for bandwidth will uplevel the user experience with the additional complexity, so you should have the intuition to not go in this direction.

Not As Important

You're designing an airline ticket checkout system, and you want to reduce the response time to sub 50ms. Even though this would be an excellent number target to achieve, customers are already accustomed to a couple of second response-time in practice, so it's not a problem.

Not As Important

You're designing photo storage, and you started focusing on the load balancer algorithm. Unless the load balancer algorithm impacts the end-user experience in a way that is unique to the question, don't invest too much time in it.

Chapter 3: Interviewer's Perspective

Now that we've gone through the system design interview framework, you might be left wondering what the interviewers are looking for. Even if you follow the framework, does that mean you will get the offer of your dream? There are many companies and interviewers with different expectations. However, if you communicate well, are knowledgeable, and are strong at problem-solving, there isn't any reason why the interviewers won't judge you as a strong candidate. This chapter will provide some guidance on what a rubric looks like and how it determines the final performance and leveling.

Rubric Leveling Explained

Since there are many companies out there, each with their own titles, to ease the explanation we will use the following terminologies for expectation setting:

Level 1: 0-2 years of experience

Level 2: 2-5 years of experience

Level 3: 5-10 years of experience

Level 4: 10+ years of experience

We left out Level 1 because entry-level engineers typically don't get system design interviews, so that would be irrelevant. As far as rubric goes, we will put the example performance in a spectrum starting from "No Hire" up to "Level 4."



The chart makes sense because when you're interviewing for Level 3, the interviewer might down level you to Level 2 if your system design

interview performance is at Level 2 and reject you if it's a No Hire. Likewise, if you're interviewing for Level 4 and your performance is at Level 2 or below, the company will likely reject your candidacy.

Generally, in an interview, you should strive to do the best you can if you have time to prepare. From our experience, junior candidates are performing better than senior or more experienced candidates because they are well prepared. To minimize the risk of rejection, you should try to do your best.

Rubric Examples for Each Section

The rubric will give some example bullet points which might lead the interviewer to concluding the candidate's level. The rubric should guide how much and how well you should achieve in a particular level. Again, this varies heavily between companies and interviewers and is intended only to give some perspective and to help understand the other side, since most people have never been a system design interviewer. Understanding how the interviewers think about the process helps you, as a candidate, provide the right data point for the interviewer.

Functional Requirement Gathering

No Hire
<ul style="list-style-type: none">- The candidate didn't ask any clarifying questions after the question.- The candidate had a difficult time understanding the problem statement and still could not continue after multiple hints.- The candidate kept discussing different features and failed to elaborate on the technical details.
Level 2
<ul style="list-style-type: none">- The candidate could come up with some features but was a bit murky about the user story. For example, "For Google Photo design, the candidate asked if we needed to display the photos but didn't come up with the wireframe on how to display the photos."- The candidate considered some cases and rushed into the next section. For example, "For Google Photo Design, the candidate asked good questions about the photo upload feature but didn't talk about how the user should view the photos."- The candidate was able to come up with important features with a little bit of guidance.
Level 3
<ul style="list-style-type: none">- The candidate interacted well with the interviewer and was very clear on the steps of the user experiences.

- The candidate considered the important features and narrowed down to a few requirements to focus on for the interview.
- The candidate did not need any hints or guidance from the interviewer.

Level 4

- Same as Level 3.

Non Functional Requirement Gathering

No Hire

- The candidate didn't ask any clarifying questions about the design constraint and the performance requirements and did not understand the scalability design.
- The candidate said some words about consistency, availability, and reliability, but did not understand how to relate the qualities to the end-user experience.

Level 2

- The candidate asked some important questions about non-functional requirements but also missed some. However, as the candidate proceeded throughout the interview, they were able to identify them.

Level 3

- The candidate listed most of the important design constraints and qualities for the design question and occasionally used them throughout the design decisions to make the final recommendation.

Level 4

- The candidate raised all the critical design constraints and qualities for the question and used those qualities throughout the interview to make design decisions and trade-offs.

API Design

No Hire

- The candidate missed some of the critical APIs for the system.
- The API design the candidate came up with wouldn't work after multiple hints.
- The API was highly inefficient, and the candidate failed to see the issues after hints.

Level 2

- The candidate could come up with a reasonable and efficient API design after some guidance and immediately understood the issues and course correct.
- The candidate was initially murky after some important components of the API and missed some important inputs or outputs, but the candidate could come up with the correct details after pointing them out.

Level 3

- The candidate was able to come up with an efficient API without much guidance and suggested some ways the API could be improved and be more extensible.
- The candidate needed little guidance on some of the trade-offs for the API.

Level 4

- The candidate could develop an efficient API with no guidance, proactively listed out all the major trade-offs, and proposed alternative solutions for major trade-offs if the requirements were to change.

High-Level Diagram

No Hire

- The candidate missed most of the major components of the architecture.
- The candidate had difficulty connecting the major components.
- The architecture the candidate came up with would not satisfy the requirements.

- Most of the architecture diagrams and data flows were unclear. Once the interviewer pointed out the issues, the candidate was still unable to provide more clarity.

Level 2

- The candidate identified most of the components needed for the system but realized some guidance to fill in some of the gaps.
- The candidate introduced components without discussing the whys but was able to course-correct quickly after some hints.
- The initially designed architecture had some issues, but the candidate could address the issue once the interviewer pointed out the problems.
- Some of the diagrams and data flow were slightly murky, but the candidate could improve upon them.

Level 3

- The candidate identified most of the major components necessary for the system and identified some of the major trade-offs to discuss in the deep dives section.
- The candidate understood most of the reasons why the components are necessary for the requirements and articulated the reasons well.

Level 4

- The candidate identified all the major components needed for the system and identified all the major trade-offs associated with the design.
- The candidate understood why some components are needed and not needed and were very cost-aware and able to remove elements that were not needed.

Data Structure and Schema

No Hire
<ul style="list-style-type: none">- The candidate had a very difficult time coming up with a schema and was often incorrect.- The selected data structures and schema were inefficient and would not work. The candidate didn't understand why, even with guidance.- The candidate failed to discuss the trade-offs of the proposed solution.
Level 2
<ul style="list-style-type: none">- The candidate was able to come up with a reasonable alternative but required guidance on another solution.- The candidate came up with a schema design but didn't self-identify some of the trade-offs associated with the design. However, with some hints, the candidate was able to address the issue.
Level 3
<ul style="list-style-type: none">- The candidate was able to develop reasonable schema designs and proposed some of the trade-offs associated with the design, but missed a couple of important points. However, with a small nudge, the candidate was able to address them.
Level 4
<ul style="list-style-type: none">- The candidate could come up with reasonable and efficient schema designs and the trade-offs associated with them. By identifying the issues, the candidate was able to propose solutions to address those issues.- The candidate was very detailed on connecting how the chosen data structures and schema relate to the user experiences and proposed potential trade-offs that the candidate needed to address.

Deep Dives

No Hire
<ul style="list-style-type: none">- The candidate was stuck after a high-level diagram, and the candidate

thought the design was complete and trivial.

- The candidate failed to understand how high traffic and volume of data would impact their initial design.
- The candidate had a significant knowledge gap in how some of the fundamental technicals work.
- The candidate could not go beyond the buzzwords and failed to understand the internals and whys of the proposed solutions.

Level 2

- The candidate could self-identify some of the bottlenecks but required some guidance on the solutions and trade-offs.
- The candidate could come up with solutions and trade-offs but didn't identify some of the critical risks. However, the candidate was able to understand the issue and propose a reasonable solution after some guidance.

Level 3

- The candidate was able to identify most, if not all, the bottlenecks and was able to provide most of the important pros and cons to the alternative solutions.
- The candidate was able to identify the bottlenecks, but some of the bottlenecks weren't significant or important enough to discuss.
- The candidate was able to provide most of the important technical details but required a little guidance on some details.

Level 4

- The candidate self-identified all the critical bottlenecks, proposed reasonable solutions with their pros and cons, and picked a final recommendation knowing the trade-offs.
- The candidate could provide enough technical details relevant to the requirements and communicated how that impacts the solutions and the trade-offs.
- The candidate could prioritize the right areas of discussion for most critical bottlenecks.

Chapter 4:

Communication Tactics and Common Mistakes

Improve Public Speaking

Believe it or not, a system design interview has a lot to do with how you present yourself. If you don't speak clearly and at a good pace, it can make the interviewer switch off, just like people falling asleep during a boring presentation. Therefore, it is worth investing in public speaking techniques like pacing, enunciation, modulation, body language, and confidence. Unfortunately, we can not emphasize the importance of public speaking enough since we, as engineers, emphasize the technical aspects and sometimes forget about all the soft skills that are just as important.

Keep the Requirements Simple

System design interviews are stressful, and many candidates attempt to get that offer by intense studying. For example, anticipating that the interviewer will ask the candidate to design a Netflix recommendation or Facebook Feed ranking, they probably studied various signals used in Netflix recommendations and Facebook EdgeRank. While the real-life signals are interesting, it unnecessarily complicates the interview by looking at many signals with different system complexities.

Don't Do This
"I will use Facebook edge ranking that takes into account affinity, weight and decay. For affinity we need to store the score between friends. For weight we need a configuration for different post types and we need a cronjob to periodically decay the score."
Do This

“I know Facebook uses EdgeRank to calculate each feed’s score. For now, we’ll assume the scores are computed and stored. If we have time, we can come back and see how we can update the scores as things change. Are you ok with that approach?”

Conclude and Make a Stance

In an interview it may be difficult to decide on the preferred option when coming up with options and trade-offs. Leaving the options in the air with an inconclusive attitude will make the interviewer think, *Ok, so what’s your preference?*

Senior engineers need to be able to deal with ambiguity and take a stance. Making a conclusion is your opportunity to demonstrate that attribute. Use the agreed-upon assumptions to strengthen your reasoning. In addition, you will need to build designs on top of previous designs. Leaving the design open will make the rest of the interview difficult to follow.

Don't Do This
“So there are option 1 and option 2 with the trade-offs so it really depends on the use cases.”
Do This
“So there are option 1 and option 2 with the trade-off. I’m going to assume the users only visit the site once a day so I will pick option 2 for this design, is the assumption fine with you?”

Articulate Your Thoughts

The system design interview is not a race to spit out the final perfect solution. It is a process of identifying important bottlenecks and problems, coming up with options, and justifying your design. Sharing your thought process is the majority of what the interviewer is looking for. Even if you are able to identify exactly what technology uses, you will still fail the interview if you don’t justify your thoughts because it is all contextual to the assumptions you make with the interviewer.

Don't Do This
“I will use a wide-column store.”
Do This
“I will use a wide-column store because the write to read ratio is extremely high with 100k write QPS where I expect a RDBMS to not perform as well. Also, based on the time series nature of the query, wide-column is a better fit because of disk locality.”

Control the Interview

During a system design interview, you will be doing most of the talking, and you want to be driving the interviewer in a favorable direction. For example, if there are multiple potential talking points during the deep dive and you are more familiar with databases than concurrency control, optimize your time and drive toward that direction where you would be discussing the database. But also be aware that sometimes the interviewer may still stop the flow and ask you to focus on a particular direction. You have to go along with their flow since they're looking for something particular.

There are many ways to pass an interview, but having structure will make your presentation much stronger.

Listen to the Interviewer

We can not emphasize enough that you need to be able to read the room and get a good sense of what the interviewer is expecting. Any system design question can go in an infinite number of directions, and sometimes the interviewer may just let you talk or interject with a direction they would like to go in.

Listen very carefully!

If you're unsure about what they're asking for, try to clarify it. And after you have answered their question, you should get a confirmation of whether you've addressed the interviewer's concern. Not doing well here will lead the interviewer to think that you don't listen well and not get the data points they're looking for.

Situation
Interviewer: “Tell me more about the queue.”
Don't Do This
“I'm using a Kafka queue because it is scalable. Now back to sharding, I would shard using user_id.”
Do This
“Would you like me to discuss what kind of queue I would use and why, or talk about the scalability aspect of the queue?”
“I would use a Kafka queue because the replay capability is useful for this streaming question in case the aggregator goes down. Should I go deeper into scalability or is there any direction you would like me to go in regarding the queue?”

Talk Over the Interviewer

Driving the interview is a leadership quality the interviewer looks for. However, you need to make sure you periodically check in with the interviewer to make sure you both are on the same page. That doesn't mean continuously talking so the interviewer doesn't have a chance to speak. And it also doesn't mean that when they do start speaking, you should interrupt them halfway through their statement. Not only is it rude, but you won't be understanding what the interviewer is looking for. In addition, if you talk over the interviewer, it doesn't matter what you're saying, they won't be listening.

Situation
Interviewer: “Tell me more about the queue.”
Don't Do This
Candidate: “I’m using a Kafka queue because it is scalable...”
Interviewer: “Can you tell me why scalability is an issue and how that...”
Candidate: [Interrupts] “Oh yeah because Kafka offers ways to shard the topics to scale the producers and the consumers and we need it for applications with a lot of users.”
Interviewer: “I mean, we are only assuming 50,000 users, so...”
Candidate: [Interrupts] “Oh yeah that’s true, so we can still use Kafka but we just need a single partition...”

Show Confidence with Your Justifications

During the interview, you should try to show confidence in your design instead of appearing unsure about your proposal. Here are some common phrases that show a lack of confidence and are indirectly asking for hints from the interviewer:

“*Do you [the interviewer] think this design would work?*”
 You shouldn’t ask the interviewer whether it’ll work since you shouldn’t be designing for something that doesn’t work.

“What do you think about the solution?”

You should be the one thinking about the solution. The interviewer is there to evaluate your thought process but not to share their own thoughts about the solution.

“Am I missing anything?”

Ideally, you shouldn't be missing anything, so this sounds like you're unsure if you're comprehensive and asking for a hint. So instead, say, “Here are the considerations I can think of. Should I continue to dig deeper and move on to the next topic, or is there anything else you would like to cover?”

“Am I heading in the right direction?”

There's no right or wrong direction. If you intend to gauge whether it's a direction the interviewer wants you to go, you can ask, “Is there a direction you would like me to head to?”

Instead of “asking for a hint,” you should demonstrate what you can think of first and conclude with:

“Here are the options and trade-offs I can think of and would pick option 1 because of the assumptions. Is there any direction you would like me to go into?”

Sometimes the interviewer might already be satisfied with your answer, and if there's any missing part, the interview might bring it up.

Of course, if you are really stuck and genuinely unsure, you can try to take a hint from the interviewer instead of faking it as if you know it. The point of showing confidence is that some candidates lack confidence throughout the interview even though they already have reasonable options, which usually leads to a low behavioral score.

Resolving Disagreement

When the interviewer appears like they may be disagreeing with you or is skeptical of your answer, they may or may not be, but you need to be very careful here. Here are a couple of possibilities:

Possibility 1: They Propose a Solution

Just like in real life, when other engineers propose solutions, you need to listen to the proposal and try to decide if it's better or worse with the assumptions. The interviewer may sometimes propose an alternative solution to see how you react. For example, they may say, "Instead of doing a push here, why can't you do a pull instead?" It may be in a direction you didn't think of, so you should digest it and come up with the pros and cons and rethink your conclusion. It is acceptable not to pick their solution based on your trade-off discussions. But if you're "wrong" about your solution and continue to be adamant about your initial choice, it will look bad.

Possibility 2: They Seem to Like Their Solution

If the interviewer is inclined toward their solution and you want the job, don't get defensive or argumentative. If they seem to disagree, it is likely the assumptions you two are basing your thoughts on are misaligned. Try to neutralize the disagreement by taking a neutral stance and acknowledging where they are coming from. Understanding the interviewer is extremely important because many candidates have been rejected for not managing the conflict well.

For example, imagine the interviewer thinks a driver location update every 30 seconds is too infrequent, leading to a terrible experience for a ridesharing application. If you disagree, you can neutralize by saying, "I understand your concern is the freshness of location. In practice, we can experiment with different update rates through A/B testing and adjust accordingly. For now, we can go with a more frequent update."

Don't make the interviewer look bad. Try to understand where they might be coming from. But, of course, don't always give in to whatever the interviewer says. You need to have some backbone as well. Just make sure there are strong justifications for your stance.

Discussion Point With Trade-Offs

For a system design interview question, there are usually a couple of core discussion points. For those discussion points, it is helpful to bring up a couple of options, discuss each option's pros and cons, and sell the interviewer on your final choice. If you just describe what you're going to do, it doesn't give the interviewer confidence that you've considered alternatives. It also makes you look like you've just recalled a solution you've memorized from a blog or other prep material. Coming up with alternatives with pros and cons and picking a final candidate also demonstrates your ability to deal with tough design dilemmas where there isn't a strictly better solution.

Don't Do This
“To design for a ridesharing service, we need to update the drivers' location. To update the driver location data, I am going to send the driver location every 5 seconds and we will update the location data into a quadtree.”
Do This
“To design for a ridesharing service, we need to update the drivers' location. There's a trade-off to be made based on the frequency. The advantage of higher frequency is more accurate data but the system will need to handle a higher QPS and vice versa. While accuracy is important, it isn't the end of the world if the assigned driver isn't globally the best so we have some room. Let's start with 20 seconds per update and readjust if we need to.”

Quality Over Quantity

Interviews are short. Avoid the trap of discussing many requirements in a hand-wavy manner and focus and deep dive into that core requirement. For example, Uber has thousands of engineers working full time. You are not going to cover everything about a ridesharing service. It is more important to demonstrate your ability to identify the most important topics to talk about than demonstrate breadth on a shallow level.

Don't Do This

“Here are the requirements I can think of. When the rider opens the app, the rider can see all the drivers around them. We can call the location service and we can use a quadtree for that. We can match the riders with drivers, the drivers can have different tiers of cars, we can pass and store the driver metadata. When a user requests for a ride, we can show the rider the ETA and we can call the Google Maps API for that.”

Do This

“Above are the requirements I can think of. I am only going to focus on the rider and driver matching part for now. If we have more time, in the end, we can go through more requirements. I will go over the API, high-level diagram, and deep dive into any interest areas for this core use case. Are you ok with that?”

Math With a Purpose

There are quite a few resources out there suggesting you do a back-of-the-envelope calculation to calculate metrics such as the number of databases needed, query per second, bandwidth, etc. However, the interviewers aren't looking to assess your basic algebra skills. Instead, use the results to justify your design choice.

Also, candidates often try to perform back-of-the-envelope calculations after gathering the initial requirements and before API, high-level diagram, and schema design. Early math calculation is quite dangerous because without defining all the APIs and schema, you might only be calculating the number for one of the APIs. In addition, your storage capacity calculation might be inaccurate since you haven't finalized the schema.

The purpose of the back-of-the-envelope calculation is to demonstrate your ability to make reasonable assumptions and derive a result to justify a design you're going to make. For example, if you decide you want to start sharding the database or introduce a cache layer, calculating the QPS and

identifying the system has a high QPS can be one of the factors to justify your desire to partition or introduce a cache. If the QPS is low, proposing database partitioning is introducing complexity with no benefit.

Don't Do This
“I’ve calculated the QPS is 100k and storage is 20 PB, let me talk about the API design now.”
Do This
“ I’ve calculated the QPS is 100k, looks like we will need to scale our app servers since each app server I’m assuming can only take 30k QPS.”

Focus on the Right Things

After you've clarified the requirements and are diving into a requirement, there are still many technical components you can discuss. There are a few core design points unique to a requirement. While you're not a mind reader into what the interviewer is thinking, we need to develop intuition to focus on what matters. For example, suppose the interviewer asks you to develop a private messaging system to enable one-on-one chat, while authentication is important. In that case, it is a waste of time to spend 10 minutes on how OAuth 2.0 works.

There are two caveats. The first caveat is when the interviewer specifically asks you to focus on an area. Then you need to adjust the course to accommodate their expectations. The second caveat is when the interview is domain-specific, like security, where the heart of the problem changes.

Don't Do This
“To design an e-commerce checkout API, we need an API to call the API Gateway. For this API, I will use TCP/IP instead of UDP because of reliability. I will call the authentication server first to fetch the security token. There will be a load balancer to handle my request and I will use a round robin algorithm instead of session based because it is simpler to

keep the load balancer stateless and it scales better. I will make sure I have TLS set up for Https.”

Do This

“To design an e-commerce checkout API, we need to ensure it is available and low latency because there are studies that show low latency and availability result in greater profit. I will focus on the signature of the API, how different services interact with each other, and the schema of the storage layer to ensure we meet the low latency and highly available requirements.”

Spewing Technical Details With an Intention

While knowing the ins and outs of how a piece of technology works is impressive, spewing details about how a specific technology works without an intention isn’t helpful in a system design interview. Technical knowledge is only impressive if you’re able to apply it to justify a chosen design. For example, when the interviewer asks you to design a chat system network protocol, instead of spewing the deep technical details of WebSocket, focus on a couple of network protocol options and describe how the technical details of the chosen solution help solve the requirement at hand.

Don’t Do This

“I would use WebSocket to establish the connection between the client and the API gateway. WebSocket is a channel over HTTP through TCP connection, allowing client and server to communicate bidirectionally. WebSocket has gained popularity in recent years and is popular for chat applications.”

Do This

“Our requirement is to have a seamless chatting experience where users should receive the sent messages almost instantaneously. For this we have two options, we can have the client periodically pull from the server or establish a WebSocket connection. Since WebSocket establishes a bidirectional connection between client and server, it allows the server to

push new messages immediately to the client. This is better than a periodic pull that would result in a slight delay.”

Don’t Jump Into a Real-World Design

Studying for a system design interview is challenging. Most people study by reading technical blogs, watching tech talks, and exploring other materials for design ideas. Please don’t start memorizing and retelling a design that you’ve read somewhere.

Instead, apply the fundamentals and articulate how the learned fundamentals are related to your current design. You might face unique requirements and nonfunctional assumptions in a real interview, so the same design might not apply. Also, it’s obvious when a candidate memorizes a solution because they are not flexible.

Don’t Do This

“I will use Cassandra to build the chat application and use Snowflake to generate the ID as a cluster key because that is a well-known design.”

Chapter 5:

Technical Toolbox

This chapter will go over most of the concepts that will likely show up in a system design interview. For each of the concepts, we will cover the definition and some of the design options. Most importantly, like mentioned previously, we will discuss when to apply each concept and the implication to the end-user experience.

Back-of-the-Envelope Math

What is the Purpose?

One of the purposes of back-of-the-envelope math is to justify a design. Let's say you did a back-of-the-envelope calculation that resulted in 10^8 QPS, and you assume a single machine can only handle 10^6 ; you should identify that as a problem and come up with proposals to deal with that problem. It will look great on you if you can dismiss the need to scale based on the calculated results.

Conversely, if the numbers result in 10 QPS and you start talking about caching and sharding, it will not look good on you. Spewing random system design concepts won't get you brownie points. You need to have a solid reason to introduce scalable solutions.

Do I Need to Do Math?

A lot of system design interviewers don't want you to waste time doing some basic algebra math. If you are unsure whether you should, here are a couple of things you can do:

1. Ask the interviewer for QPS and storage capacity directly. They might just say, "Assume a lot," which just means you need to scale. Or if they give you the QPS, it saves you a ton of time.

2. Ask the interviewer if you can just assume if you need to scale beyond a single machine. In most interview contexts, it doesn't matter if it's 5 or 50 machines. You need more than one machine. In practice, the magnitude of scaling does matter, so you need to proceed accordingly. The point is, you don't want to waste time on back-of-the-envelope math when the interviewer isn't interested.
3. If the interviewer prefers you to do the math, do it quickly, efficiently, and accurately, and don't spend too much time on it. Practice it well using the strategies described below.

Types of Back-of-the-Envelope Calculations

Calculate QPS Formula
<p>QPD (Query per day) = [Daily active users] x [% of active users making the query] x [Average number of queries made by each user per day] x [Scaling factor]</p> <p>Query per second (QPS) $\sim=$ QPD / 100k</p> <p>There are 84,600 (24 x 60 x 60) seconds per day, you can round that up to 100k for easier calculation.</p>

Tips
<p>When you divide by 100k, just remove 5 from the power of 10.</p> <p>For example: $10^{11} / 100k = 10^{11-5} = 10^6$</p>

You want to design for the worst case. In practice, there are different levels of measurements like p50, p90, p95, and p99. You want to consider the QPS that can break your system.

Daily Active Users

The daily active user count is something that you should ask the interviewer. Sometimes they may tell you the number of users instead of *active* users. In that case, you assume a percentage of total users to be active users.

% of Active Users Making the Query

Sometimes not all active users will perform a specific action. For example, if the interviewer asks you to calculate the QPS for News Feed posts for Facebook, you should assume only a fraction of the users will post since most are just News Feed readers. Usually, 1% - 20% writer to reader is a fair assumption, depending on the application. Either way, you should clarify the assumptions with the interviewer.

Average Number of Queries Made by Each User Per Day

You should make an assumption of how many requests an active user makes each day. Even though the interviewer doesn't expect you to know the real-life numbers, you should still apply some common sense here. For example, if the question is about the average number of News Feed posts per active user per day, 1 - 5 posts would be reasonable. Something way out of the range like 1,000 - 2,000 posts per day may raise some eyebrows. Of course, the interviewer may change the assumption since a system design interview doesn't always have to reflect a real system. In that case, you should go with the interviewer's flow.

Scaling Factor

Give a product scenario to show good product sense. For example, when requesting a ridesharing service, mention how weekend nights have more activity and that you'll assume it has 5-10 times the average traffic. Since the above number is an average, you should multiply by some factor to consider the worst-case scenario. The point isn't to show you can multiply a number with a number. The purpose is to demonstrate to the interviewer that you have thought about a potential bottleneck.

Storage Capacity Formula

Storage capacity for time horizon =
[Daily active users] x
[% of active users making the query to persist] x
[Average number of queries made by each user] x
[Data size per query] x
[Replication factor] x
[Time horizon]

Daily Active Users
Similar to QPS.

% of Active Users Making the Query to Persist
Similar to QPS.

Average Number of Queries Made by Each User
Similar to QPS.

Data Size per Query

Make sure you consider all the data you're storing. If you're saving an Instagram post, make sure you remember to calculate both the photo and the post metadata. Most candidates try to be very precise about each column row data size and sum them up, taking a lot of time. Instead, just try to assign each column to a rough memory, sum them all up, and round it to a nice number. It doesn't matter if a String column is 20 or 30 bytes. No one cares. If the sum is 68 bytes, round it to 100 or 50 bytes, no one cares if it's precisely 68 or not. Look at Appendix 2: Capacity Number to help with assumptions.

Replication Factor

Generally, companies replicate a database into other databases for backup. Typically that number is 3. The point is to show the interviewer you have thought about it.

Time Horizon

You need to know the time horizon to plan for capacity. Usually, somewhere between 1-5 years should be fine. In practice, capacity planning is very contextual.

Back-of-the-Envelope Calculation Techniques

Doing the back-of-the-envelope-math incorrectly during an interview will leave a bad impression and waste precious time. It is important to understand some of the techniques and practice before the real interview. We will use an example to demonstrate the technique steps:

Daily active users: 200 Million

% Of users who post photos: 20%

Average number of photos per user: 10

Memory per photo: 10 MB

To solve for the total memory can be quite intimidating and most candidates make mistakes.

Step 1: Convert All Numbers to Scientific Number

Multiplying 200 Million to 10 MB is very difficult and error-prone. You should convert the numbers into the scientific number ($A \times 10^b$), then sum all the power of 10s together and compute the appropriate result.

Tips

As a mental trick, every comma you see is a power of 3 shifted over by 10s.

For 10,000, you see one comma, so it'll be 10^3 shifted once to the left, so you add one to become 10^4 .

If you see 100,000,000, that's two commas, so it'll be 10^6 shifted twice, so it'll be 10^8 .

You should come up with the scientific notation in less than 2 seconds using this method without a mistake.

Example: Calculate the amount of photo memory stored per day.

Daily active users: 200 Million $\rightarrow 2 \times 10^8$

% Of users who post photos: 10%

Memory per photo: 10 MB $\rightarrow 1 \times 10^7$

Average number of photos per user: 1×10^1

Replication factor: 3

Step 2: Group all the 10s together

$$10^8 \times 10^7 \times 10^1 = 10^{16}$$

Step 3: Group all other numbers together

$$10\% \times 3 \times 2 \times 1 = 0.6$$

Step 4: Find out the final number

$$0.6 \times 10^{16} \rightarrow 6 \times 10^{15}$$

Step 5: Convert the final number into a readable number

Use Appendix 1: Conversion Table to read about converting the scientific number into something readable quickly and accurately. You don't always have to convert it back, but sometimes it helps drive the conversion as humans are used to thinking in Petabytes instead of 10^{15} . Unfortunately, you need to memorize the conversion table, but it shouldn't be too bad.

$$6 \times 10^{15} \rightarrow 6 \text{ Petabytes}$$

Step 6: Do something with that number

Use Appendix 2: Capacity Numbers to justify whether you need to scale or not. Assuming each host can only store around 1 TB, we need to scale up the servers.

Common Mistakes

Common Mistake 1: Spending A Long Time Doing the Calculation

Interview time is short and precious. You and the interviewer will likely spend 5 minutes on the introduction and another 5 minutes on the Q&A,

leaving you 35 minutes for the technical portion. If you spend more than 10 minutes on the back-of-the-envelope math, you will spend a third of the time demonstrating your algebra skill. This means there is much less time to demonstrate your system skill. Spend time practicing and do them quickly.

Common Mistake 2: Making Calculation Mistakes

Most candidates make some mistakes. For example, candidates have misrepresented the result by a magnitude of 1,000 because they confused GB with TB. They were missing a magnitude of 10 because they missed a 0. While an innocent one-off mistake might not be a game-ending one, it certainly doesn't look good to the interviewer. Practice using the methods below so you can do the math quickly and accurately.

Common Mistake 3: Not Doing Anything With the Results

You have just invested some time calculating 6 PB of photos stored per day; do something with that number instead of doing nothing with it. The interviewer isn't looking to see your ability to do simple algebra. They want to know how you use it to identify a bottleneck and propose solutions for the bottlenecks.

Common Mistake 4: Calculating Unimportant Numbers

Time is precious in an interview, and you need to develop some intuition on what is important to calculate to identify bottlenecks. If the design is about a heartbeat service with lightweight requests, and you spend 2-3 minutes calculating whether bandwidth is an issue, the interviewer will question your technical judgment.

Common Mistake 5: Doing the Math Too Soon

The current trend is to try to do the math shortly after the requirements gathering before API and schema design. Doing so is illogical and error-prone because QPS is associated with an API, and storage is associated with a schema. Most design questions will have more than one API and more than one table. For Instagram, you have post_feed and view_timeline APIs and metadata and photo storage. Without going through the API and schema, you might accidentally miss out on some.

API Design

We have talked quite a bit about API in the system design interview framework. Here we will go deeper into some of the things to watch out for. Ultimately, your goal is to ensure the API is clear, efficient, and solves the problem at hand. Here are some advanced concepts to talk about during the API design:

Idempotency

Imagine the interviewer asks you to design an API to decrease the quantity of an item in an e-commerce order. Let's say you have an order of quantity 3 and you want to make it 2. There are two ways you can do it:

```
decrease_quantity(order, quantity_to_decrease)  
update_quantity(order, quantity)
```

In `decrease_quantity`, you will call `decrease_quantity(order, 1)`, and the item quantity will be decremented by 1. This API isn't idempotent because if you call it multiple times, the quantity will keep decreasing.

In `update_quantity`, you will `update_quantity(order, 2)`, and the system will update the order quantity. This API is idempotent because no matter how many times you call it, the quantity will still be 2.

Idempotency is usually preferred since it's possible to accidentally retry and result in unwanted behavior due to system and user errors. However, idempotency isn't always possible. For example, if you place an order multiple times, it is difficult to tell if it was a mistake or if you intentionally want to buy twice. You can introduce features to warn the user placing the same order within a short time frame, however, that is not considered idempotency.

Client vs Server Generated Data

When you have an input parameter in your API, you might be tempted to have the client pass as many inputs as possible. For example, a common data needed is timestamps. You can have the server generate the timestamp

instead of the client. The client clock may not be in sync, or the client could be malicious, tampering with the timestamp field.

Efficiency

Make sure your API is efficient. For example, if you are fetching a list of news feeds, make sure you don't return every news feed for the user. You should think about pagination.

Correctness

Correctness might sound obvious, but please make sure the API signature is true to its task, and the input and output are sufficient to satisfy the question's requirements.

Focus on the Core

For a given API, focus on the input, output, and signature name. Don't focus on a protobuf and RESTful schema since that's not the core of the question and is pretty wasteful. Only focus on it if the interviewer directly asks for RESTful design.

Schema Design

During the system design interview, it's critical to have clarity of the schema. In an interview, it's not critical to choose a particular database right away since that requires digging deeper into the access and storage pattern. Come up with a logical schema first and ensure the integrity and the relationships are clear and how the APIs will fetch the information.

We will design a file storage system with folder structures to showcase a schema design example. We will also have the concept of user-created label files for users.

Defining Schemas

A table should have a primary key that uniquely identifies a row. That is usually a system-generated id, but it doesn't have to be. You can have an application-level unique ID as well. For an interview, a system-generated ID is probably good enough. Don't worry if the ID is UUID, integer, or String (not that efficient as an ID); it is usually not the core of the question unless it's a TinyURL related question. The primary key can also be a clustered index which means you can fetch a primary key efficiently. Other tables can use the primary key as a foreign key (FK). A table has columns to store row-level information.

Below, we will create one-to-many and many-to-many structures. A lot of candidates make mistakes here.

Step 1: Create the Tables Needed

User Table

user_id

Folder Table

folder_id	user_id (FK)	folder_name
------------------	---------------------	--------------------

File Table

--	--	--

file_id	user_id (FK)	file_name
----------------	---------------------	------------------

Label Table

label_id	user_id (FK)	label_name
-----------------	---------------------	-------------------

Step 2: Each File and Folder Belongs Into a Folder

Folder Table

folder_id	user_id (FK)	folder_name	parent_folder_id (FK)
------------------	---------------------	--------------------	------------------------------

File Table

file_id	user_id (FK)	file_name	folder_id (FK)
----------------	---------------------	------------------	-----------------------

The parent_folder_id is a foreign key to its table. This table is an example of one-to-many. One folder has many files. One folder has many subfolders.

Step 3: Each File Can Have Many Labels Applied to Them

File Label Table

id	file_id (FK)	label_id (FK)
-----------	---------------------	----------------------

For Label, we create a separate many-to-many table to show that a file can have many labels applied to them, and a label can be applied to many files.

With these tables, we can answer some of the following queries and many more.

“For a given folder, give me all the files and folders in that folder.”

“For a given file, tell me which folder I belong to.”

“For a given label, tell me all the files applied by that label.”

Defining Key Value Schemas

Another approach you can use to deal with some of the schemas is to apply a key-value schema. For example, for the File Label Table, you can also specify file_id → [label_id]. While you’re able to answer “give me all the

labels associated with the file” efficiently, it becomes tricky with “give me all the files that belong to a label_id.”

Our recommendation is to keep them simple with relational like abstract schema with no commitment to a database choice until the deep dive or the interviewer asks. Typically as long as the schema is reasonable and can satisfy the queries, it's not the focus of the interview.

Normalization vs Denormalization

Some people get stuck debating if they should normalize tables or keep them denormalized because of the inefficiency of the join. Normalization doesn't matter that much in an interview setting since it's a generic debate that's usually not unique to the interview question. However, it might be worth bringing up if you identify the read throughput to be an issue.

You can briefly touch on the normalization topic if you want during the deep dive, but in practice, companies do both, and it's hard to tell which one is better without real-world metrics. Our suggestion is to keep things normalized with the proper logical separation, so the entities are clear and well defined with the interviewer.

Indexing Strategies

When you're designing the schema, you need to think about the efficiency of your query with your proposed schema. Without proper indexing, it may result in a full table scan. In a system design interview, it's worth mentioning how you would index to achieve better performance.

Once you've designed the schema in an interview, one talking point would be to discuss an indexing option. Sometimes this may be straightforward, sometimes there may be options, but it will give you some credit if you bring it up and come up with a reasonable solution.

What is Indexing?

You can use indexes to locate a piece of data quickly instead of searching for every single row. There are various data structures for indexes, but we

will focus on primary and secondary indexes.

Abstractly, an index is just a sorted table that allows the search to be $O(\log N)$ using binary searching with a sorted data structure. Internally, an index can be implemented using B-Tree or LSM, which uses Sorted Strings Table (SST). Since an index table is a sorted table with $O(\log N)$ search, the performance will still degrade once the number of records (N) increases. However, index lookup with $O(\log N)$ is still significantly faster than a full table scan $O(N)$.

Reminder

Database indexes are not hashmaps. Hashmap is an in-memory data structure. You may implement an in-memory solution on top of the database, but the index stored on disk isn't a hashmap.

Primary Index

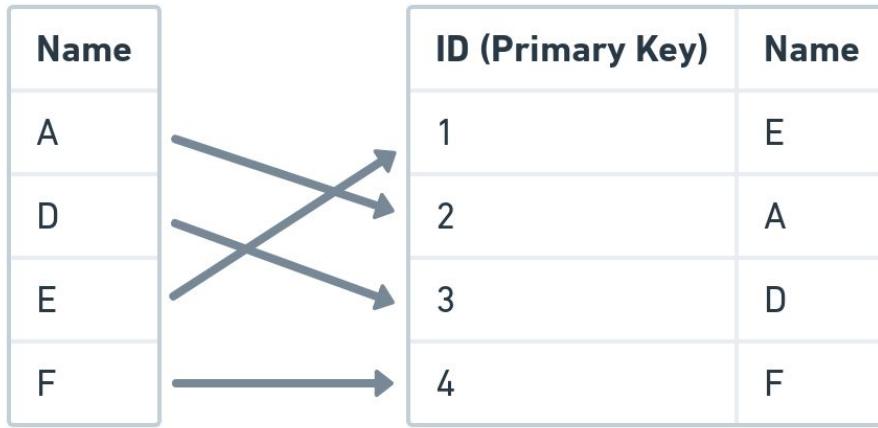
When the main table is created, by default, the primary key is also the clustered index which means the primary key on disk sorts the main table itself. Therefore, searching by the primary key will be efficient.

ID (Primary Key)	Name
1	A
2	D
3	E
4	F

The write to the main table will be efficient if the primary key is sequential but may need to be reshuffled if it's not and results in a less efficient write. Note that the clustered index doesn't have to be the primary key.

Secondary Index

Sometimes the clustered primary key may not be efficient enough for your query. You may want to consider adding in a secondary index which is another sorted table to reference the main table record. The advantage is a quick reference to the record of interest, and the disadvantage is the write will be slower with more tables to update. Having additional indexes is still preferable for higher read-to-write ratio applications.



Index Use Cases

Now that we've learned about indexes, we will talk about applying them in an interview setting.

Key or Attribute Lookup

The most straightforward lookup is by a key or attribute. For example, imagine you have a table with the following columns:

`car_id, color`

`Car_id` is the primary key with no secondary index. If you want to search for a `car_id`, it will be efficient since `car_id` is the primary key. However, if you want to search for any car with a specific color, it will have to scan row by row. To improve this, you can add an index on the column `color`.

Range Lookup

Because the index table is sorted, retrieving rows in a sorted fashion will be efficient if it is indexed. For example, imagine you have the following table where `post_id` is the primary key and `created_time` is not indexed.

`post_id, created_time`

If you want to search for the 20 most recent posts by `created_time`, you have to retrieve all the records and sort them. If you index on `created_time`, you can fetch for the latest 20 jobs more efficiently since the index is a sorted column.

getting post_id from time 2-3 is efficient because it's sorted

Time	Post ID
1	1
2	2
2	3
3	4
4	5

Prefix Search

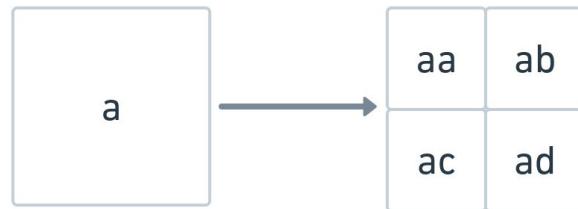
Since the index table is a sorted table, it's not surprising that you can search by prefix. For example, if you have a table of previously searched terms with a corresponding score, you can fetch all the terms with the prefix and sort it by score.

get elements that start with prefix "a" efficiently because it is sorted

Name
a
ab
ac
b
b

In a type-ahead system design interview, the latency to fetch to disk and sort by score is probably not good enough for the question's latency-sensitive nature, but it is an option worth discussing as a trade-off.

Another use case is geohash, as identified by 9 characters. The longer the geohash, the more precision. For example, imagine you have a square “a” and as you get more granular, you append another letter to it. With prefix search, you’re able to get all the “aa,” “ab,” “ac,” and “ad” if you search for the prefix “a.”



Composite Index

So far, we’ve talked about an index for a single column. This section will talk about indexing on multiple columns and requires a bit more thinking on top of just throwing an index to a column. The idea is the same for row key design for NoSQL design, where NoSQL concatenates the key instead of defining it as separate columns.

Imagine you have a table showing driver status:

driver_id	status	location
1	Waiting	New York
2	Busy	New York
3	Busy	New York
4	Busy	Chicago
5	Waiting	Chicago
6	Busy	New York

To demonstrate composite key, we will use two examples:

Option 1: Create an composite index (status, location)

driver_id	status	location
4	Busy	Chicago
2	Busy	New York
3	Busy	New York

6	Busy	New York
5	Waiting	Chicago
1	Waiting	New York

Efficient Queries:

Give me all the busy drivers

Give me all the busy drivers in Chicago

Inefficient Queries:

Give me all the drivers in New York since the location is only sorted in the status scope

Option 2: Create an composite index (location, status)

driver_id	status	location
4	Busy	Chicago
5	Waiting	Chicago
2	Busy	New York
3	Busy	New York
6	Busy	New York
1	Waiting	New York

Efficient Queries:

Give me all the drivers in Chicago

Give me all the busy drivers in New York

Inefficient Queries:

Give me all the busy drivers since status is only sorted in the location scope

As you can see, the way you index will impact your end design so it's important to know what your query is.

Geo Index

The world geodatabase is complicated with alternatives like R-Tree, kd-tree, 2dsphere, Google S2, etc. Since geospatial questions still come up quite often with ridesharing services and Yelp, it's worth talking about

approaching the geo index. In a system design interview, it's doubtful the interviewer is interested in the deep technicalities of geodatabases, so try to keep it simple and proceed.

If you wish to go to disk, here are two simplistic options that are good enough for an interview:

Option 1: Geohash solution like in the prefix search section.

Option 2: Reverse index of location_id to a list of objects like drivers or points of interest.

location_id → [object]

You can represent location_id as a tuple (longitude, latitude). Then, you can calculate the values using the floored longitude and latitude. That way, you don't need separate storage to store longitude and latitude ranges to a grid ID. For example, (1.11, 2.25) can be floored to the (1, 2) grid and use (1,2) as an ID to a location block.

(3, 1)	(3, 2)	(3, 3)	(3, 4)
(2, 1)	(2, 2)	(2, 3)	(2, 4)
(1, 1)	(1, 2)	(1, 3)	(1, 4)

Databases

Definition

A database is an organized collection of data that is stored by the application for durability.

Purpose

Most system design interviews will require permanent storage to persist data and allow other services to read the data. Therefore, choosing the right database is important to the performance of your architecture. In a system design interview, the interviewer will sometimes be looking for your ability to justify the database of your choice based on fundamentals. Some interviewers will dig deeper than others, so the more you are prepared, the more likely you will get a solid score.

How Do I Choose a Database?

Choosing a database is difficult. In practice, engineers could spend weeks making that decision because it is difficult to migrate once the database is in place. In addition, the database is a very complex space with a lot of competing products and complex internals. Therefore, you need to strike the right balance between going too shallow and too deep during an interview. For example, here are some common database discussions that lack depth:

“I will use NoSQL over MySQL because it’s more scalable.”

“I will use a database to store photos.”

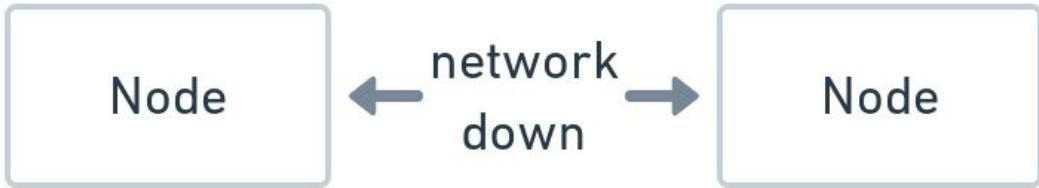
“I will use NoSQL because it scales well.”

“I will use Cassandra because it scales well at our company.”

While the above answers are not wrong, it just doesn’t show deep technical database knowledge. Someone without any software engineering experience can give the same level of answer.

In a system design interview, we encourage you to reflect on your design patterns and choose a database that mostly meets the needs of the design. Then, you should give a list of reasons that make that database a stronger choice than the other ones. Justifications like, “I will use Cassandra because it scales well,” are too hand-wavy because other databases also scale well. All databases will scale, otherwise they wouldn’t be competing in the database space. And yes, MySQL scales too.

Reminder



CAP theorem is a generalized term to help understand distributed databases.

C stands for consistency, which means all the databases see the same data at the same time.

A stands for availability, which means the database is up to serve traffic.

P stands for partition tolerance, which means the system continues to work if there's a network issue between databases.

CAP theorem states you can only have two of the three.

AC: Means there's no partition with a single machine.

CP: Means if there's a network issue between the databases, the request fails in favor of consistency.

AP: Means if there's a network issue between the databases, the request continues to work in favor of availability.

Choosing a database is hard. So many people use the CAP theorem to justify a database. You might need to dig deeper than just the CAP theorem in an interview. For example, if you say MySQL is a “CP” database and Cassandra is an “AP” database, what happens if you read from a follower of a MySQL with asynchronous replication? It's not really “CP” anymore. We think the best way to choose a database is to understand a list of categories of databases and what they are good at. That way, in an interview, you're able to choose one that's the most appropriate for your use case.

Sometimes the interviewer may dig a bit deeper into the internals. For example, if you claim Cassandra is an “AP” system, they may ask you why Cassandra is an “AP” system. If you claim HBase is good with chat applications, the interviewer may ask you why not MySQL or MongoDB.

These are fair questions that may come up, so you should dig deeper into each of the databases to help you with that.

Databases are very complicated and are often not an apples-to-apples comparison. In an interview, it's really about if they make sense, and how you sell your justifications to the interviewer. Databases exist to solve a particular problem, and the better you can articulate how a database solves your problem, the better you'll perform.

Also, keep in mind that you may need more than just one database for different use cases in a single question. For example, imagine you're storing YouTube videos. You'll probably need a blob store and a metadata store for the video.

Types of Databases

In this section, we will give a high-level overview of each database type. It's helpful to know the strengths and weaknesses of each database to justify the database of your choice. Although you don't necessarily need to refer to a specific database like Cassandra in an interview, it's fair to assume some high-level database categories with the fundamentals.

Interviewers generally aren't too interested in the specific technology. Instead, they're usually more interested in the fundamentals of the category of the database.

“I would use a RDBMS database since the query patterns are transactional and require non-trivial SQL queries.”

For some databases, we will also discuss a layer deeper into the internals on how the database is good at what it does if the interviewer digs deeper. We'll provide some popular databases associated with each category if you wish to go deeper. Still, the interviewer shouldn't go too deep into a specific database internally, unless you claim to have the domain expertise.

Relational Database

When to Use?

In relational databases, it is easy to represent the entities as tables and define the relationship between them. It supports transactions to write to one or many entities and supports simple to complex read queries. You're able to add in an ad-hoc index to improve the read performance at the expense of write performance. Also, if you have multiple entities with shared attributes, you can fetch results with a joined or unioned table. Also, it supports updates for a record well. Some databases are append-only.

However, it doesn't mean a relational database is efficient at all the queries. So you need to think about the query you're making and have some intuition on whether it'll be efficient for the relational database.

In a system design interview, you can consider using a relational database unless you find a better reason to use another database choice.

Reminder

Don't fall into the trap of saying relational databases don't scale. Relational databases do scale, and all big companies use them. However, relational databases may not scale for particular use cases, and there are better options, and you should be clear about why. For example, if the use case is metrics collection where there's a high write ingestion without the need to join tables and is append-only, a relational database probably "doesn't scale" for that use case.

Advanced Concepts

Most traditional relational databases use B-Tree based indexing, which is better for reads and less good for writes.

Relational databases can provide stronger data integrity through transaction guarantee by enforcing constraint through entities through foreign keys. For example, if you're designing a file system with a table folder and a table file. If the folder table has a column called number_of_files, you may have a transaction to add a file in that folder and increase number_of_files by 1. RDBMS does this well by providing transactions to this operation. If the

database increases `number_of_files` by 1 first and the operation to add a row to the file table fails, `number_of_files` will get rolled back.

Reminder

Many resources will claim that relational databases have good ACID properties, so it's worth looking into it quickly. However, ACID is a very high-level term that isn't necessarily unique to relational databases. For example, it's possible to have transactions on a row or document level in a NoSQL database. NoSQL also provides strong durability with replications. So, in an interview, don't argue for relational databases by saying it's ACID compliant and other databases are not.

Atomicity: The transaction is all-or-none. If the transaction fails, the database will roll back the transaction.

Consistency: The transaction doesn't leave part of the data committed if the transaction fails.

Isolation: The transaction keeps other transactions from accessing or writing to the same data until finished.

Durability: Guarantees the data will not be lost.

Examples

MySQL, Postgres.

Document Store

When to Use?

You should use a document database if the data you are storing is unstructured and doesn't fit well in a relational database. For example, if you're storing a product catalog where the schema between each product catalog could differ significantly, creating a relational table with all the known columns may be wasteful, since the storage will be sparse. Documents can use formats such as JSON and XML, and if your data fits naturally to those representations, then it's a good fit.

In an interview setting, the data is usually structured unless the interviewer intentionally sets it up in a way that they're looking for a document store.

When in doubt, favor relational over document since it's just easier to deal with the flexibility of different queries by having tables.

Examples

MongoDB, Amazon DocumentDB

Columnar Store

When to Use?

The columnar store has a schema that looks the same as a traditional relational database. However, columnar stores are more optimized for OLAP queries to fetch data in a column fashion. For example, assume you have a table for time_stamp | metrics_value. Then, you might query for all the metrics_value from time_0 to time_1. This query pattern is good for analytics databases and time-series databases which have a lot of overlaps.

For a time-series database that stores its data in a columnar fashion, writes happen more frequently than read. Update to any record is rare since most are append-only. When deletion happens, the deletions happen in batches instead of an ad-hoc operation like in a relational database.

In a system design interview, a good candidate for a columnar store would be an analytics dashboard. Most analytics dashboards show some sort of graph with an x-axis and y-axis. And for each graph, you will want to fetch a series of data to be displayed on the dashboard. However, this significantly depends on the dashboard query, so make sure you understand the data on the dashboard graphs before committing to a columnar store.

Advanced Concepts

Pretend you have a database table that looks like this:

<u>Id</u>	<u>Date</u>	<u>Hours Played</u>	<u>Total Score</u>
1	1/1	5	6500
2	1/2	6	700
3	1/3	3	500

In a normal row oriented relational database, each row is physically stored together:

1, 1/1, 5, 6500
2, 1/2, 6, 700
3, 1/3, 3, 500

So to get all the data for a given row is efficient. If you want to provide a time series of “date” and “total score,” you will need to access multiple rows, which is inefficient. In a columnar store, data is stored in a columnar fashion:

1, 2, 3
1/1, 1/2, $\frac{1}{3}$
5, 6, 3
6500, 700, 500

If you want to provide a time series of “date” and “total score,” you can fetch the date and total score columns. This difference becomes more apparent when there are more rows, like in time series and analytics queries.

Another advantage is the ability to compress the data for more efficient storage. As a simplified example, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 2, 2, 2, 2, 2 can be compressed to 1:8, 3:5, 2:6 and more apparent when the number of rows becomes longer.

Examples

InfluxDB, Pinot.

Object Store

When to Use?

When you’re designing to store objects like photos, videos, file documents, etc. You need to handle objects differently due to the potential amount of data taking up the bandwidth and storage. Blob stores are immutable. Once

stored, you can not change the file. You can change by inserting or appending new data.

If you try to store a blob in a relational database, you will take up a lot of memory on the table and worsen the performance of the regular OLTP transactions.

In a system design interview, good candidates for a blob store would be:

Design Instagram: For Instagram, assuming you're storing videos and images for each post, you will want to store the videos and images in a blob store.

Log Storage: When you need to store logs, you can consider using an object store to store the log dump.

File Storage: For any files that you need to store, you can use object storage. For example, if the interviewer asks you to design a system that takes in a pdf of CSV and has a data pipeline to process the files, you may want to store the pdf or CSV in the object store and have some pipeline pull from the object store to be processed.

Examples

Amazon S3.

Wide Column Store

When to Use?

Wide column store works well when there's a lot of data collected at a very high write speed. The read performance is also great when the read targets a row key that is defined.

A wide column store isn't good when you need to do joins across tables. Although a wide column store doesn't support ad-hoc modifications to a persisted value, the read query does well when it is append-only.

In an interview, a wide column store can be a good candidate for questions like:

Design Facebook Chat: The chat messages usually come into the system with a high write throughput and typically appends only. On read, it's usually reading a list of messages for a `chat_id`.

Design Metrics Collection: For the metrics collection type of question, the write throughput is usually very high because metrics are generally assumed to omit data constantly. Your system has to deal with the high rate of data collection. The read query targets a group of specific `device_id`'s that omit the metrics without a complex query.

Advanced Discussion

Here's a brief explanation of a wide column store followed by an example for a chat application.

Row: Collection of column families

Column Family: Collection of columns. Column families are loaded together and stored together.

Column: Consists of name, value, and timestamp. When the database creates a value, it also creates a new version timestamp.

In a system design interview, it's unlikely you'll need more than one column family. For a chat application, one possible schema design would be:

Row: `chat_id`

Column Family: `message`

Columns: `message_id`, `message_text`, `author_id`, `created_time`

And you cluster order by `created_time` which means the columns are sorted by `created_time` since the typical access pattern of chat is to fetch by timestamp intervals.

From the outside, the term "wide column" is just the schema. However, it's coupled with some popular databases like Big Table, HBase, and

Cassandra. Internally, those wide column databases use LSM indexing strategy, which is more optimized for writes. Also, within a column family, data are stored together physically next to each other for a given row. For example, for a chat application mentioned above, the data might look like the following for a chat_id:

```
message_id: 1, message_text: "hello" author_id: 1, created_time: t1  
message_id: 2, message_text: "sup" author_id: 2, created_time: t2  
message_id: 3, message_text: "nm" author_id: 1, created_time: t3
```

Since these records are physically next to each other, it's much faster to query.

Replication Strategy

Even within the wide column database type, there are different replication strategies. HBase uses a leader-follower replication, whereas Cassandra uses a leaderless replication. Because of the leaderless replication, Cassandra has to deal with conflicting writes with lower consistency than HBase. In an interview, you can pick one of the two based on your design requirement, and it's worth talking about your strategy on conflict resolution if you decide to use a leaderless solution like Cassandra.

Examples

Big Table, HBase, Cassandra.

Reverse Index Store

When to Use?

When designing for a search-related problem where you need to build reverse indexes, you can consider a reverse index store.

```
search(search_query) → [item]  
search(term) → [item]
```

Internally a reverse index is just a key-value store where the key is the term token, and the value is a posting list. A posting list is an abstract term for

the item you're searching for. Here's a canonical example of a search problem. Imagine you have two documents:

Document 1: "my dog house"

Document 2: "their dog"

The reverse index is going to be:

"dog" → [doc1, doc2]

"house" → [doc1]

"my" → [doc1]

"their" → [doc2]

When a search query "dog OR house" comes in, we can take the union of "dog" and "house" and combine the result, which is doc1 and doc2. When a search query "dog AND house" comes in, we can take the intersection of "dog" and "house" and combine the result, which is just doc1.

Examples

Elastic Search, Lucene.

In-Memory Store

When to Use?

Sometimes you may not need to go to disk if the requirement is to achieve better performance at the cost of worsened durability.

For example, if you need to store the driver locations for a ride-matching service, it's not critical to have durability since the location changes frequently. If you need the locations for analytics, you can write back to another database. If analytics isn't a core use case, you don't need a database here.

For more information on caching strategies, visit the caching chapter.

Examples

Redis, Memcache.

Geo-Spatial Databases

When to Use?

Geospatial databases are great to use for location-based queries. For example, given a coordinate longitude and latitude, give me all the points within a radius, polygon, or custom shape supported by the database.

```
find_point_of_interest(coordinate) → [point_of_interest]  
create_point_of_interest(point_of_interest) → status
```

Geo-spatial database is helpful for questions like the following:

Design Yelp: For a given point on the map, give me all the interesting points of interest within a particular region.

Design Find My Friend: For a given point on the map, give me all the friends near me.

Reminder

Sometimes candidates mention Quadtree as if it is a database. A Quadtree is technically an in-memory data structure, not a database.

Advanced Discussion

A geodatabase is a very niche database, so it's doubtful the interviewer will go too deep into this. Don't overly index on geospatial knowledge if you're just studying for the interview.

If you want to self-learn to go deeper, in that case, you can go into the 2dsphere index, which essentially divides up the world into multiple levels of granularity of grids and stores them into B-Tree. Another option is Google S2, which uses Hilbert Curve to enumerate the cells.

Examples

Google S2, MongoDB Geo Queries.

ZooKeeper

When to Use?

ZooKeeper is more than just a store. But it's worth talking about storage since ZooKeeper is commonly used to store configurations and name registries. In addition, you can use ZooKeeper because it provides strong consistency with good fault tolerance to the end-user.

In a system design interview, here are some places where A ZooKeeper may be used:

Shard Manager

You will have a list of physical nodes you need to call for a given logical shard. You can store this information in ZooKeeper. Knowing this might come in handy if the interviewer asks you for more detail about your sharding architecture.

Configuration Manager

If you have an application that needs to read a global configuration service that needs to be strongly consistent, you can consider ZooKeeper.

Leader Election

When you have a chosen leader-follower replication for your database when you're discussing the fault tolerance of your database, you can mention that ZooKeeper will monitor the health of your leader and follower nodes and perform an election to elect a new leader.

Advanced Discussion

The intuition is that ZooKeeper uses a consensus protocol to ensure the cluster is fault-tolerant by selecting a leader when it is down. It also uses a consensus protocol to ensure the writes are strongly consistent. Strong consistency means that multiple clients read the data as if there is just a single object. In reality, multiple machines are powering this strongly consistent property. This property is also known as linearizable. Since the configuration and name registries have low write-to-read ratios, writes are usually strongly consistent. To scale for reads, you can scale by asynchronously replicating read replicas at the expense of less consistency.

Consensus protocols like Zab, Paxos, and Raft is a very deep and complicated topic. It is unlikely an interviewer will ask you about the nitty-gritty of a consensus protocol unless you have deep experience in that area. There are videos online about Raft to build a good feel of the consensus protocol, but to ask you to prove a consensus protocol will be outside the realm of a generalist interview.

Examples

Chubby, ZooKeeper.

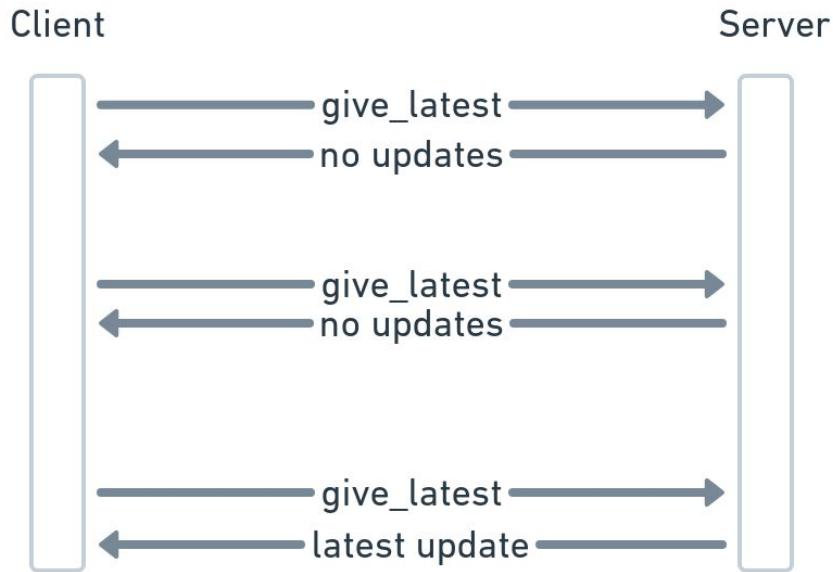
Real-Time Data Update

Use Case

Some interview questions require thinking about how end-users receive updated data from other systems and users while the end-user is online. For example, while other users send messages in a chat system, the end-user needs to see the new messages in real-time. In a notification system, the end-users need to receive a notification whenever a relevant event happens. In a time-series dashboard like a stock graph, users want to see the updated graph as new price feeds come in. Real-time data update is significant since many system interview questions involve pushing updated events from the servers to the clients. Knowing the technical details and the trade-offs will give you opportunities to deep dive.

Short Poll

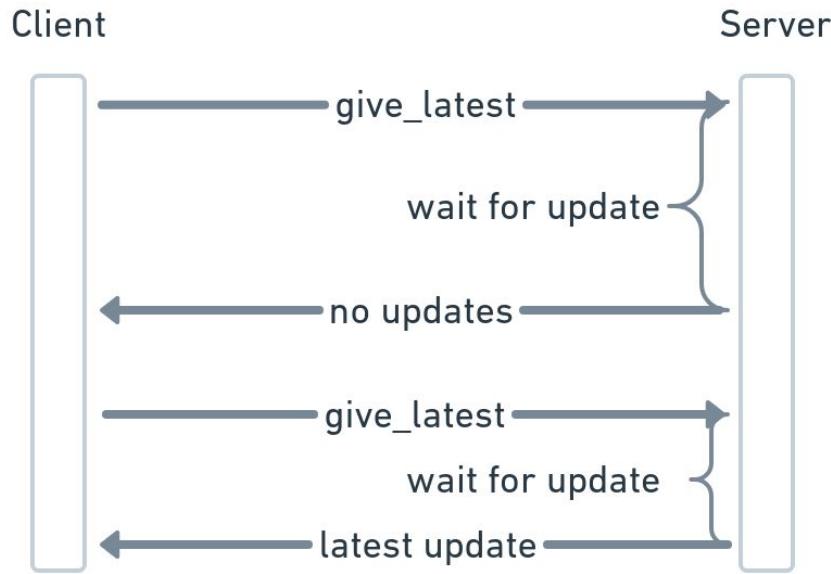
Short polling is the process during which the client sends a request to fetch updated information periodically.



Short polling is not preferable since you may be invoking many requests with no update and overloading the servers unnecessarily. The only possible reason you would ever do this is for a startup and prototype project without the overhead of maintaining server-side connection. But this is unlikely the reason for a scalable system design question.

Long Poll

In long polling, the client sends a request to the server, and the server keeps that connection open until there is data to respond with. Suppose there's no response after some time, then the connection times out. After the long poll timeout, the client can decide to send another request to the server to wait for an update. Long polling has the advantage over short polling, where the server notifies the client when there is updated data. The downside is the additional overhead and complexity to be connection aware.



Server-Sent Event

In an SSE (Server-Sent Event), the client requests the server for SSE connection. The server keeps the connection open, but the client doesn't keep the connection open. SSE is appropriate for applications like stock price feed where only the server pushes data to the client but not the other way around.



WebSocket

WebSocket is a bidirectional connection between the server and the client. The client establishes the connection with the server using a TCP connection. TCP also means the connection is stateful. The connection life-cycle is tied to a machine. When that machine crashes or restarts, the connection needs to be re-established.



How to Pick a Protocol

There are hardly any good reasons to pick the polling protocols in a system design interview unless it's for a quick prototype or you need to deliver it fast.

Since WebSocket is bi-directional, there isn't much polling, and SSE protocols can do what WebSocket can't do. Then the question is, when would you pick SSE over WebSocket, even if the communication is unidirectional?

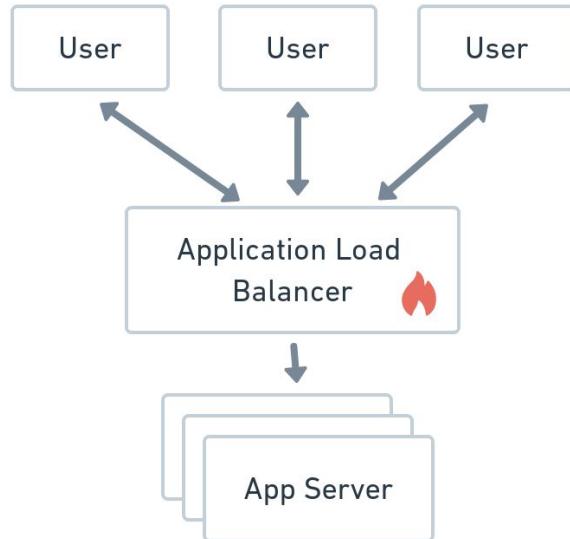
For SSE against WebSocket, you can consider SSE as less complex as it uses the traditional HTTP, which means it's less work to get SSE working than the need for dedicated WebSocket servers to handle the connections. You can opt for SSE for unidirectional events from the server to the client; otherwise, use WebSocket. Opting for a WebSocket solution for unidirectional would be fine as well.

WebSocket is the most commonly used solution for real-time data, so we'll dive deeper into some of the challenges and deeper talking points.

How to Scale WebSockets

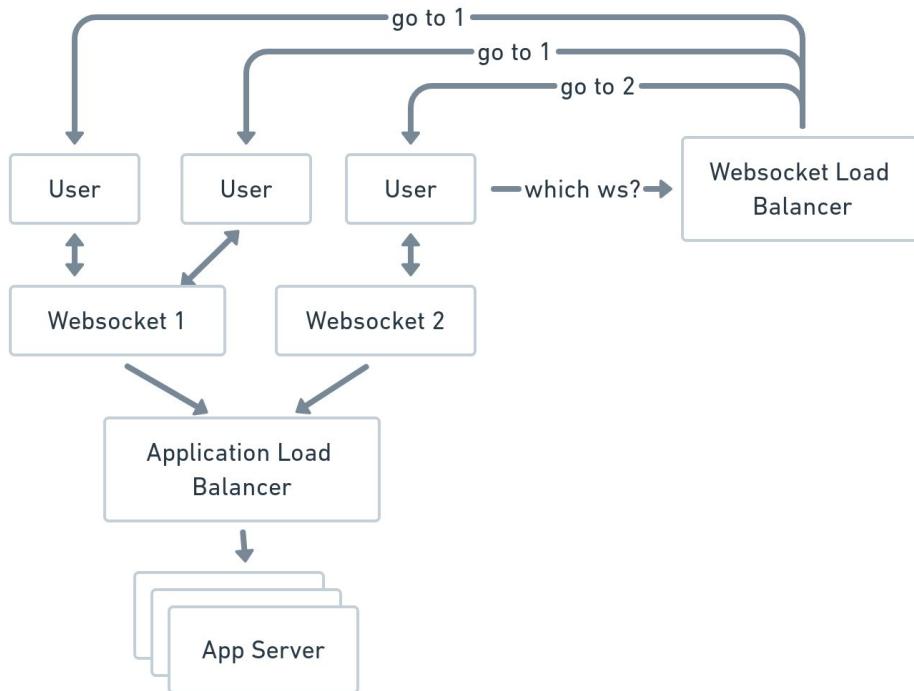
In a system design interview where you need to scale up open connections because there are millions of users, it is important to know the technical details since it's unlike a typical load balancer in front of app servers. For a WebSocket, you need to establish a stateful connection to a physical server with IP and port.

A common mistake is thinking the connection is established onto the app servers' load balancer. Connecting directly to the application load balancer doesn't scale because that single load balancer will run out of memory. You might wonder why the connection can't be on the app server. The reason is that the client establishes the stateful connection to the load balancer's machine with an IP and port. Also, maintaining an open connection to the app server's load balancer isn't what the load balancer is designed for.

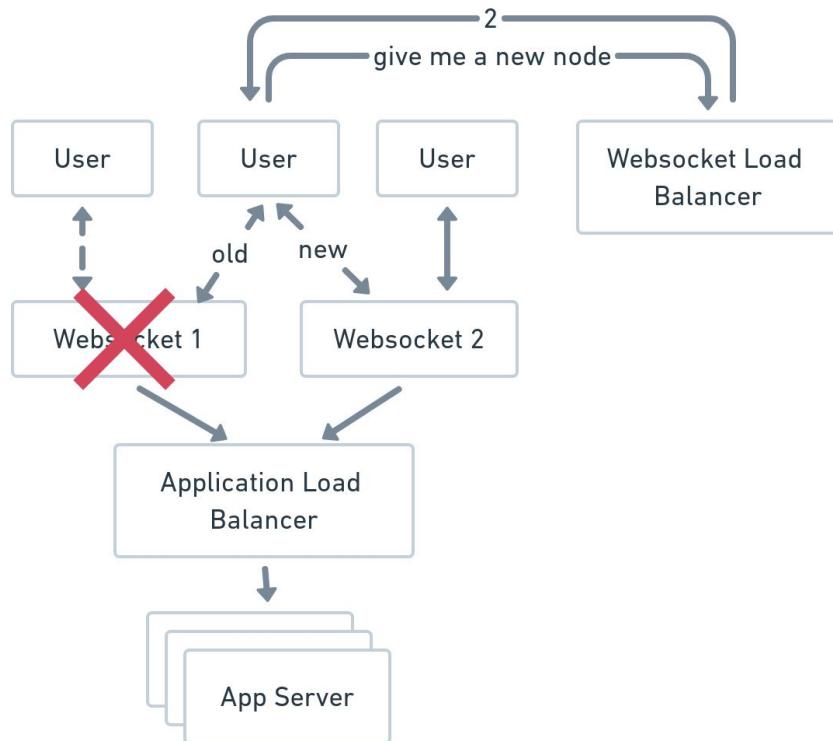


WebSocket Load Balancer

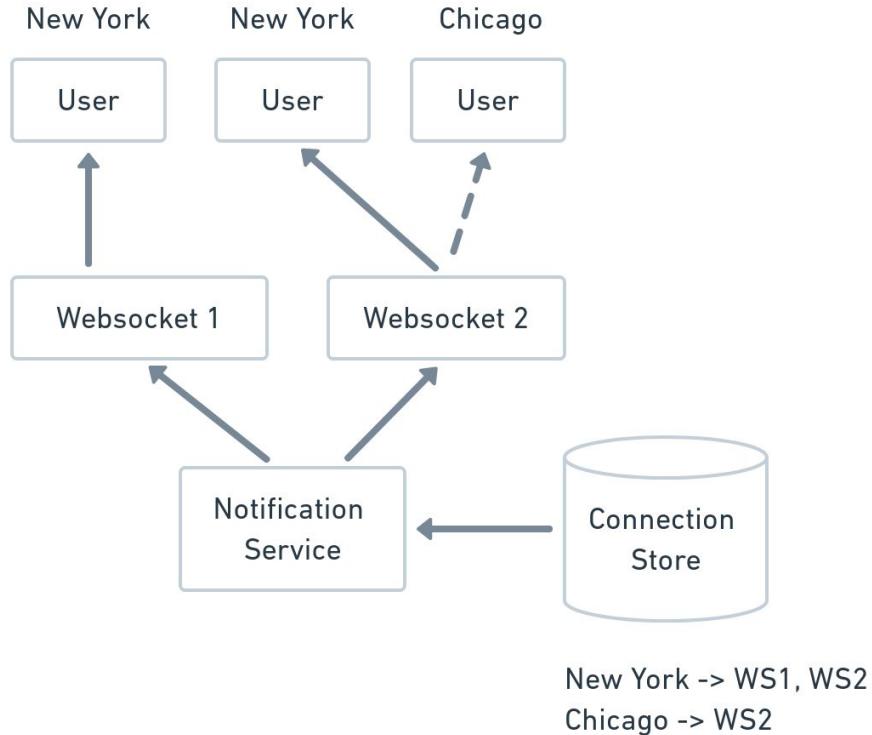
The more common approach is to have a load balancer that hands out the endpoint of a WebSocket proxy farm. Each WebSocket server maintains a list of connections to the user. A connection consists of from_ip, from_port, to_ip, and to_port. The server can use that tuple to figure out which IP and port to use to forward the events back to the client. Since a user can have multiple sessions, you can consider each connection as a session.



Since the connection to the WebSocket servers is stateful, when the WebSocket server goes down, all the clients connected to the WebSocket server need to reconnect. This disconnection can cause a thundering herd onto other servers. The more connections a server has, the bigger the thundering herd when it goes down. Conversely, the less connections a server can handle, the more servers the system needs. So this is a trade-off discussion point you can potentially have.



You need to maintain a mapping store from a user attribute to all the WebSocket servers to know which server to forward the message to. For example, say you want to notify all users who live in New York. You can have a store where the key is location and the value is a list of WebSocket servers where the user is from New York.

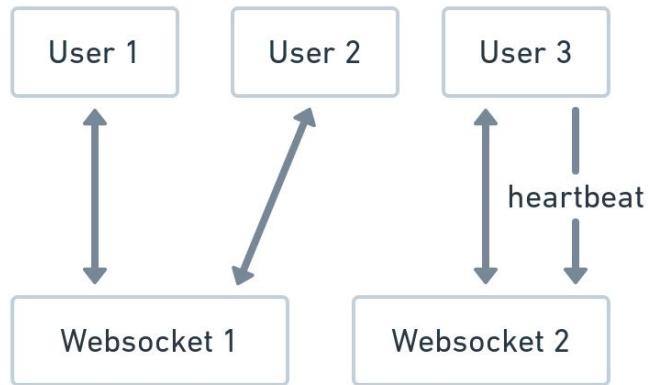


Is Connection Still Alive?

When you have an open connection via WebSocket, it is important to determine if the client is still alive. There are elegant ways to close a connection by sending a close request, but sometimes due to network interruptions the close request isn't elegantly called. When that happens, the server won't know the client has disconnected. Also, sometimes the interruption is temporary and quickly recovers. In those situations, you don't want the server to have to shut down and reconnect again.

When the client is closed, an open connection implies that the server will continue to send data to the client unnecessarily. As a result, the connection servers may be holding onto dead connections and wasting memory.

For some system design questions like Facebook Chat's user online status, it is likely they use the open connection as an indicator of online and offline. In addition, for system design questions like designing a notification server where the WebSocket connection is the core of the design, you need to think about the edge case of unexpected client disconnection.



Last Heartbeat
 1: 10:00:01
 2: 10:00:03
 3: 10:00:02

The clients send heartbeats to the server to handle temporarily interrupted connections. Thus, the server knows the client is still alive, and the client knows the server is still alive. Like in any distributed system, the browser may temporarily hang and the network may have an interruption such that the server doesn't receive the ping as frequently as they expect, but it doesn't mean the client has disconnected. The way to handle this is to have a timeout setting on the server-side. Every time the server receives a ping, the server remembers the timestamp. If the server doesn't receive another ping from the client in the next buffer_seconds, then the server determines the connection as dead. You can consider buffer_seconds to fit your needs.

For trade-off discussions in a system design interview, you can talk about the frequency of the ping and the size of the buffer seconds, and how they impact the end-user and the design. A more frequent ping will result in better accuracy of the connection status but will overload the server more. On the other hand, a bigger buffer_second will result in less toggling between online and offline when the client is still online, but results in inaccuracy if the client has disconnected because you have to wait for the buffer_second to expire.

Concurrency Control and Transaction

Definition

Concurrency happens when more than one thread is trying to access and modify the same resource.

Purpose and Examples

In a system design interview, it is important to discuss what problems can arise due to concurrency issues because it may lead to unexpected behavior if not handled properly. Concurrency is a topic many candidates avoid, yet it's a topic commonly faced in daily engineering challenges, so demonstrating competency in this area will be perceived strongly by the interviewers.

During the interview, if you have any data sources that can be read or modified by multiple requests, there is a potential concurrency problem. Concurrency is important because it affects the correctness and reliability of the system. If left unaddressed will lead to unhappy customers. Concurrency and transaction are important since they're the core of some system design questions, like meeting and ticket booking systems.

Here are some examples of where concurrency could come up in an interview:

Design a Global Counter

A problem with the global counter is the read-modify-write problem. For example, imagine two threads are trying to increment x by 1, and the current value of x is 1. Both threads read x simultaneously and see that x is 1, and both increment x by 1. Each thread thinks x is now 2 and persists x as 2. Instead of 3, the final result becomes 2.

Ridesharing Service

When the interviewer asks you to design a service that matches riders with drivers, another request may also be reading from the same list of drivers when you read from the location service for available drivers. As a result,

the matching service may assign the same driver to multiple riders because the matcher used the same driver list to match rides.

Design a Ticket Booking System

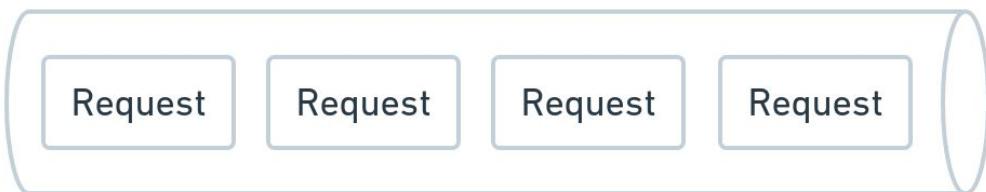
When the interviewer asks you to design a ticket booking system where each booker is assigned a unique seat number, you need to ensure you don't allow multiple users to book the same seat. Otherwise this could cause issues at the event when two people show up for the same seat.

Design a Meeting Scheduling System

When the interviewer asks you to design a meeting booking system where each meeting has a `from_time` to `to_time`, you need to ensure a room doesn't get double-booked for any of the timeslots.

Single Thread

One way to deal with concurrency is to process the requests one by one, also known as processing things serially. By inserting the requests into some sort of queue they can be processed one by one. That way, no resources can be accessed by more than one request.



The advantage of this solution is the system no longer has concurrency issues. The disadvantage of this solution is potentially poor throughput, since processing things one by one can be slow. However, in an interview, you need to make sure that's not a problem. Perhaps the QPS is very low such that it is not an issue.

Single Thread Micro Batch

Processing requests one by one may not satisfy the throughput needed for the system because for every request, you may be bounded by disk IO or some other expensive operation. One improvement would be to batch a couple of requests together and solve the problem in a batch. For example, in a ridesharing matching example, you can dispatch a couple of requests

and fetch a list of driver locations and solve the matching algorithm for multiple requests and multiple drivers in the solver, so it is still effectively one thread.



The advantage of this solution is potentially improved throughput by batching the request. Instead of going to disk for every request, we batch multiple requests into disk for only one disk IO.

The disadvantage of this solution is by having the batch request, the system may delay the freshness because we have to wait longer to buffer up multiple requests instead of processing the requests as we see them.

Partition Into Multiple Serial Processing



Another way to increase the throughput of your system would be similar to databases, you can shard your application by a request attribute, and for each shard, you can process them serially within each shard.

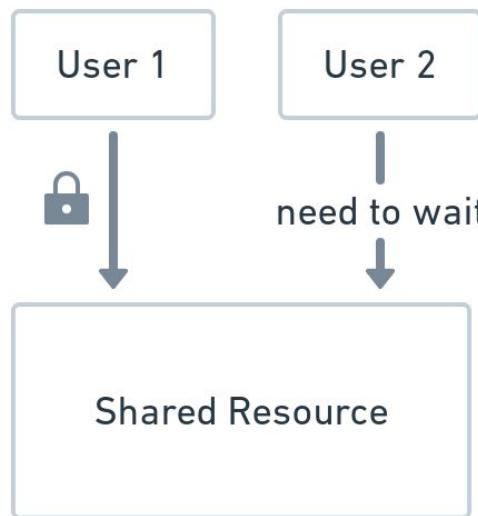
The advantage is that you can increase the throughput by as many shards as you have since each shard can process in a mutually exclusive manner.

The disadvantage is the additional complexity to maintain the sharding mapping. Another potential disadvantage is sometimes the application may

have a better user experience if more shards' data are considered instead of narrowing down the search space into a single shard. If you need cross-shard consideration, there's the overhead of cross node calls.

Pessimistic Locking

In pessimistic locking, you will need to acquire a lock to access and write to a particular resource. The lock can have different levels of granularities and can be across various entities for a given transaction. There's a difference between a read and write lock in pessimistic locking that may be important to scale your system design question.



Write Lock: Also known as an exclusive lock. The holder of the write lock does not allow other transactions to write and read the resource. An exclusive lock is very safe since only one thread can access the resource at a given time. However, the trade-off is limited throughput for both read and write.

Read Lock: Also known as a shared lock. The holder of the read lock allows others to read but not modify the resource. Internally, it will create a local copy for other threads to read while the transaction modifies an intermediate copy. The advantage of this is an improved throughput for reads since multiple threads can read simultaneously. The problem is both read and write lock still limit the write throughput.

The advantage of pessimistic locking is the simplicity to reason about the correctness of the design where you need to acquire a lock before the transaction.

A pessimistic lock is preferable in a system design interview if the read and write queries are low and correctness is critical. The disadvantage is that in a highly concurrent environment to a given resource, the throughput can be limited since both write and read locks can block write, and a write lock can block both write and read. Also, as your application becomes more complicated, deadlocks might happen if not designed carefully. In addition, for pessimistic locking, sometimes you may be bottlenecked for a rogue thread, causing other threads to wait on the lock. In this case, you'll need some timeout to release the lock, and the rogue thread just wasted other threads' time.

Scope of the Lock

As discussed previously, there are multiple scope levels for a resource, and different databases may come with varying locking features. In an interview, as long as you logically define the scope of the resource, it would be acceptable. Let's take a look at a couple of examples:

Database Lock

You can lock up the whole database such that there's only one transaction per database. Clearly this has extremely low throughput.

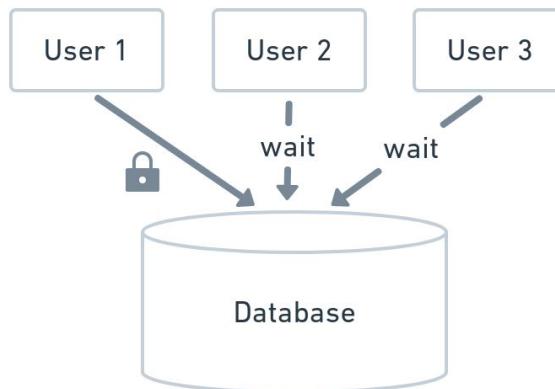
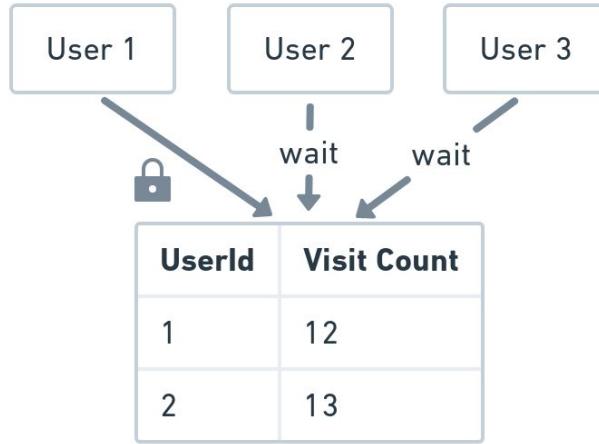


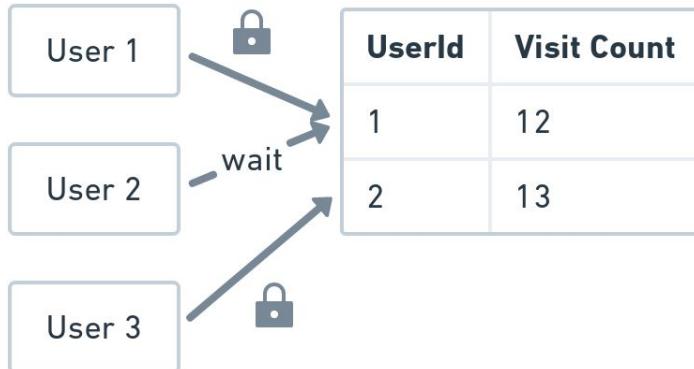
Table Base Lock

You can lock up the whole table where only one transaction can access the table at a time. This has very low throughput, but not as low as database lock.



Row Level Lock

You can lock up a particular row and only one transaction can access that particular row at a time. Row level locking has much better throughput than a database and table level lock, since multiple transactions can access multiple rows. However, this may potentially be problematic if there are many concurrent transactions to a single row.



So far, we have mainly talked about the database-level transaction and concurrency controls. Sometimes you will have data structures in the app servers and cache for a system design interview question like quad-tree, tire, concurrent queue, etc. Concurrency control may also be necessary for those data structures that would significantly impact the end-user experience. Similar to a data structure interview, you need to think logically about the type of lock and the implications.

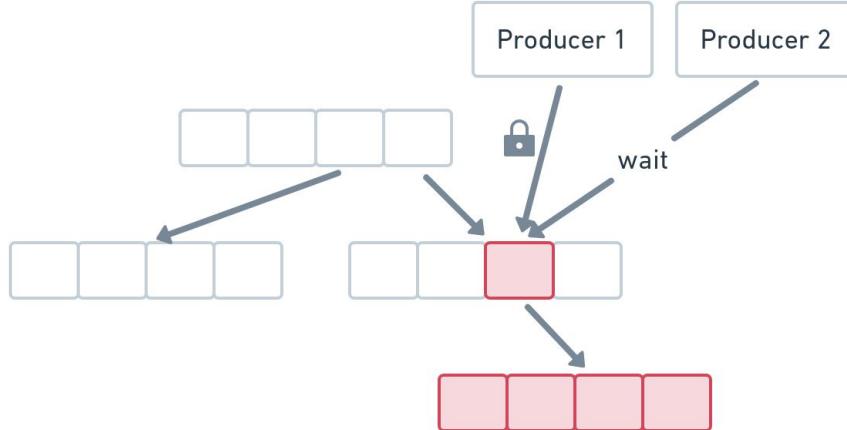
Queue Data Structure



If your system design has some sort of in-memory queue and multiple threads inserting into the queue, it's worth talking about concurrency control for that queue. You can either lock up the whole queue such that you can't insert new events while the consumer is consuming events, or have two locks, one for insertion and one for consumption.

For example, if you're discussing rate limiter and your proposed algorithm, store a rolling window with a timestamp. You should proactively discuss the throughput of this queue data structure since it's highly concurrent.

Tree Data Structure



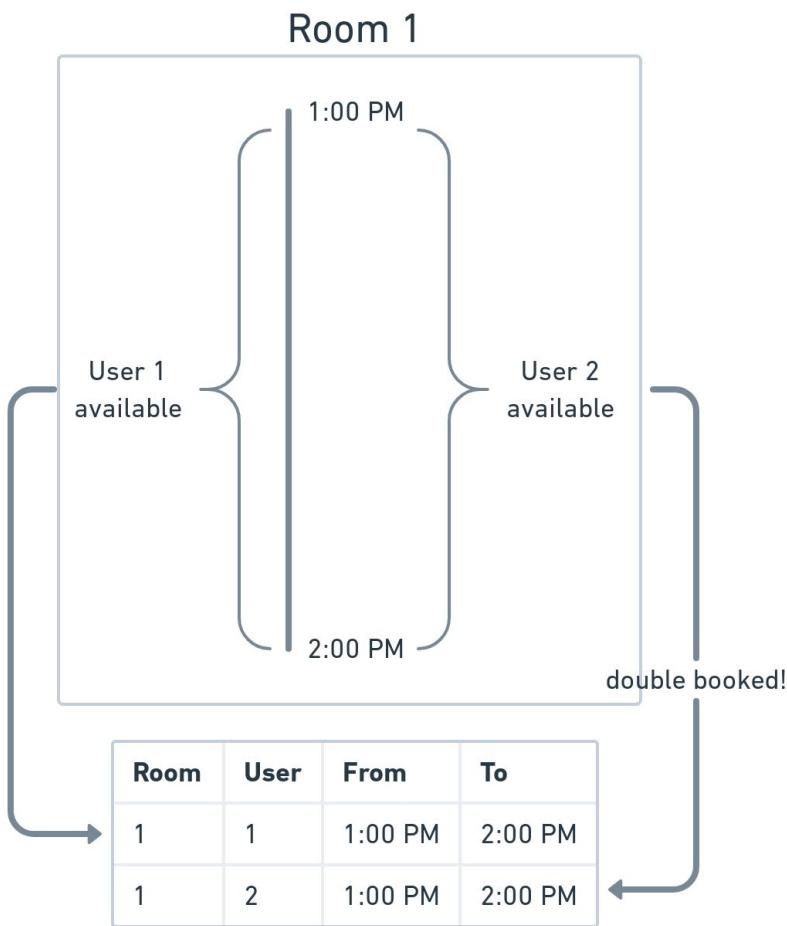
In a tree data structure, you can technically lock hierarchically as well. If a subtree is locked, no threads can access that particular subtree. Tree level locking could be a discussion point for a ridesharing service if you have decided to use Quadtree to fetch a list of drivers. You could lock up a particular region so no other threads can use that region for consideration. There are complexities and performance considerations if you are trying to acquire a lock on an ancestor of an already locked subtree. Or you can just limit the lock to a particular height level.

Other Data Structures

In a system design interview, you may develop a data structure that is well suited for that particular context and question. We can't possibly address all the data structures that may come up, but the point is to raise it to the interviewer as a potential issue if the requirement set up is such that it's a highly concurrent environment and correctness matters a lot to the end-user.

Write Skew With Phantom Data

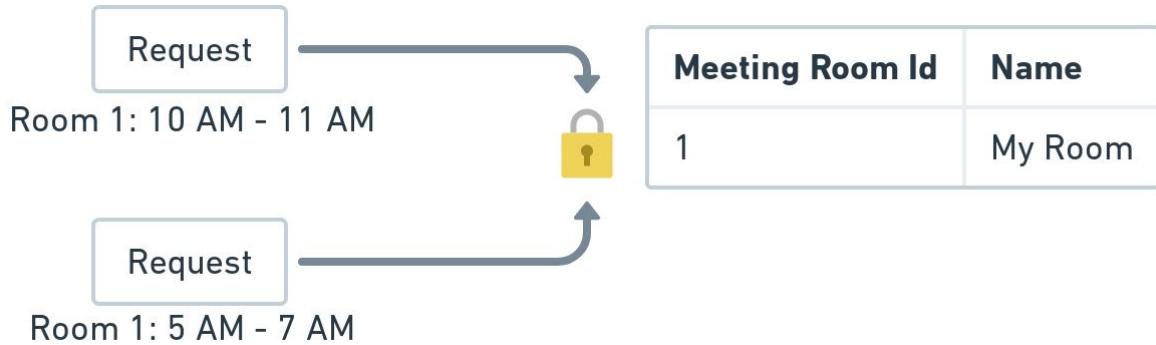
Most of the locks happen on an existing resource, either on the table's schema or the data structure. However, sometimes you may wish to lock some resources that do not logically exist in any data structures and schemas. One classic example is when you try to book a room with a `from_time` and `to_time`. While there may be a table named Room and each row is a room with a unique `room_id`, there may not be records of time to lock since time is continuous. Two threads trying to query whether a time range is available will lead to a read-modify-write issue.



Here are a couple of solutions you can mention in the interview:

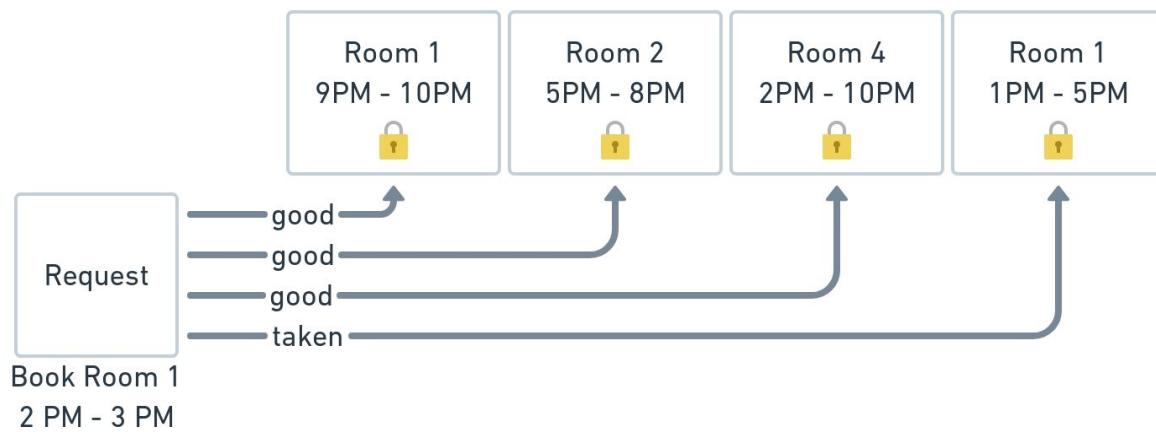
Scope Up to Lock

Instead of locking up the specific granularity, try to find a resource that exists that is a superset of all the granular resources you're trying to lock. For example, if you have a meeting room with time ranges to book for, you can lock the whole meeting room using the meeting table's meeting room row instead of locking specific time ranges.



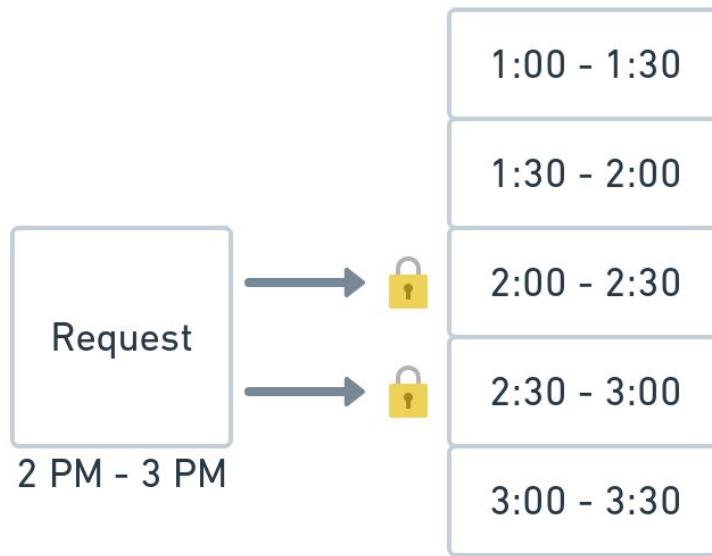
Predicate Lock

If the meeting table does not exist, you can use a predicate lock based on queries. When you're trying to occupy a meeting room, you can traverse all the predicate locks to see if there's overlap. Predicate locks can be highly inefficient when you have a lot of them.



Materialize Data

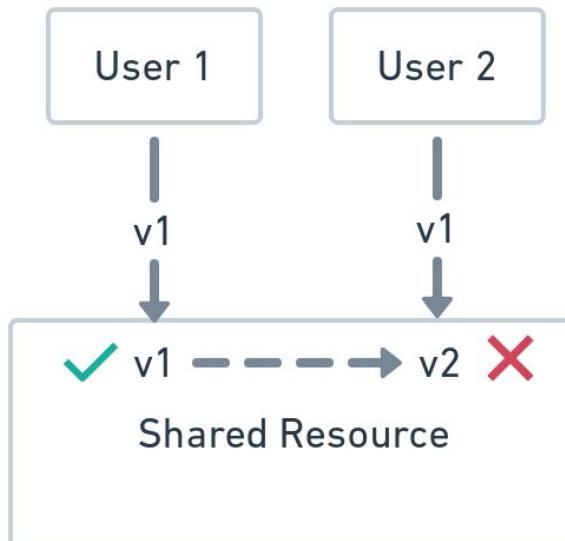
If you don't have data to lock on, you can create data to lock on. For example, if you want to lock it based on time range, you can agree with the interviewer to modify the requirement such that people can only book in 30-minute blocks, and you can create records for those 30-minute blocks. Then it becomes similar to a ticket booking service. Changing it to 30-minute blocks changes the application requirement, so make sure you discuss it with the interviewer. Also, you need to think about how you will materialize the blocks since you need to keep creating new time slots over time.



Optimistic Locking

In optimistic locking, the transaction will verify if another thread has updated the resource in question before committing the update. If another transaction has updated the resource, the transaction will fail, and the client can retry with the updated copy. Optimistic locking happens by having a version number (a.k.a eTag), and the version number is a monotonic increasing number whenever the database commits a new transaction. The version number is associated with the resource.

For now, we'll call the resource x starting from version 1 (v1). Imagine there are two threads, each reading x with v1. When one of the threads modifies x and commits to the database, the database will check the transaction's version number v1 against the current version v1 and decide no other transaction has modified and commits and increases the version number to v2. When the second thread comes in with v1, when the database tries to commit, it checks that v1 is outdated against the current version number v2 and fails the request. The second thread will have to load v2 and modify again.



The advantage of optimistic locking is that when threads try to access a particular resource, it doesn't have to wait on a lock like in pessimistic locking, it continues to apply its business logic and persists back to the database. The throughput will be better without the overhead to acquire and release the pessimistic lock.

The disadvantage of optimistic locking is in a highly concurrent environment where many threads access and write to the same resource it can lead to many failures and retries. These failures may bubble up to the end-users and lead to poor end-user experiences.

Change Product Requirement to Simplify

In a system design interview, instead of being tunnel-visioned into the “obvious” way to do a certain thing, sometimes you can take a step back and try to reframe the question and ask yourself why you’re doing it in the first place. For example, if you have a globally distributed counter, instead of having a distributed database with shared memory, you can switch the solution to keep a log and conduct a batch job for the count. The downside is a much longer delay than having an up-to-date variable, but perhaps that’s acceptable for the requirement you’re trying to design for.

In a ridesharing design, let’s say you decided the requirement is the rider can choose from a list of drivers. What happens if multiple riders request the same driver at the same time? You have a concurrency problem. You

can always change the requirement to not allow riders to select the driver, claiming the concurrency is too complex to deal with. Instead, the system can choose the driver for the rider. Not only is the user experience simplified, but the technical complexity is too. Sometimes technical complexity is a result of product complexity. It's always good to take a step back to rethink the product experience.

How to Pick a Concurrency Strategy

Concurrency is a difficult topic, but we have covered most of the tactics you need for a generalist system design interview, unless you have very deep experience in database transactions and distributed locks, in which case the interview becomes more domain-focused.

When faced with a problem, our suggestion is to brainstorm a couple of options and think through the pros and cons with respect to the design requirements. Don't generically discuss the pros and cons, but focus on what the end-users would experience based on your suggested solution. The interviewer isn't necessarily looking for your ability to come up with the best solution right away, but to see your ability to present options and trade-offs.

Replication

Definition

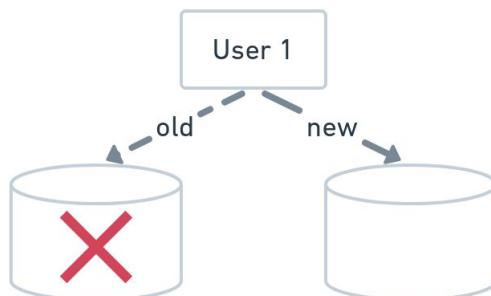
Replication is the process of copying data from a data source to other data sources. For example, imagine database A has records 1, 2, and 3. When database A finishes replicating to database B, database B will also have records 1, 2, and 3.

Purpose and Examples

Before we get into the details of some of the replication strategies, it's worth talking high-level on why replication is needed. We will use databases as examples, even though other data sources like cache or app servers can apply replication strategies as well.

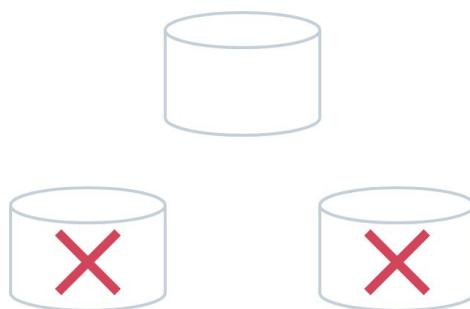
Improve the Availability of the System

When one database is down, the system can use another database to serve the live traffic.



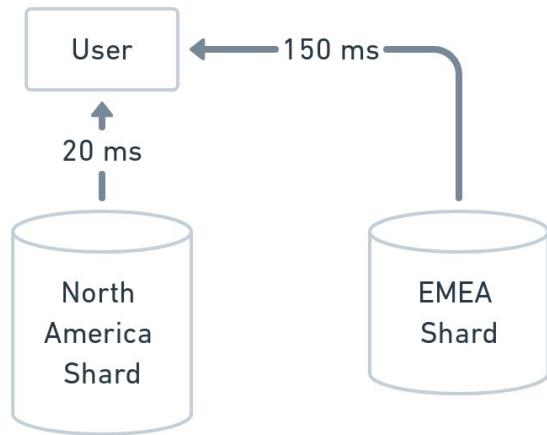
Improve the Durability of the System

When databases fail and break, there are replicated databases with the same data, so the data isn't lost forever.



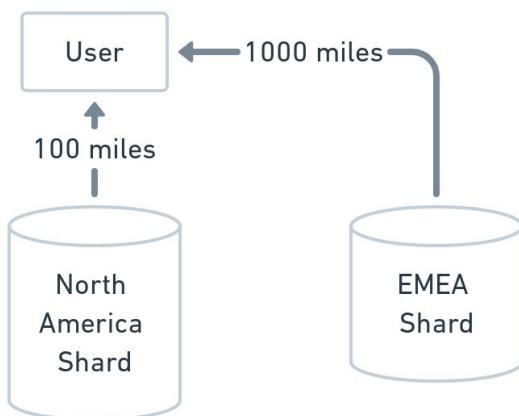
Improve the Latency of the System

The system can replicate databases to another data center or point of presence such that the data is closer to the user. By being physically closer to the user, there's less distance the data has to travel to reach the user.



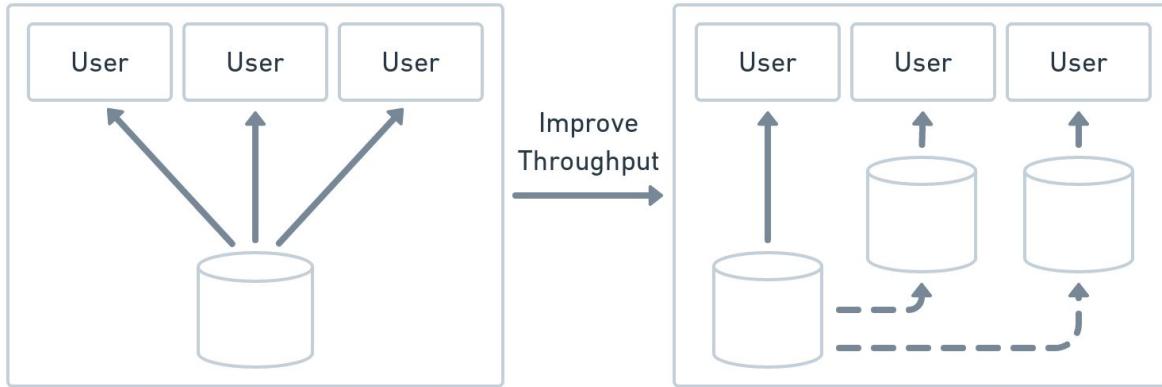
Improve the Bandwidth of the System

When the data is far away from the user, it is more likely the bytes will need to travel through more networks and encounter a network that creates a bottleneck. Imagine placing a dinner delivery 1 mile away vs. 100 miles away. The delivery person needs to incur more time and a higher gas fee for 100 miles. That is true with CDNs as well when the origin server is further away from the user.



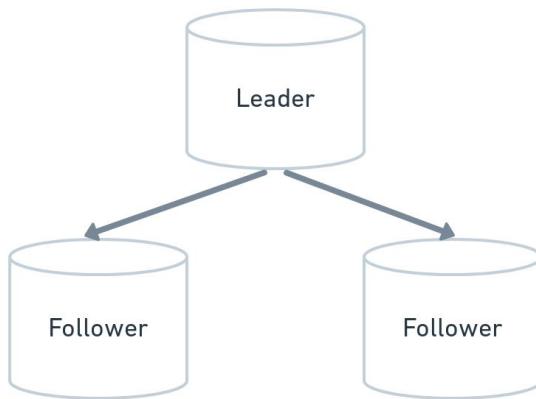
Improve the Throughput of the System

With replication, there will be more databases with the same data, and the system can handle more requests since there are more databases.



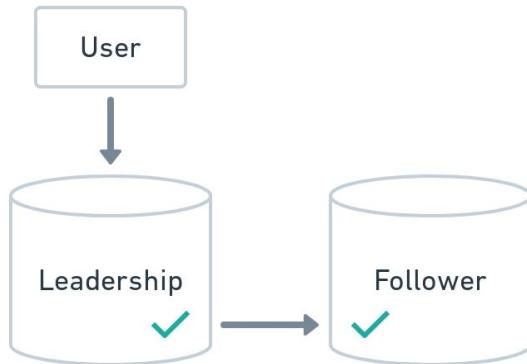
Leader-Follower Replication

When the query writes to the leader in a leader-follower replication, the system will replicate to one or many followers. Read queries can happen on any of the databases. There are two types of replication: synchronous and asynchronous.



Synchronous Replication

For synchronous replication, the write query to the leader must wait for the leader and follower to commit before claiming the write query is successful.

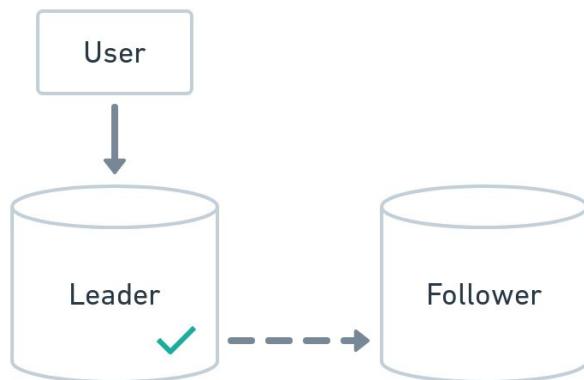


The advantage of synchronous replication is the follower will have up-to-date data with the leader when the write query is successful.

The disadvantage of synchronous replication is that it is slow and relatively unavailable. Imagine you do synchronous replication to two followers. If either of the followers is located far away, the latency will be high and unpredictable. If either of the followers is unavailable, the write query will fail and lower the overall availability.

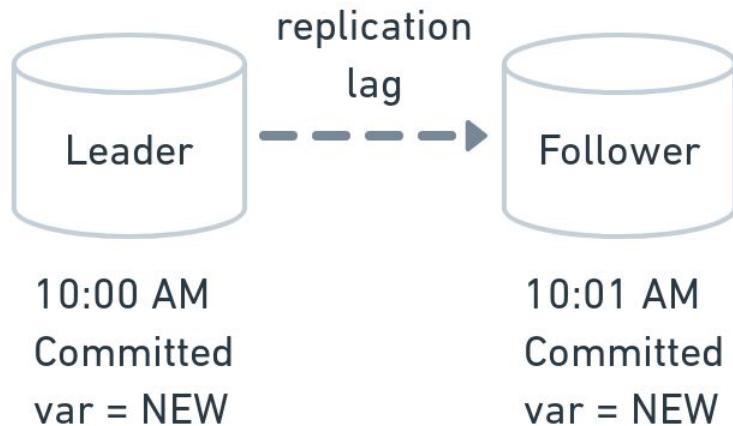
Asynchronous Replication

For asynchronous replication, once the write query is committed to the leader, it fires and forgets to replicate to the followers and commits the write. Because you no longer have to wait for the followers to commit, the write is faster and more available. Furthermore, if any of the followers fails, the write query is still successful.



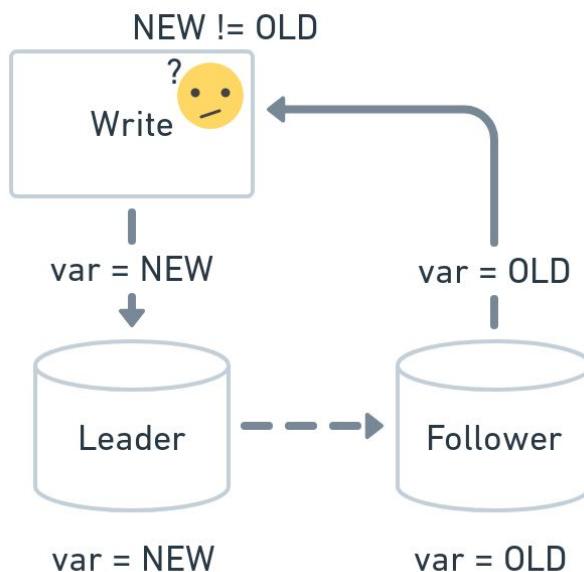
The advantage of asynchronous replication is a faster transaction since you don't need to wait for the follower to commit.

The disadvantage of doing so is inconsistent data between the leader and the followers. If I have a user querying any of the databases at any given time, I may get a different result due to replication lag. Replication lag means it takes time to copy a piece of data from the leader to the followers.



In practice, asynchronous replication is still commonly used. However, if you do decide to go with asynchronous replication, you need to think about the following:

Read Your Own Write

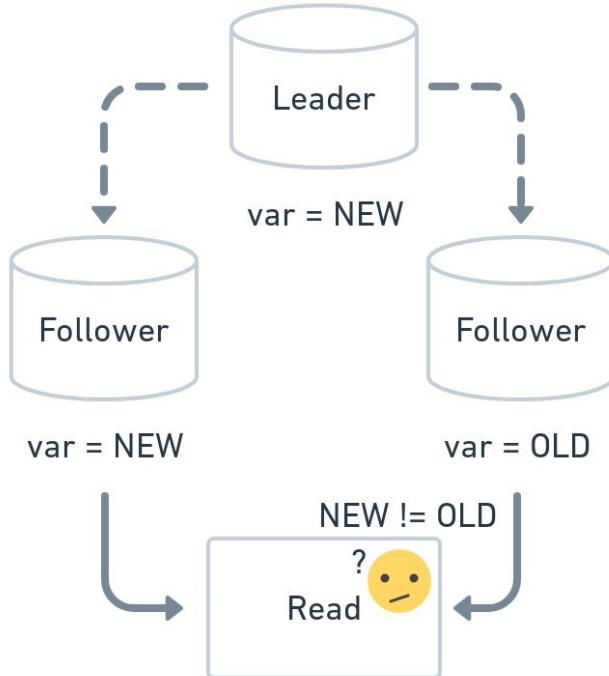


If you write to the leader and read from the follower, you may design a bad user experience where the user feels like they haven't committed because the newest data has not made it to the follower yet.

In the system design interview, think about the implications for the end-user. For example, for photo upload, if you claim the metadata query is

from read-replicas and a user just uploaded an image, you may run into a replication lag where the user doesn't see their newest image.

Inconsistent Read from Different Read Replicas



You might get inconsistent results from different read replicas if you have a load balancer that round robins between read replicas. This might be acceptable for some applications but not for others.

Follower Failure

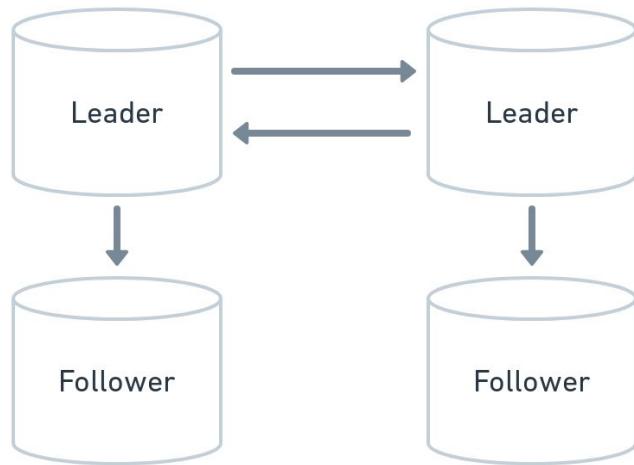
If the follower fails to respond, the system needs to detect the follower is faulty and stop forwarding requests to that follower. However, this may lead to a snowball effect if not careful. For example, if the followers are close to capacity, forwarding additional traffic to other followers may bring down more followers.

Leader Failure

If a leader fails to respond, nobody will be available to handle writes. If you're designing an available system in a system design interview, you need to talk about handling writes when the leader fails. Once you've

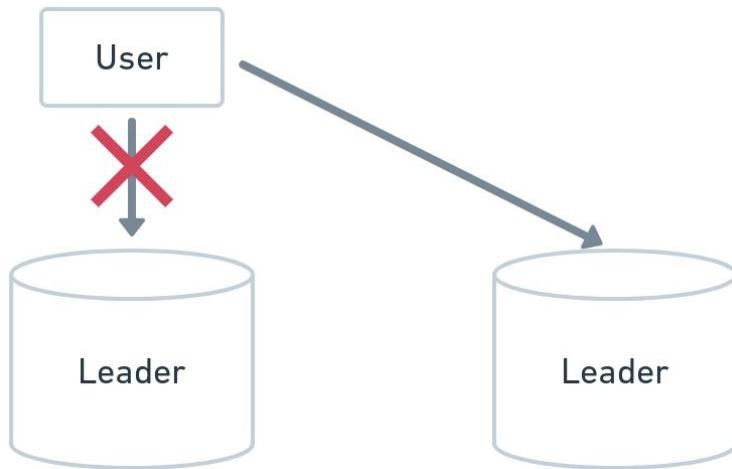
identified that the leader is down, you can either manually configure a follower to be the leader or perform a leader election using quorum. The downside to picking a new leader is that the process takes time. The system could be down for a couple of seconds or more because it takes time to determine a leader is down using timeouts and for a quorum election to take place. This delay might be unacceptable for some systems.

Leader-Leader Replication

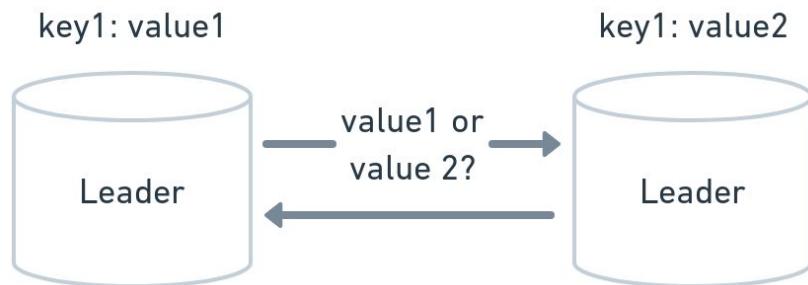


The problem with a single leader setup is that when the leader is down, nobody will be available to take writes until a new leader is available. In a leader-leader configuration, there will be multiple leader nodes to take writes. The system will replicate the data to each leader to keep them up to date. Each leader can still have follower replication for backup and reads.

The advantage of this approach is the writer will be more available since if one leader goes down, the other leader can take over. However, note that the performance will probably worsen because the new leader might be further away. Also, you have to deal with replication lag where the new leader might not have the most updated data. Another advantage is that if the leader is closer to a user, the latency will be faster than other leaders.



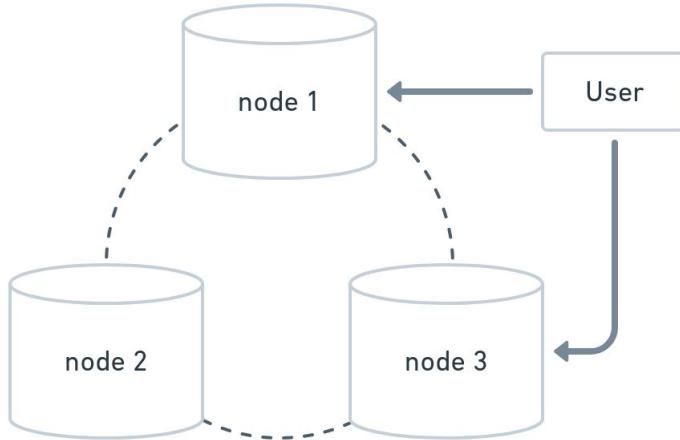
The disadvantage of this approach is the complexity of conflicting data. For example, if users are writing the same key to each of the leaders, while there are multiple ways to resolve the conflict and the solution depends on the application, there is additional complexity the engineers have to worry about. Similarly, if the leader goes down, a new leader has to be selected to take write.



How do you know which one is correct? Like the following, should it be value1 and value2?

Leaderless Replication

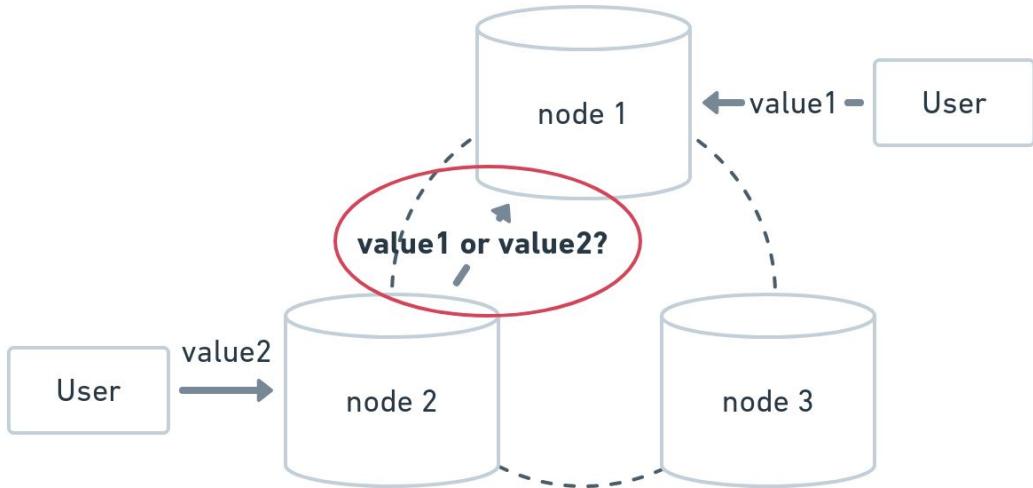
In a leaderless replication, a write request (also known as quorum write) is committed to some replicas, and if at least w nodes are successful, the main write query is successful. A read request (also known as quorum read) reads from some of the nodes, and at least r nodes are successful before the main read query is considered successful. For example, take a cluster of 3; if the user configured w to 2, 2 of the nodes need to acknowledge success before the write request is successful. If any one of the nodes is down, the write can continue to operate. The algorithm is true for read query as well as defined by r .



w and r are tunable numbers where the higher the w and r , the greater the probability your read request will read up-to-date data. For now, imagine n is the number of nodes in the cluster. If $r = n$, you're guaranteed to find the latest data, whereas if $r = 1$, you are more likely to randomly hit a node that might not be up to date. Similarly, for w , if you always write the data to all the nodes with $w = n$, even if r is 1, you're guaranteed to read the latest value. Since $w + r > n$ has a stronger guarantee for up to date data than $w + r \leq n$.

The advantage of doing leaderless replication is you don't have to worry about leader selection and election when the leader is down. In addition, the cluster can continue to take writes and reads even when some nodes are down. Thus, the leaderless leads to better availability.

The disadvantage of doing leaderless replication is you have to deal with the complexity of data consistency. Multiple requests can write the same key to multiple nodes, similar to multi-leader replication. As the write gets propagated to different nodes during quorum write, it's unclear which update should be the winner. In an interview, you should talk about the conflict resolution strategy of your design if you chose leaderless replication.



How to Pick a Replication Strategy

Now that we've covered a couple of replication strategies, we will go over some examples of applying this knowledge to a system design interview chapter. Whenever you're dealing with data sources, it's an opportunity for replication strategy. Replication strategy doesn't have to apply to databases. It can apply to cache servers or app servers that have in-memory data structures. Generally speaking, you should list them out as options and think about the trade-offs using the knowledge above.

What Should Be My Replication Factor?

Generally, the industry average is 3, but you should talk about the trade-off in an interview. More means more cost to maintain databases, and if it's synchronous replication, it can lower the query performances. On the other hand, a lower count means a lower durability guarantee with fewer replicas as a backup.

Sharding Strategies

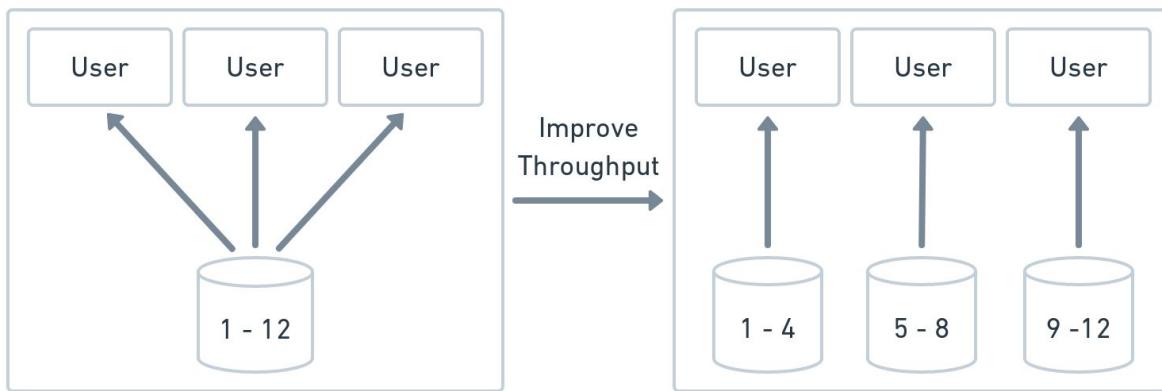
Definition

Sharding is dividing the data into smaller chunks so that each chunk lives on different servers. It also need not be storage-related, since you can also shard app servers and cache as well.

Purpose and Examples

Improve the Throughput of the System

You can have multiple shards that take write instead of a single shard. As long as the shards are *share-nothing*, the throughput is improved.



Improve the Capacity of the System

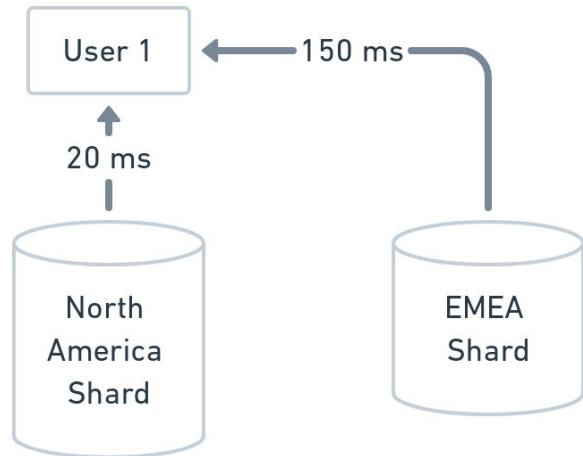
Assume each database can only store up to 1 TB of data. When you shard, you're able to store more than 1 TB of data.



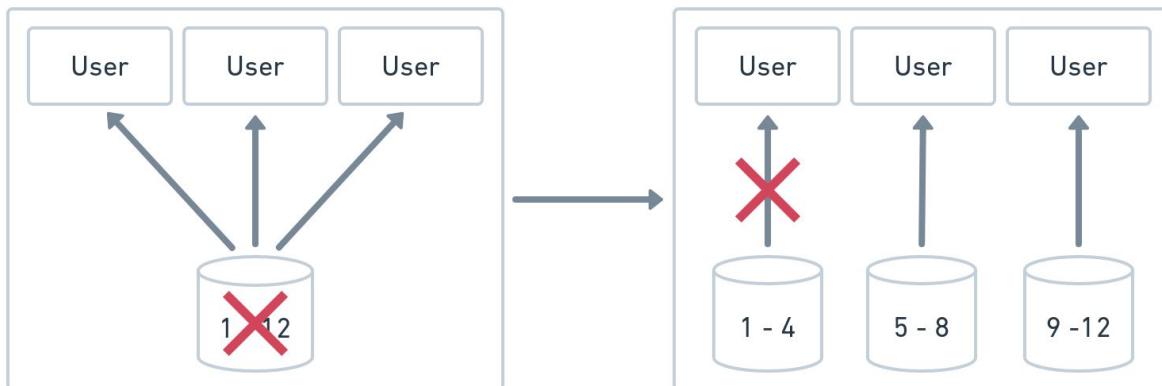
Improve the Latency of the System

If there's only a single shard with a single leader to take in writes, there will be additional latency if far away users need to route their request to that shard. By sharding, you can create a local shard to take in local writes and lower latency. Another reason for lower latency is when each shard has

fewer data. When a database has less data, querying will be faster since there is less data to process.



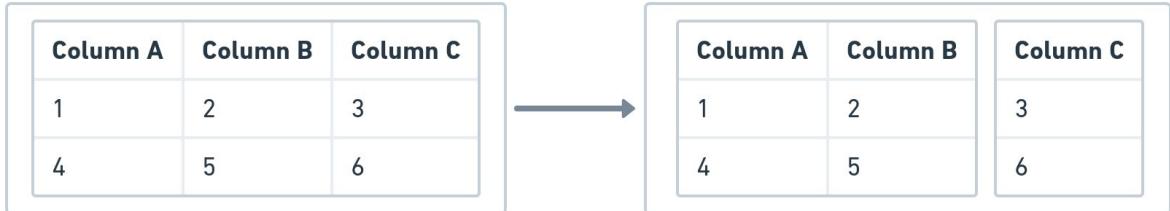
Improve Perceived Availability



When there are more shards and a shard goes down, only the failed shard is impacted. If there's only a single shard, the whole application will be down. So even though with more shards, the chance of partial failure is more likely, the variance of a disastrous complete failure is less due to a correlated failure such as a database going down. Note that this doesn't improve overall availability, however.

Vertical Sharding

When a database has too much data (more than the storage can handle), you can shard your database by migrating some table columns into new tables due to differences in query patterns and storage capacity.



The advantage is that this reduces the amount of data for a given table and the amount of storage needed if a column is sparse.

The disadvantage is you'll need to make multiple updates for the same key to multiple tables and joining the tables on read is comparatively more expensive. It's unlikely you'll have to worry about a vertical sharding optimization since you typically don't have that many columns to even talk about. However, it's worth knowing in the event that the interviewer sets it up so that you already have many columns, and asks you to think about scalability.

Horizontal Sharding

When a database has too much data, you can shard by dividing up the rows into multiple different tables. As the system adds more rows to a table, it will eventually run out of space, memory, and CPU to handle the queries. Having a shard to handle parts of the request will reduce the burden on a single global shard.

In an interview, before you decide to shard your application servers or databases, you need a reason to do so. For example, you can calculate how much total memory you'll need in the next few years and check if a single database can handle that capacity. If not, you need to shard. To improve the throughput of your database, you can calculate QPS and determine if a single database can handle the QPS, then decide if you need to shard. Similarly, if latency is an issue, you can potentially have a geo-shard to move the database closer to the user.

Sharding is just one of the many solutions to fix the problems discussed, and you shouldn't immediately jump into sharding as the only solution. For example, if total storage is an issue, you can move some hot data into cold storage. For bandwidth issues, you can use compression. Finally, if QPS is

an issue, you can potentially batch the query or reduce the calling client. So consider sharding only if it makes the most sense.

When you decide you need to shard, it is not enough to just say you want to shard the database horizontally and expect everything to magically work. Ideally, if you followed the system design framework, you would have the database schema already. It is essential to give at least 2 sharding scheme options and discuss the trade-off. The interviewer isn't looking for your ability to come up with a single solution, but the ability to come up with options and trade-offs.

Hash Key

One way to shard is to take a hash of an attribute and distribute it to a shard. Say your hash function generates a number between 0 to 2^{32} , and you want to scatter your data across 4 shards. You might distribute it by saying data with $0 - \frac{1}{4} 2^{32}$ goes into the first shard, data with $\frac{1}{4} 2^{32} - \frac{1}{2} 2^{32}$ d goes into the second shard, and so on.



A famous algorithm called *consistent hashing* deals with how another shard can pick up shard failures without thundering herd impact by transferring all the data from one shard into another shard. It is fine to mention consistent hashing in an interview, but be prepared to be quizzed about the whys.

A hash-based sharding scheme's advantage is that the system will distribute the data well across shards to minimize any hotspots. However, we will further discuss below that it doesn't completely solve the problem if there's an abnormally hot key.

A hash-based sharding scheme's disadvantage is there's no relationship between the keys within the same shard. Sometimes when you make a query, you might be fetching for multiple rows, and you might need to get

your query from multiple shards. If you need to do a range query by the sharding key, this might not be the right solution. However, once you are within a shard, you can add primary keys and indices as you normally would optimize for queries for that shard.

Consistent Hashing

A common problem with hash based partitions is rehashing. Let's assume a simple hashing scheme $\text{hash mod } N$. If I have a key 100 with 10 servers, it will go to server 0; if I have 11 servers, it will go to server 1; if I have 12 servers it will go to server 4. The rebalancing will be a disaster. A common way to solve the problem is through consistent hashing.

Reminder

A lot of candidates mention consistent hashing like it's the silver bullet to all problems. It's not. It has trade-offs based on the data distribution, queries, and hotkeys.

Problems Consistent Hashing Tries to Solve:

1. Distributing the keys more evenly across shards.
2. Minimizing data migration due to shard redistribution when servers are added, removed, and down.

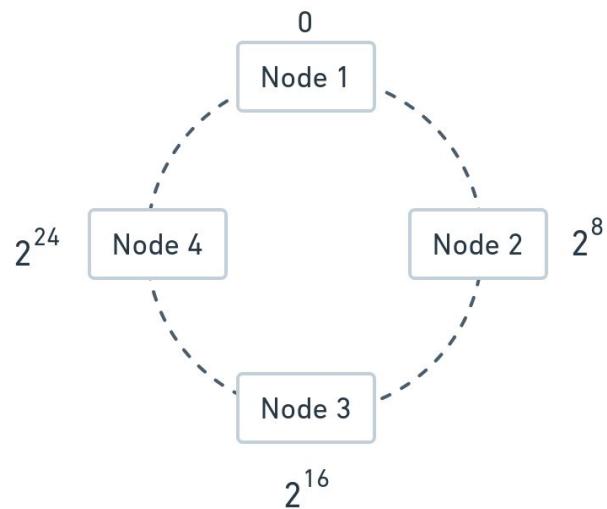
Problem Consistent Hashing Does Not Solve:

1. If a key is super hot targeting a shard, that shard will still be hot. Imagine 99% of the traffic goes to that key. That shard will be on fire.

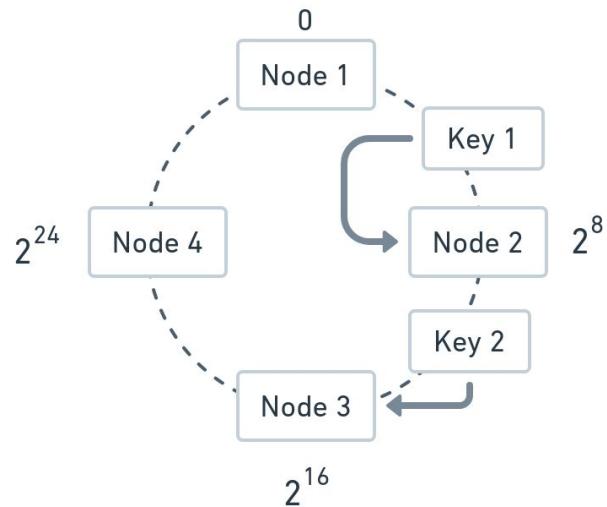
Algorithm

We will touch on the algorithm here since it's a well-known algorithm. But in an interview, know when and why you need consistent hashing, don't start retelling how consistent hashing works unless it's important to the context of the question.

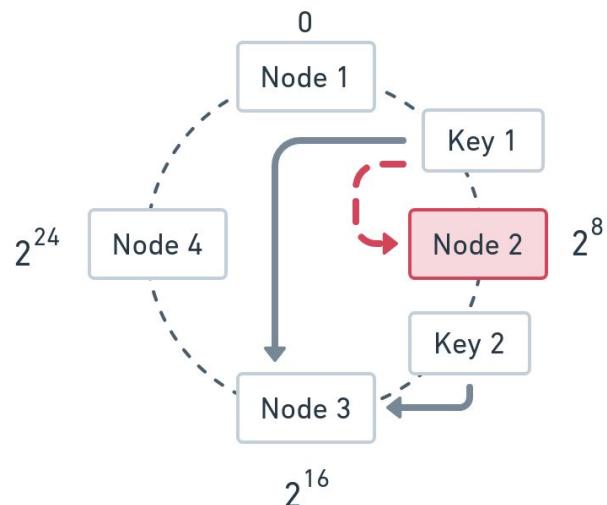
Concept 1: Hash Keys are Distributed in a Circle, say 0 to 2^{32}



Concept 2: Keys Within the Range Go to the Next Node

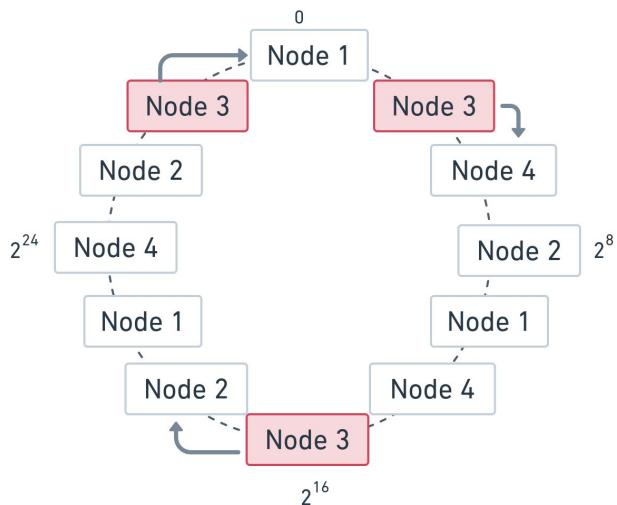


Concept 3: If a Node Fails, the Next Node Picks it Up



Concept 4: To Prevent Thundering Herd, Add More Nodes

In concept 3, imagine there are many keys between node 1 and node 2 and Node 2 goes down, then all the keys will flood to node 3. You can add more virtual nodes in between so the impact of a failed node won't have as big of a thundering herd effect.

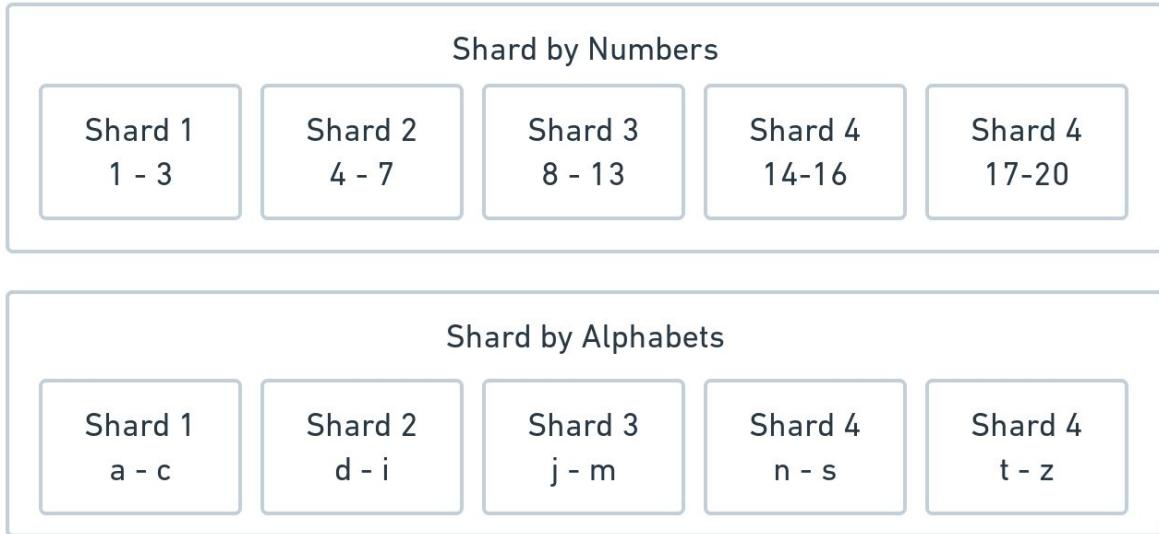


As you can see, adding more nodes helps distribute the data more evenly throughout the keys. Failures in one node will be more evenly distributed across the next nodes. If Node 3 fails, nodes 2, 1, and 4 can pick up Node 3's data.

Range Key

Building on top of the disadvantage of hash-based hashing, applications may need to ensure a range of keys live on the same shard so the query

won't need to do a big scatter-gather. The keys are sortable for the range key sharding scheme, and once sorted, the algorithm assigns each range to different shards.



The advantage of this approach is that a query becomes efficient when looking for data within the same shard. A common usage is to shard by timestamp so when the user queries for data with the latest timestamps, the query can get it from the same shard as opposed to scatter-gather from multiple shards.

The disadvantage of this approach is the likelihood of hotspot on writes and reads. For example, when you generate events with timestamps, all the event writes and reads will go into the same shard if you shard it by time buckets. However, it is possible users may only query for historical data such that read does not become a factor for hotspot. It is important to understand the query pattern in an interview.

Let's take a look at a couple of examples:

We have a system where we store user tweets and each row in the tweet table has `tweet_id`, `user_id`, `timestamp`, and `tweet_message`. For simplicity, we will ignore the date for now. The common queries are:

Write query:

`tweet(user_id, tweet_message)`

Read queries:

`get_tweets_for_user(user_id, from_time, to_time)`
`get_all_tweets(from_time, to_time)`

Option 1: Shard it By Tweet Timestamp

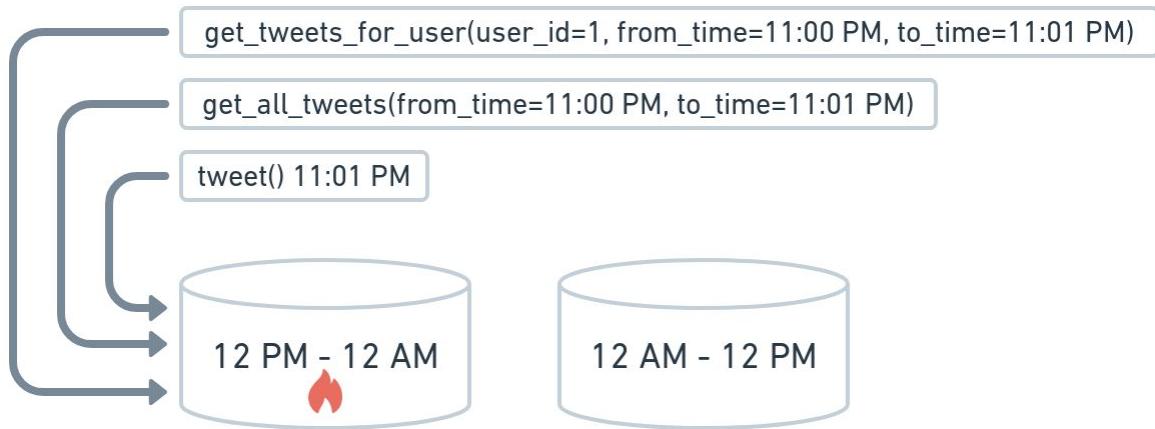
For a given hour time interval, store it in a shard. For example, from 12 AM to 12 PM PST, the tweet goes into shard 1, and from 12 AM PST to 12 PM, it goes into shard 2.

Assume right now it is 11:01 PM.

For the tweet write query, it will go into one of the 2 shards based on the current time for all the users; this could lead to a hotspot.

For the `get_tweets_for_user(user_id=1, from_time=11:00 PM, to_time=11:01 PM)`, we only go into one of the 2 shards regardless who the user is.

For `get_all_tweets(from_time=11:00 PM, to_time=11:01 PM)`, we will just need to go into one of the 2 shards.



Option 2: Shard it By User ID and Tweet Timestamp

The sharding key is now `user_id + tweet_timestamp`, so the same users are grouped. For now, assign (1, 12 AM) to (5, 12 PM) to go into shard 1, and (6, 12 AM) to (10, 12 PM) to go into shard 2.

The tweet write query will be more evenly distributed across the two shards based on the user ID for a given time.

For `get_tweets_for_user(user_id=1, from_time=11:00 PM, to_time=11:01 PM)`, we will load it from one shard.

For `get_all_tweets(from_time=11:00 PM, to_time=11:01 PM)`, we will need to fetch from all the shards because users are distributed across shards.

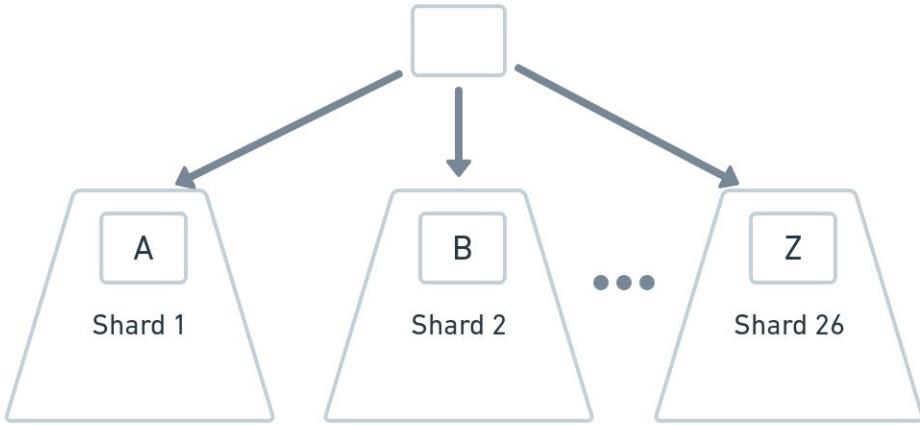
Final Recommendation

Choosing between the two heavily depends on the assumptions you make about the three API calls. The purpose of an interview is to list out options and discuss the trade-offs and make a final recommendation. For example, one recommendation would be to assume `get_all_tweets` to be a less frequent query and go with option 2 because write becomes more evenly distributed and the trade-off of a big scatter-gather is not that significant of a problem.

Other Data Structures

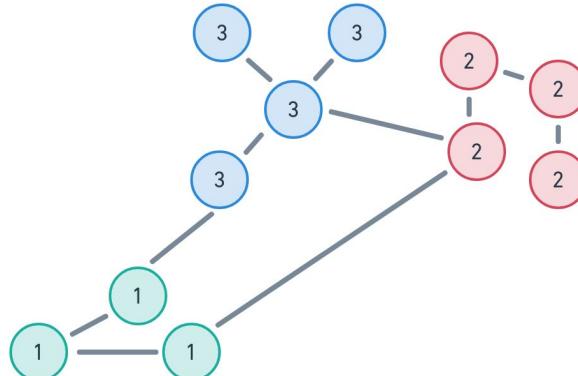
So far, we've talked about sharding in the context of database rows. As discussed previously, there are data sources other than database rows. Sometimes you may have a tree structure, grid structure, graphs, hashmap, etc. The same rules apply: Think about the read, write, and data distribution patterns to justify your solutions and trade-off and make a final recommendation. We will show a couple of graphical examples to demonstrate how you can shard the data structures.

Tree



Similarly, you need to think about how much data you store in node A and if you need to shard it further if the A subtree becomes too big. For example, if you frequently query a prefix in a trie, how do you reduce the hotspot? The sharding scheme is useful for tree-based data structures like an n-ary tree, quadtree, and trie, which show up frequently in interviews.

Graph



You need to think about a given shard if the nodes within that shard hold too much data and if the read and write queries are too overwhelming for them. The graph is useful for location-based questions and social graphs. For graphs, it's worth thinking about the proximity of data between shards since, in a graph-based model, a common query is to look for adjacent nodes for a given node.

Grid

shard 1	shard 1	shard 2
shard 1	shard 1	shard 2
shard 3	shard 3	shard 4
shard 3	shard 3	shard 5

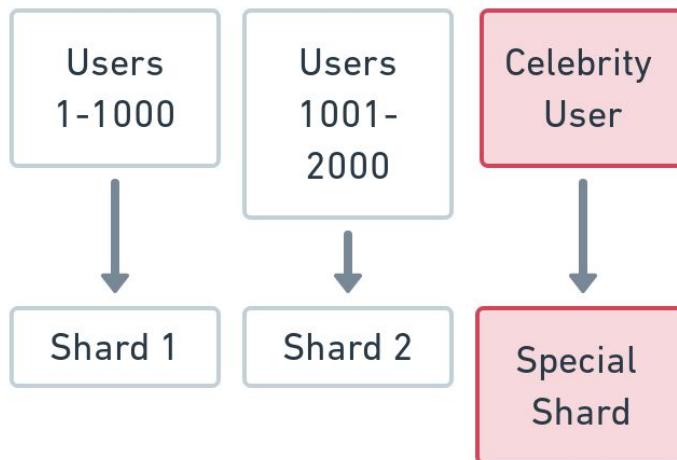
You can also shard a 2-dimensional array. Geo-location-based questions use grids frequently. You need to think about hotspots in a given cell shard and the need to query adjacent cells. Similar to graphs, if adjacent cells don't have good locality and you frequently query adjacent cells together, your read performance may suffer.

Outlier Keys

Sometimes you may have a few keys that are outliers due to celebrity, big enterprise companies, or power users. The unfortunate shards that handle those outlier keys will be hot regardless of your sharding scheme.

Option 1: Dedicated Shard

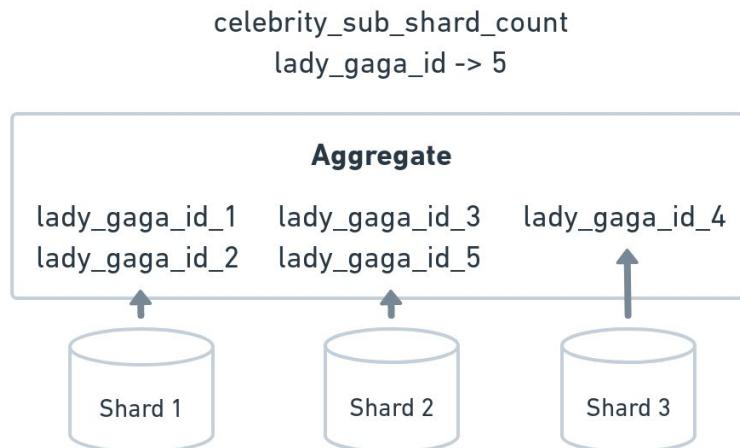
A possibility would be to take the outlier keys out of the generic sharding problem space and have a dedicated shard. Dedicated shards are common in enterprise applications where you have a few big companies with outlier query patterns. The advantage is that you can prevent outliers from the normal problem space, but the disadvantage is the complexity of maintaining these one-off shards with configurations.



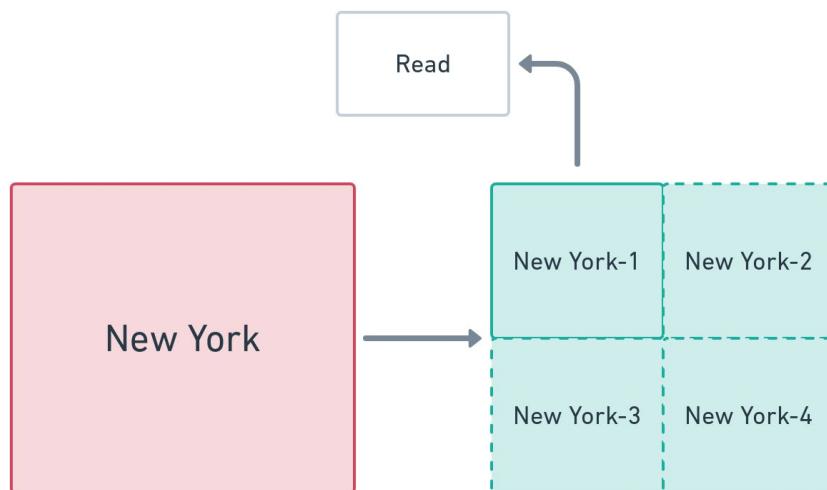
Option 2: Shard Further

Another possibility is to divide up the shard further. However, sometimes even if you shard further, you may still need to scatter-gather all sub-shards.

For example, if you further shard a celebrity, you need to maintain the keys for scatter-gather read.



Sometimes it's not a problem when you just need the data from a sub-shard. For example, to look for drivers in the city of New York, you can just search within one of the sub-shards because you only need one driver.



Shard Key and Primary / Index Key

When sharding, the sharding key can be separate from the primary and index key. Sharding keys mainly helps you determine what to use to break the data down. Once you reach a particular shard, you can still have your primary and index key to optimize for the read and writes.

For example, say you have a tweet table with columns `tweet_id`, `user_id`, `tweet_message`, and `timestamp`. You may decide to shard `user_id` by hash. Then, for a given shard, you might index the database by `user_id` and `timestamp` so you can fetch a list of tweets for a given timestamp for a given user efficiently by first going to the user shard and then using the index to efficiently query for the list by timestamp.

The diagram illustrates two shards of a tweet table. Each shard contains a timestamp index at the top and a data table below it. Arrows point from the shard labels to their respective timestamp indices.

<code>user_id</code>	<code>timestamp</code>
1	10:00 AM
1	10:01 AM
1	10:02 AM
2	10:01 AM
2	10:02 AM
2	10:03 AM

<code>user_id</code>	<code>timestamp</code>
3	10:00 AM
3	10:01 AM
3	10:02 AM
4	10:01 AM
4	10:02 AM
5	10:01 AM

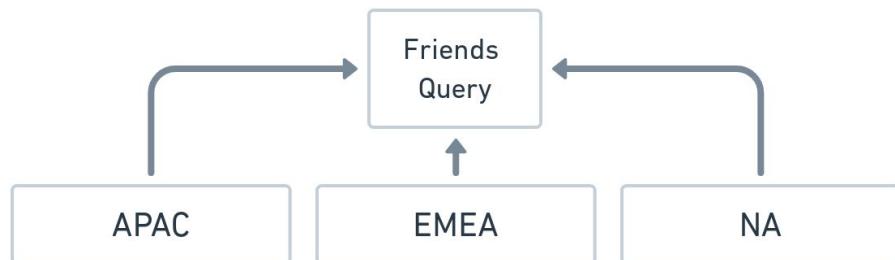
Geo-Shards

Also, it's possible to shard into multiple layers, and the common use case is geo-sharding, also known as zone. You can create geo-shards, so user requests closer to the geo-shard will go there. Then, within each shard region, you can shard it further. If each user is only interested in querying for one geo region, the design becomes simple, since their requests always go to the same shard.

APAC		EMEA	
user_id	timestamp	user_id	timestamp
1	10:00 AM	3	10:00 AM
2	10:01 AM	4	10:01 AM

user_id	timestamp	user_id	timestamp
5	10:00 AM	7	10:00 AM
6	10:01 AM	8	10:01 AM

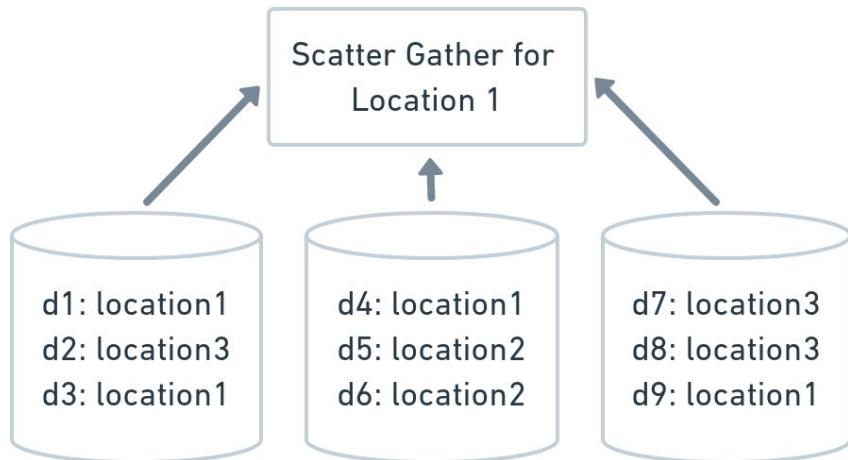
Sometimes in social graphs, you may be interested in your friends' data, and if the sharding scheme scatters your friends across different geo-shards, then you need to scatter-gather. To optimize for this, use the heuristic that friends tend to be closer to each other by location, and you should group friends closer if possible to minimize the scatter-gather.



Sharding Considerations

Here's a list of things to consider when you design and are thinking about the trade-offs. *There is no shortcut here*. You need to think critically about the options and the trade-offs. But we will provide some frameworks and examples to help guide your problem-solving below.

Scatter/Gather



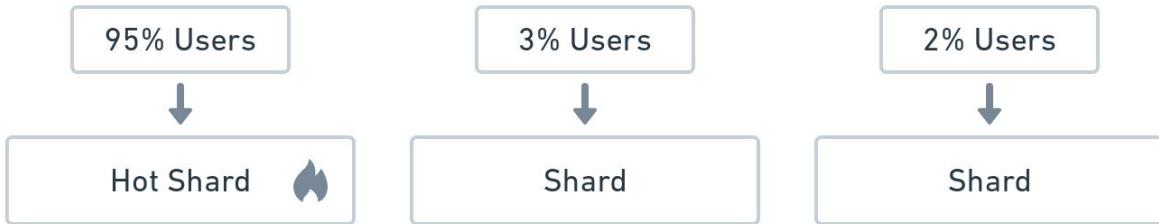
When you shard, you need to think about retrieving the data you need based on your sharding scheme. For example, assume you have a driver location table with `driver_id` and `location_id`. If you shard it by `driver_id`, each shard will likely have drivers with that `location_id`. When you fetch drivers for a `location_id`, you need to scatter-gather all the shards. Scatter-gather is not as performant as if you were to fetch it from a single shard.

Hotspots

Whenever you shard, you need to think about distributing your data across the shards and think about if there are real-life scenarios that could lead to hotspots. In theory, you can always come up with hypothetical examples where a shard gets a hotspot. For example, say you have a user table with `user_id` and `name`, which only the users can write and read from. You decide you want to shard the table with `user_id` through consistent hashing. Consistent hashing should distribute the data well across nodes. However, what if there's a superuser who loves to update their name? It doesn't matter how you shard, if the user is doing a high QPS to that shard, it will lead to a hotspot.

It is your job to develop reasonable assumptions about real-life query patterns to make the final sharding recommendation in an interview. The interviewer might change the query assumptions to see how you adapt to the new assumptions.

Here are some more realistic hotspot examples: If you're designing a chat application and each message has a timestamp, if you shard it by time range (10:00 AM to 11:00 AM go to the same shard), your shard that's taking in the current time will be pretty hot. If you're storing the driver locations and you're sharding it by location ID, a location ID might be associated with a populated city and lead to a hotspot. If you are designing for a social media app database that stores feeds, celebrity posts will be frequently read and become hot.



When you're figuring out hotspots, you need to consider both read and write queries. That should come from your APIs defined in the system design framework, and you can think about how those read and write APIs impact your design. Data storage patterns are usually correlated with reading and writing queries, but not always. It's entirely possible that due to some regulation in Europe, you need to store more data than in North America, even if you assume the QPS to be the same. So look at your queries, schema, and data distribution to make an informed decision.

Machine Hops

Machine hops happen when you need to read from shard to shard for subsequent queries. For example, if you're designing social graph storage, you have to query a friend of friends. If you shard it by name, all your friends may live in different shards and need a big scatter-gather. And to fetch for friends of friends, you need to read from other shards again if they don't have a good locality. However, you can also shard it by their primary location because friends usually live close to each other. Then you may only need to gather from fewer machines because they might already be on the same shard. Of course, this could lead to hotspots for particular areas, but that's a trade-off you need to mention and discuss with the interviewer.

Making the Final Recommendation

One thing you need to realize is that there isn't a perfect solution for sharding. As interviewers, we can always develop a hypothetical scenario where a hotspot can occur even if you use techniques like consistent hashing. It is more important to talk about options and the trade-offs in an interview setting and make a final recommendation by making your assumption. Make an assumption that sets yourself up for success and does not overcomplicate things. The interviewer may follow up by changing the assumption, and then you need to think critically about the updated pattern.

and step back to reconsider the options and add additional techniques to mitigate the hotspot.

We will go through an example below because we often see candidates struggling with the sharding design:

Imagine you have a driver table with columns `driver_id` and `location_id`, and the queries are:

1. `update_location(driver_id, location_id)`
2. `get_drivers(location_id)`

Option 1: Consistent Hashing by Driver _id

Hash the `driver_id` and assign the `driver_id` to a node on the consistent hashing ring.

The advantage of doing this is we have an even distribution of drivers across the nodes, and it is unlikely a driver will lead to a hotspot since all driver patterns should be similar. The disadvantage of doing this is that we need to go through all the shards when we query for a list of drivers for a given location.

Option 2: Consistent Hashing by Location _id

Hash the `driver_id` and assign the location to a node on the consistent hashing ring.

The advantage of doing this is when you query for a list of drivers for a `location_id`, you just need to go into a single shard. However, this could potentially lead to a hotspot for a given `location_id` in populated cities. Also, if a given `location_id` does not have enough drivers, you might need to look at adjacent `location_ids` and have machine hops to adjacent `location_ids` since they may live in a different node on the consistent hashing ring.

Option 3: Custom Sharding by Location _id

shard 1	shard 1	shard 2
shard 1	shard 1	shard 2
shard 3	shard 3	shard 4
shard 3	shard 3	shard 5

Imagine each grid to be a location_id. You would group adjacent location_ids together into a shard. The advantage of doing this is similar to option 2 where get_drivers(location_id) is more efficient because all drivers are in one shard. The additional benefit is that querying adjacent tiles has a better locality. However, it has a worse issue with hotspots where s1 tiles can be associated with the populated city.

Final Recommendation

“I am going to assume the query get_drivers(location_id) is called often, and it is very expensive to hit every node on every read. I would go with option 2. Even though with option 3 it is nice to have the locality for adjacent tiles, cross shard call is not a common use case. If hotspot becomes an issue for option 2, we can further shard the tiles.”

Cache Strategies

Definition

Cache is a storage that improves query efficiency. Cache is different from a database in that cache is volatile, which means that the data will be lost when the cache goes down.

Purpose and Examples

Before we dig into specific caching strategies and their trade-offs, it is worth going through the benefits of using cache. It is important to understand the benefits because almost all candidates introduce cache without really justifying the whys, as if a cache is a magical layer that will solve all scalability problems. In reality, cache comes with a high cost of maintainability and complexity. Before proposing cache as a solution, it's important to understand the problem you're trying to solve.

Improve Latency

According to the latency numbers in Appendix 3: Latency Numbers, reading 1 MB of disk compared to memory is almost 100 times slower. However, just because cache is 100x times faster doesn't mean it is strictly better, because cache comes with complexity. In a system design interview, this is a great spot to think about the latency non-functional requirement. For example, if the end-users of your application are fine with 500ms latency, it doesn't matter if you reduce latency from 5 ms to 0.1 ms. However, if the end-user is hoping for less than 20 ms latency, reducing the latency has a much bigger impact.

You can use cache to improve more than just a database. For example, you can use cache to materialize various data sources into fields where the client can quickly access, especially if computing the data is compute-intensive. The challenge of this pattern is to figure out how to keep the materialized field up to date as the data sources continue to mutate. But a similar thought process applies. You should only cache for latency if the gain is adding value to the end-user per non-functional requirements.

In a system design interview, don't just claim you want to improve latency without knowing why improving the latency is important. Like in real-life design, you're not going to add a cache layer on every stack if it doesn't add much value to the end-user.

Another way cache can improve latency is similar to replication where the architecture can bring the data physically closer to the user. One canonical example of this is using the CDN, which is considered a form of a cache.

Improve Throughput

For a thought process, assume disk and memory server have similar specs. If both the disk and memory have a single thread with a single core and if memory is 100 times faster than disk, then in theory, memory can process 100 times more work in the same amount of time. Perhaps in an interview, you calculate a QPS, and you see that a single database can't handle that QPS. You might consider caching as a possible solution to the problem.

Improve Bandwidth

Similar to replication, you're able to bring the data source physically closer to the user. By bringing the content closer to the user, you reduce the number of bytes that need to go through the internet and improve the overall bandwidth capacity. This is one of the main problems CDNs try to solve.

Cache Considerations

Cache Hit Rate

Cache Hit: The data the request is looking for exists in the cache.

Cache Miss: The data the request is looking for doesn't exist in the cache.

Cache Hit Rate = Cache Hit / (Cache Hit + Cache Miss).

There is a non-trivial cost to having a cache that fronts other data sources. We need a database because of its durability, but we don't always need a cache. If we have a cache, the cache needs to be useful because it costs money. After you've identified the reasons to use a cache for your particular system design question, you need to think about utilizing the cache. If your cache hit rate is low, it might not be worth it. Depending on what you cache

and the expected query pattern of the user, it may be that a user only reads it once and never makes that same query ever again. In that case, there's no point in caching. The trade-off is why it is important to understand and make assumptions about the query patterns in an interview so the interviewer knows how you're thinking about it.

What Are You Caching?

A lot of candidates see cache as a magical layer that solves all the problems. Similar to database schema design, it is really important to articulate what you're going to cache.

Situation
The candidate is contemplating on using a cache.
Don't Do This



Candidate:

"I am going to put a cache in front of the database and it is going to scale the database because it can handle more traffic and will be faster."

Do This

Candidate:

"From the non-functional requirement, we are trying to achieve sub 20 ms latency with more than 100,000 read throughput. Let me consider a cache since it will help achieve the latency and throughput we're looking for, and I will discuss the complexities that come with it. The key is the user_id and the value is the recommendation list, and here are a couple invalidation strategies."

Analysis

In the Don't section, the candidate trivialized the complexity with a cache

without demonstrating why we need the cache in the first place, what we're caching exactly, and the trade-off. In the Do section, the candidate has a purpose to use the cache and explains the complexities that come with it.

Let's go over an example to demonstrate that details matter. Let's say we're designing for a simple search service that takes in a free-form text with English words.

`search(free_text) → [doc_id]`

The search result should contain a list of documents with those words. For simplicity, we will assume there are two documents:

Document 1: “System Design Interview”

Document 2: “Coding Interview”

Option 1: Cache the Search Query

“My OR Design” → [1]

“Super OR Interview” → [1, 2]

Option 2: Cache the Text Token

“Design” → [1]

“Interview” → [1, 2]

Option 1 is a query key-value look-up which is faster compared to option 2 where you need to break the text into tokens and search each token individually and combine the result. However, option 1 will have a much worse hit rate because end-users are less likely to search for the same free form text. The above is an example where details matter a lot. You need to talk about what you're caching and what the implications are.

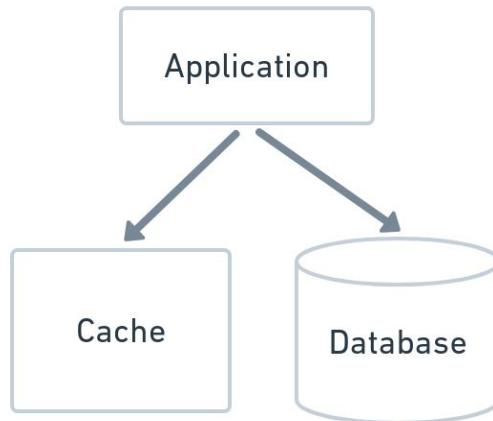
If you have an in-memory map, talk about what is the key and the value. If you have a queue, talk about how the queue represents an event. If you have a tree, talk about what is in the node. Discuss read and write access patterns

and how they relate to the end-user experience for each data structure. Then you can articulate how you will use the database schema to build the cache and talk about the cache hit rate with the assumed query patterns.

Write to Cache

This section will talk about various caching strategies to populate the cache when writes come in.

Write-Through



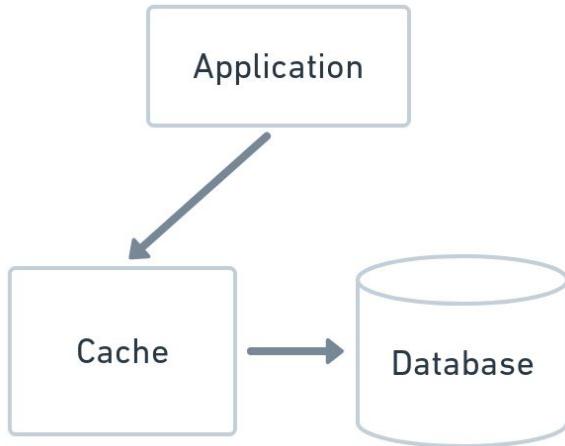
For write-through cache, the data is written synchronously to both the cache and the underlying data store.

The advantage of this solution is the data will, in theory, exist in both the cache and the data store.

The disadvantage of it is higher write latency and lower availability since synchronous write to 2 sources has a higher failure rate than just 1. Also, since the cache and database are two separate sources, doing a synchronous write to two separate sources means the atomicity of the write isn't guaranteed. This will be discussed more in the distributed transaction section.

Write-Back

In a write-back cache, we first write to the cache and update the database later.



The advantage of this write-back cache is the latency is lower because you only need to write to cache without writing to the database. The data would be immediately available to be served by the reads.

The disadvantage of write-back cache is that the data on the cache could be lost before the system persists data on the database.

In a system design interview, this could be an option if the write latency needs to be performant and occasional data loss due to cache failure is acceptable.

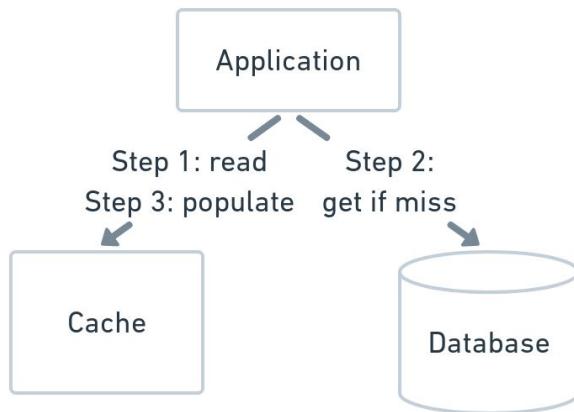
Write-Around



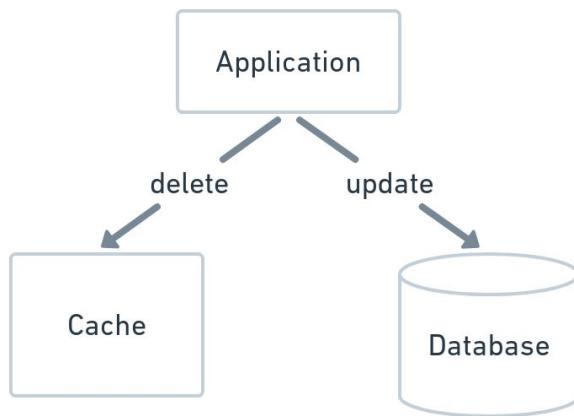
In a write-around cache, the system only writes to the database without writing to the cache.

The advantage is the data is durable by going to disk first. In addition, you don't have to worry about writing to cache and having the cache fail on you before it goes to disk.

The disadvantage is the cache isn't populated, so if the client accesses the cache for the first time, the latency will be slower. Also, there's additional complexity to update and warm up the cache.



With a write-around cache, you can incorporate a read-through cache. Upon read, if there's a cache miss, you fetch the database and populate the cache.



Upon updating the key from the database, the system invokes a delete operation onto the cache. Update to the cache is also possible but is not idempotent. For example, if there are two updates (A and B) to the database, it doesn't guarantee A will hit before B in the cache in a

distributed system. It will be difficult to guarantee updates ordering if requests happen concurrently.

Cache Invalidation

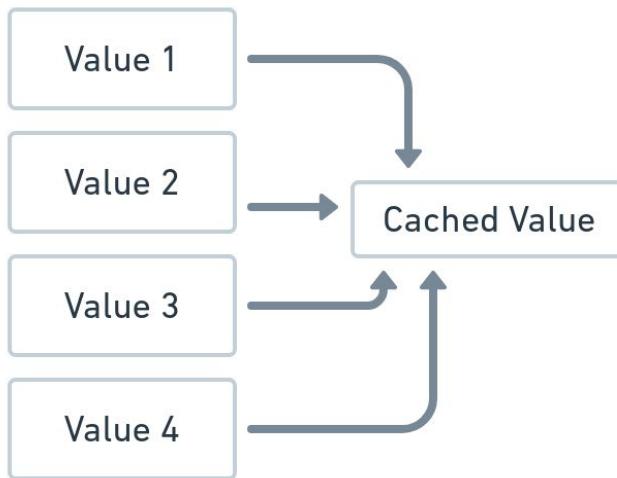
There are only two hard things in Computer Science: cache invalidation and naming things.
- Phil Karlton

Designing for a cache is very hard, which is why you need a strong justification. Once you have a cache in place, you need to think about updating the cache when the underlying source of truth changes. Let's take an example where you frequently update a materialized value from value_1, value_2, value_3, and value_4. Let's make the following assumptions about the values:

Value 1: The value is computed daily via a batch job.

Value 2, 3: The values are persisted by the end-user.

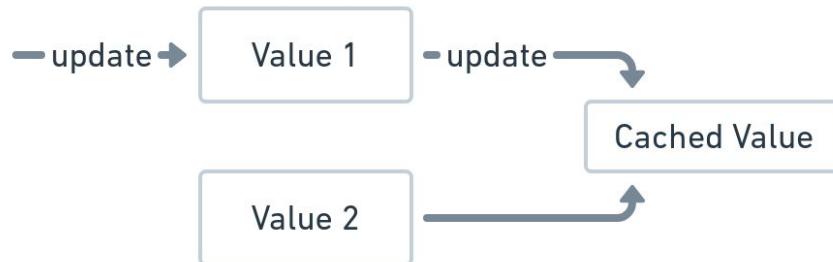
Value 4: The value is derived from another set of values.



Let's say the cached value is the sum of values 1 to 4. If any of the values changes, the sum also needs to change. You need to update the cache value to compute the materialized field again.

Option 1: Have a Listener on the Values

You can have a listener on any of the values, and when any of the values change, you can immediately delete the cache value. The listener is an abstract term since it depends on the data source. It could be a pub-sub queue for post-commit changes.



The advantage of updating the key is the cache will be fresh since it is immediately updated.

The disadvantage is that it depends on the flow of data. The architecture could lead to a lot of unnecessary invalidations if the list of dependencies is huge. Sometimes it's expensive to build a listener pipeline just to update a value.

Also, you can just listen to a subset of the values if the staleness is acceptable for your application. In an interview setting, try to think about the cache value means to the end-user then apply the appropriate strategy.

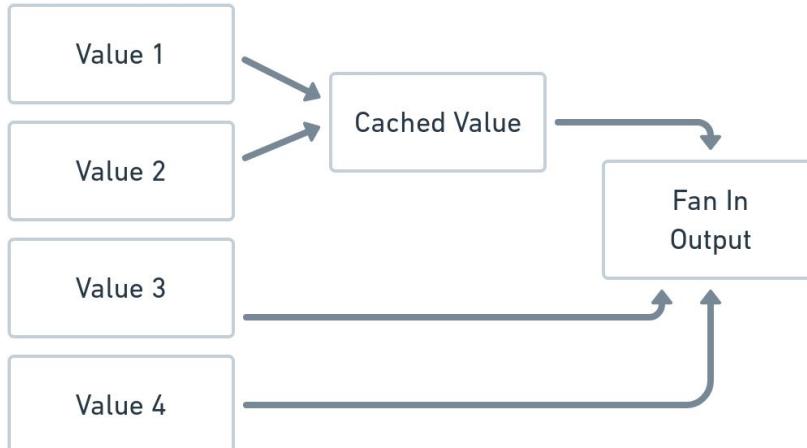
Option 2: Have a Periodic Job to Calculate the Cache Value

Instead of listening to value changes, you can have a periodic job to compute an updated materialized value. The trade-off is that a more frequent run will result in fresher data.

The advantage of periodically updating the cache is the simplicity of not needing to listen to a list of dependent values. Another advantage is the value will already be populated when the user queries from the cached data.

The disadvantage is the staleness of data depending on how frequently you run the computation. Also, you will waste a lot of resources if the dependent values aren't queried frequently.

Option 3: Cache a Lower Layer and Fan-In Read



Instead of optimizing for a super-fast read with a single value, you can trade off a slightly slower read if it satisfies your non-functional requirement. Take the above example where the system derives cached value from value 1 and value 2. When a client queries the output, you can read from the cached value, value 3, and value 4 and compute on the fly.

In a system design interview, this is a common pattern to think about for questions like the Netflix recommendation system and generating News Feeds where candidates often have some sort of cache to store the temporal News Feed and Netflix recommendation. If any of the signals change, you should think about how that would impact the cached recommendation and News Feed.

Also, as a note, materialized property doesn't necessarily need to be an in-memory cache value. You could also store the materialized property onto disk as well, and it faces the same invalidation challenges.

Option 4: Expiration Through TTL

When you insert a cache value, you can have the option to specify a time-to-live (TTL) where the entry expires after the TTL has expired. The trade-off here is that the shorter the TTL, the more frequent the cache miss, resulting in a lookup. The longer the TTL, the more chance the entry is stale, and if it's not frequently accessed, you will be wasting memory on the cache.

For example, the browser caches the DNS address. When the website updates the IP to the DNS servers, the DNS servers don't send massive invalidation updates to all the browsers. The browser relies on TTL to invalidate the cache.

Cache Eviction

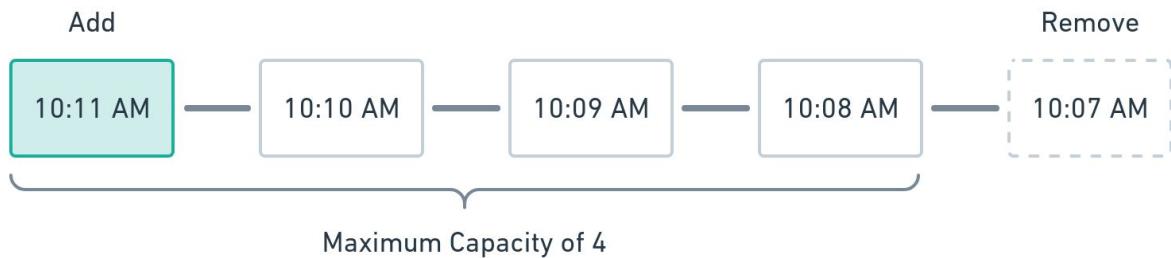
In cache invalidation, we talk about keeping the cache value up to date when the underlying data sources change such that the cache value is no longer up to date. We will discuss how we can't store everything in the cache because cache space is expensive and limited. Cache eviction is a heuristic-based question concerning query patterns and budget and there's no right answer here.

What you're optimizing for is limiting the amount of memory used in your cache cluster while maintaining a reasonable cache hit rate. If you evict a commonly queried key, you might have worse performance on average due to a higher cache miss rate by needing to hit the disk with higher latency. Ask the interviewer and make assumptions about the query patterns, then pick the most appropriate strategy. The purpose is to let the interviewer know about your thought process and not your ability to recite as many strategies as possible.

Here are some common eviction strategies and when to use them. The reasoning should be straightforward.

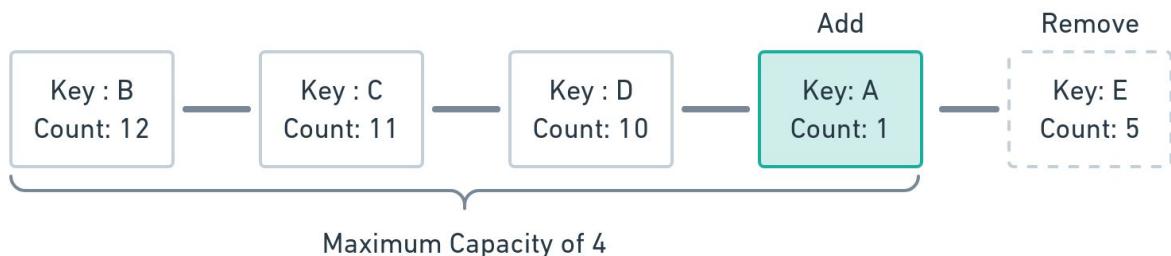
Least Recently Used (LRU)

Whenever you add, update, or access a cache value, the cache considers the values most recently used. Values that haven't been updated and accessed for a while will have a higher chance of eviction. This pattern should be intuitive since data that hasn't been accessed recently isn't likely to be accessed. But it isn't always true since it's possible a key query is cyclical and accessed for a short duration and doesn't come back until much later.



Least Frequently Used (LFU)

For each cache value, count the number of times the key is updated or accessed. The ones that are least accessed are more likely to be evicted. This pattern makes sense since frequently accessed keys should continue to be accessed, but it isn't always the case. If a stock has a lot of volatility and volume and suddenly becomes quiet, that stock's cache hit rate will be lower, even if it was just frequently accessed.



Custom Eviction

There are more eviction strategies, and they all require you to think about the query pattern. For example, you should use a heuristic to predict the cache hit rate for future queries. However, you might already know what the future looks like and pre-optimize for the future in your application. For example, a stadium hosts a concert once a month, and the traffic will be quiet until the day of the concert. Therefore, you may not want to evict that data the day before the concert, and evict it after the concert instead.

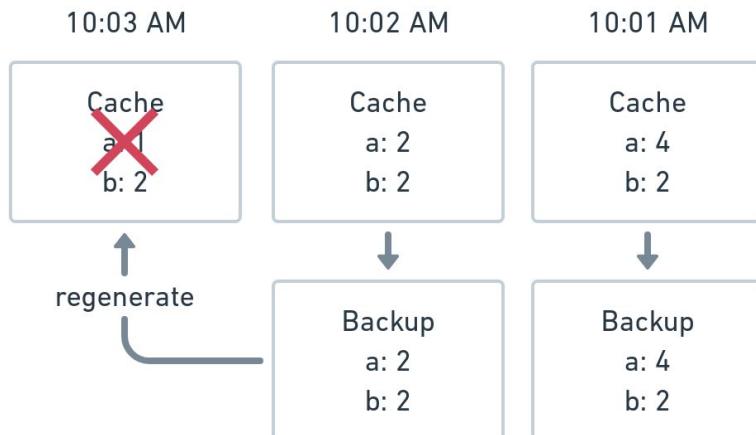
Failure Scenario and Redundancy

The cache may occasionally crash like everything else. However, unlike databases, when a cache server crashes, all data is lost. When the data is lost, and requests hit the cache, the requests may cause a thundering-herd to the database and bring the database down.

There are various options for handling failure scenarios. In a system design interview, you need to think about how each option will impact the requirements of the question you're designing. Failure scenarios are often a good deep dive discussion topic to bring up during the interview. It's important to think about the options and see how it impacts the end-user experience.

Option 1: Periodic Snapshot

The cache can periodically create a backup file of the data in the current cache. When the cache goes down, the cache server can use the backup file to recreate the cache. In the below example, we're creating a snapshot every minute. When the cache at 10:03 AM fails, we can use the backup at 10:02 AM to regenerate the cache.



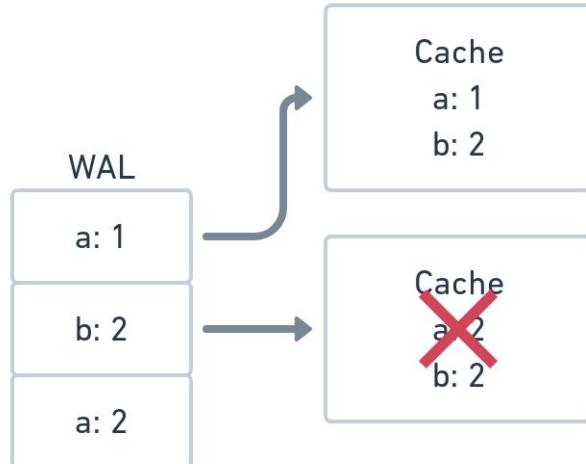
The advantage of creating periodic snapshots is the writes are faster without going to disk. In the event of a cache failure, it still has backup files to recreate a point-in-time snapshot.

Depending on the frequency of the backup, the disadvantage is the data might be outdated. Also, it takes time to uncompress the files and recreate the snapshot in time.

Depending on your requirements, an out-of-date snapshot might be acceptable. Frequency of snapshot makes a good trade-off discussion with the interviewer between frequency against the freshness requirement.

Option 2: Write-Ahead Log (WAL)

Before writing to the cache, you write to a write-ahead log (WAL) to disk. Since WAL is append-only, the operation is fast. In the example below, let's assume the cache fails after 2 WAL blocks. The 3 WAL blocks can regenerate the cache.



The advantage of WAL is the cache will have the most up-to-date records. In the event of a cache failure, we can replay from the last checkpoint using the WAL.

The disadvantage of WAL is the write will be slower because the cache update now requires the overhead of appending to WAL. Depending on the length of the WAL, recreating the cache may take some time to replay the logs. You can periodically create a snapshot so you don't have to replay WAL from the beginning. You can just replay from the latest snapshot.

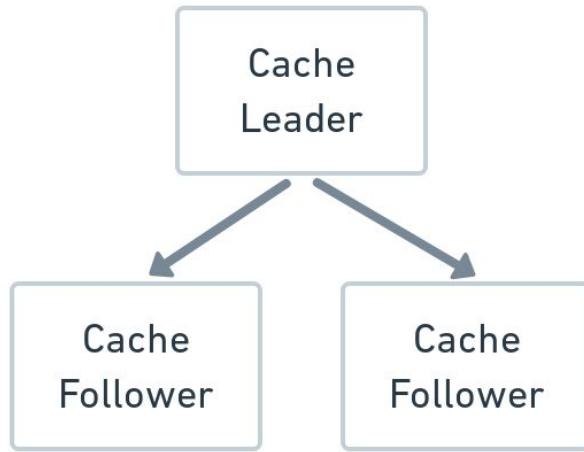
Option 3: No Backup

The data is so transient for some systems that by the time the server restarts and recreates with the backup, the data is outdated. A good example is the driver location update for a ridesharing service, where the drivers are constantly moving. You can just depend on the next update for the latest data instead of recreating from backup.

Option 4: Replication

Cache is a data source like a database. You can have replication for cache, just like a database. The pros and cons of cache replication are similar to database replication. In a system design interview, if a cache server goes

down, you can try to read from another replica to serve the read requests. Also, when the leader is fixed from failure, you can use the read replicas to rebuild the cache leader.

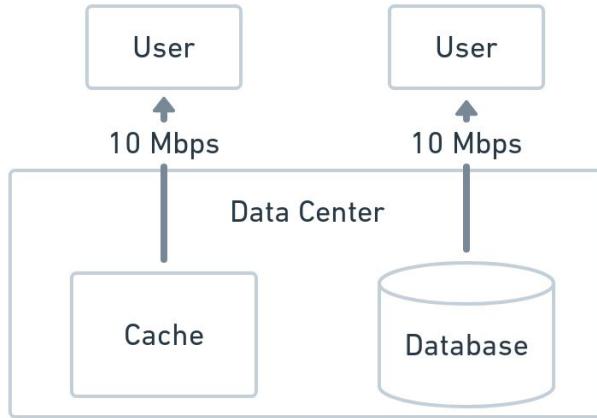


Data Structure

People typically think of a key-value store when people think about cache, but it doesn't have to be. Cache could be some in-memory data structure hosted on a server. One advantage of an in-memory data structure is that it is flexible without fitting into a database structure. For example, a trie used in a type-ahead implementation is a cache. A quadtree used in location-based system design questions is a cache.

As you use those in-memory data structures in an interview, you should also consider the same set of considerations for a cache. For example, if the trie is in-memory, what happens if it goes down? How would you rebuild it?

Warning



A common mistake candidates make for cache is thinking it improves the bandwidth directly. Streaming YouTube or Netflix videos to the user is usually network bound. Having a cache server over the database in the same data center isn't necessarily going to improve the viewing experience if the network is the bottleneck. That's why people move the data source closer to the user, like CDNs.

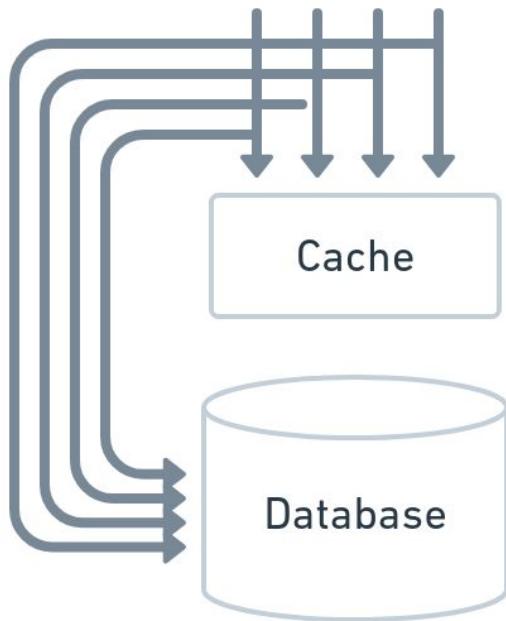
Thundering Herd

The purpose of a cache is to scale the number of clients wanting the same data. The question you might have is what happens when the cache is cold, meaning it doesn't have the data the client is looking for? Typically, engineers solve it with a read-through cache. But what happens if there's a lot of requests hitting the cache simultaneously with cache miss and causing a thundering herd to the underlying system and bringing the system down?

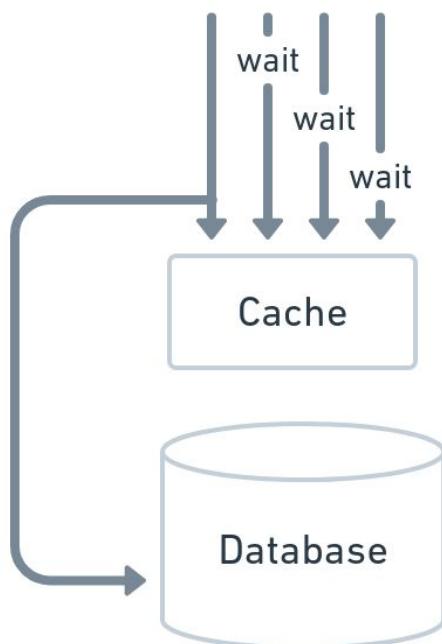
One technique to consider is cache blocking, where only one request is responsible for fetching from the main data source while all the other requests wait. The problem is, what happens when that one request fails to come back to warm up the cache? The rest of the requests will continue waiting.

One way to address this is by having a timeout and allowing the next request to fetch from the data source. A trade-off discussion that you can talk about in the interviewer is the longer the timeout, the more chance the requests have to wait if the response never comes back. The shorter the timeout, the more requests that will hit the data source.

Thundering Herd



Cache Blocking



Asynchronous Processing

Definition

Asynchronous processing is when a client executes something and doesn't wait for that task to complete before moving onto something else.

Purpose

The purpose of asynchronous processing is to offload the task at hand to a background processing such that the caller does not have to wait for the response of the task result. This leads to lower latency because the request processing doesn't block the client.

Synchronous and Asynchronous

Before going deeper into the advantages and disadvantages of asynchronous processing, it will be helpful to go over some examples of the difference between synchronous and asynchronous. Unfortunately, a lot of candidates get confused between the two. For an API call, synchronicity is concerning how much server work completion the client waits for before moving on to the next operation. In a synchronous call, the user waits for the task result, whereas in an asynchronous call, the user does not wait for the task result.



This is a synchronous call from the user to the server since the user waits for the server's response.



This is a synchronous call where the user waits for the response of the job.



This is an asynchronous call where the user doesn't wait for the server's response.



This is a synchronous call where the user waits for the server's response but doesn't wait for the job's response. The job is asynchronous concerning the user. The user experience from this scenario is usually, "We got your request, and we're processing it for you."

Task Scope

The scope of the task is important here. Knowing how to differentiate the two will allow you to describe the user experience in the system design much clearer.

Here's an example. Let's say you are designing a simplistic checkout API for an e-commerce platform. To place an order, you have to:

1. Click the checkout button.
2. The system will charge the order payment.
3. The system will ship out the item.

Option 1: Don't Wait for Anything

If the synchronicity scope of the API is up to step 1, "Click the checkout button," the user and the browser client will not wait for steps 2 to 3. The end-user will likely receive a message saying, "We are processing your order, and you will receive a notification when your payment is charged and the item is shipped out." While the system is processing, if the payment charge and shipment are successful, the user will get a notification.

The advantage is the user doesn't have to wait for the payment processor to finish before placing the order. It has a better perceived latency. However, the disadvantage is a poorer user experience if the payment processor fails and the user has to come back and try to place the order again. The failure doesn't even have to be a system failure, it could be that the user's payment method got declined.

Option 2: Wait for Payment Processor

If the API synchronicity scope is up to step 2, the end-user will receive a message once the payment is successful saying, "We have successfully

charged your payment, and are in the process of shipping your item.” If the payment fails, the user will receive, “We can not charge your payment, please try again.” The advantage of this approach is the quick feedback loop of payment failure, and the user can change their payment method right away.

Final Words

You need to be clear about the scope of your synchronous call because it significantly impacts the user experience, especially if the invoking user is waiting for some result of their call during a system design interview. API scope is a great spot to bring up options and trade-off discussions. Remember, the interview is a trade-off discussion, not just a single solution.

Reasons for Asynchronous Processing

The main idea for asynchronous processing is you don’t want the end-user to wait for your task to complete because waiting for the job to complete would be a bad user experience. So, in an interview, you should justify why asynchronous processing would improve the end-user experience. Here are some examples that might be a good candidate for asynchronous processing:

The Nature of the Job is Asynchronous

In the checkout example above, the end-user is waiting for some kind of response. Sometimes the scope of the system design interview problem does not immediately involve an end-user. For example, in a web crawler design, a list of URLs is already prepared for you to be processed, so no end users are waiting for the crawling process to finish. For a log search problem, you might already have a bunch of logs to process where the clients who generate the logs aren’t part of the scope of the problem.

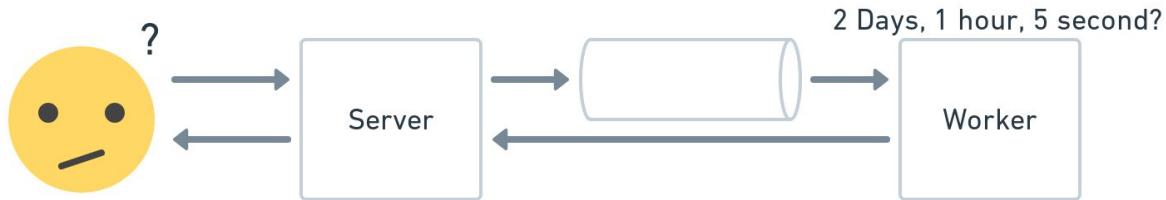
Another example would be applications like chat and email. Once the end-user has successfully sent a chat or email, the end-user shouldn’t have to wait for the recipient to receive the message before letting the sender know.

However, sometimes you will get trending topics for trending tweets questions where an end-user generates the data. In this case, a typical

solution is to use some sort of queue, and the request to insert into the queue is synchronous, and everything post queue insertion is asynchronous. This architecture means that if the request fails to insert into the queue, the user will get a message saying, “Sorry you’re unable to tweet now, please try again.”

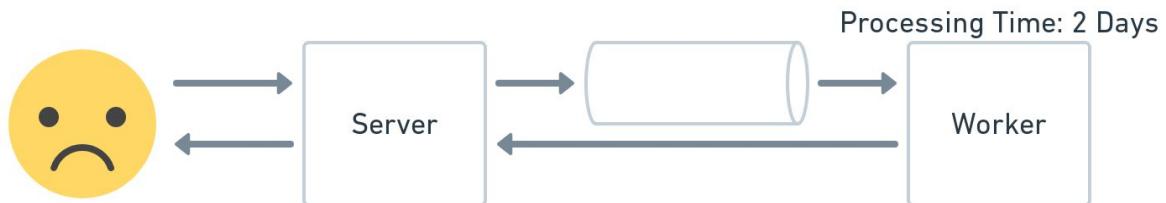
The Processing Time is Indeterministic

It isn’t clear how long the job is going to take for some jobs, and it’s usually better to tell the end-user to wait for a response. For example, when you request a ride to Uber, once the request is in the processing queue, you’re told to wait for your ride because the system isn’t sure how long it will take to match the ride. The user experience is because the processing of matching rider and driver is likely asynchronous.



The Processing Time is Long-Running

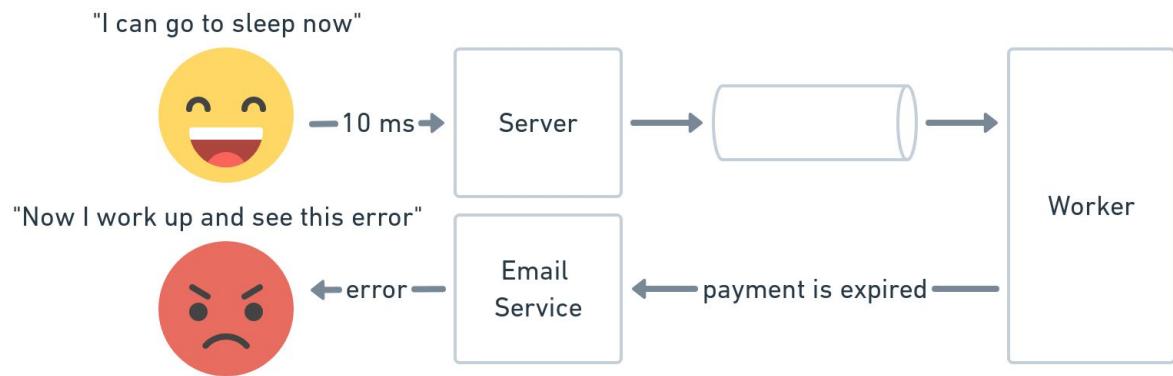
If the processing time is long-running, you may not want the user to wait for a response until the system completes the job. For example, when you place an order on Amazon, it takes two days to receive it. You don’t want the browser to spin for two days until the item arrives at your door. Another example is a system that transcodes movies, which can take a long time. You don’t want the user who uploaded the movie to wait for an hour before the movie finishes transcoding.



Improve Perceived Latency

As discussed in the e-commerce checkout platform, if the user doesn’t have to wait for the payment, the checkout will be faster than if you have to wait for some system to complete. However, it does come at the cost of fixing

some errors if the system is unable to process, whereas if it was synchronous, the user could immediately fix the issue. However, you need to be careful here because this can be an anti-pattern to reduce the latency but incur a much worse user experience downstream.



Batch Processing

Purpose

Batch processing is a form of asynchronous processing. In batch processing, the system processes a large amount of data periodically to generate output to let clients consume later. Here are some use cases for batch processing:

- Run payroll, billing, and accounting for the company
- Generate reverse index for documents
- Generate word count for documents
- Distributed sorting

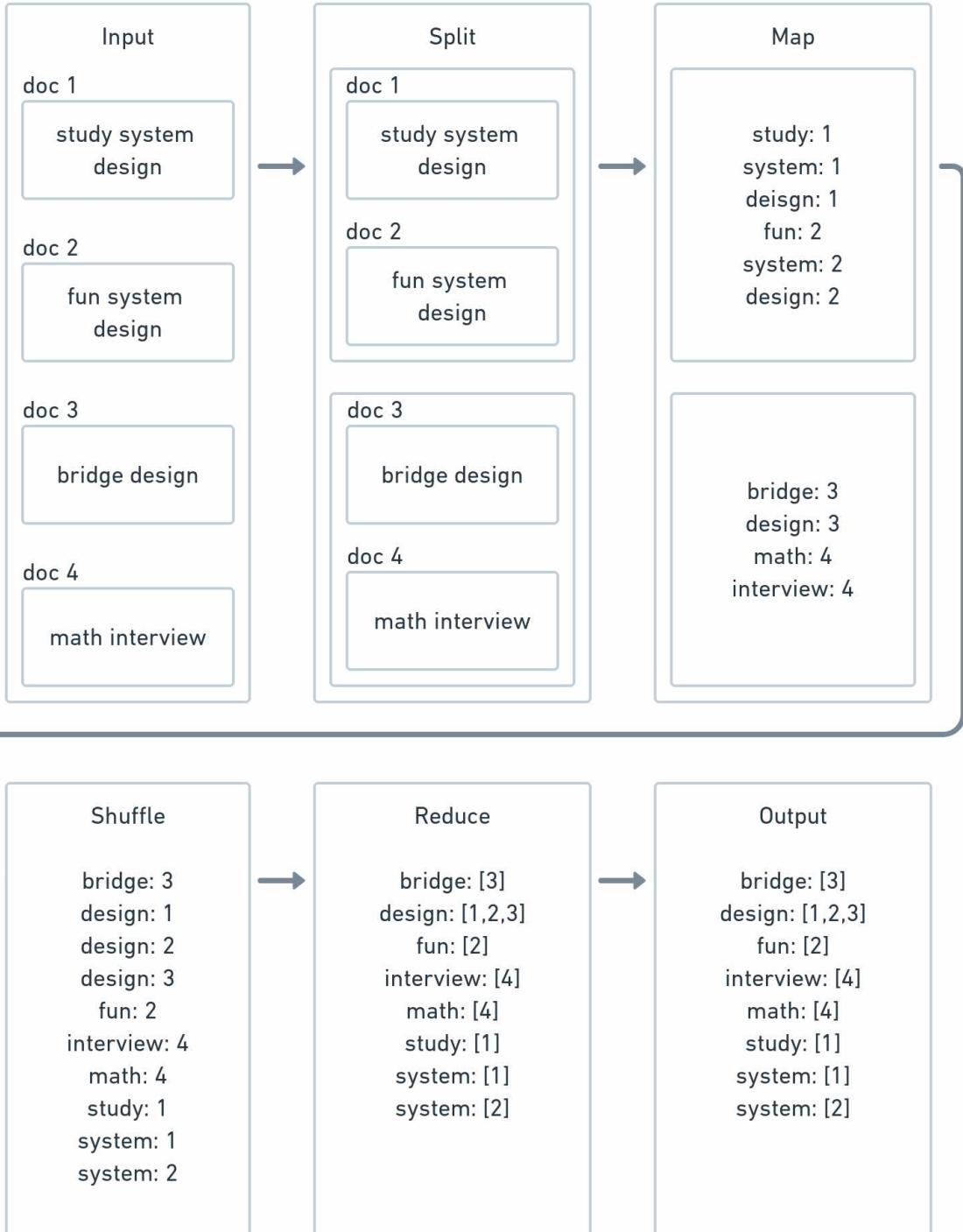
When talking about batch processing, most people will think about MapReduce. It doesn't have to be. Batch processing on a high-level means grabbing some data sources periodically, applying custom business logic, and creating an output to be consumed by another consumer. You can write custom code for each process. However, since many big data applications use MapReduce, it's worth knowing an example of how MapReduce works if the interviewer asks you to provide details about your algorithm in your interview.

MapReduce consists of:

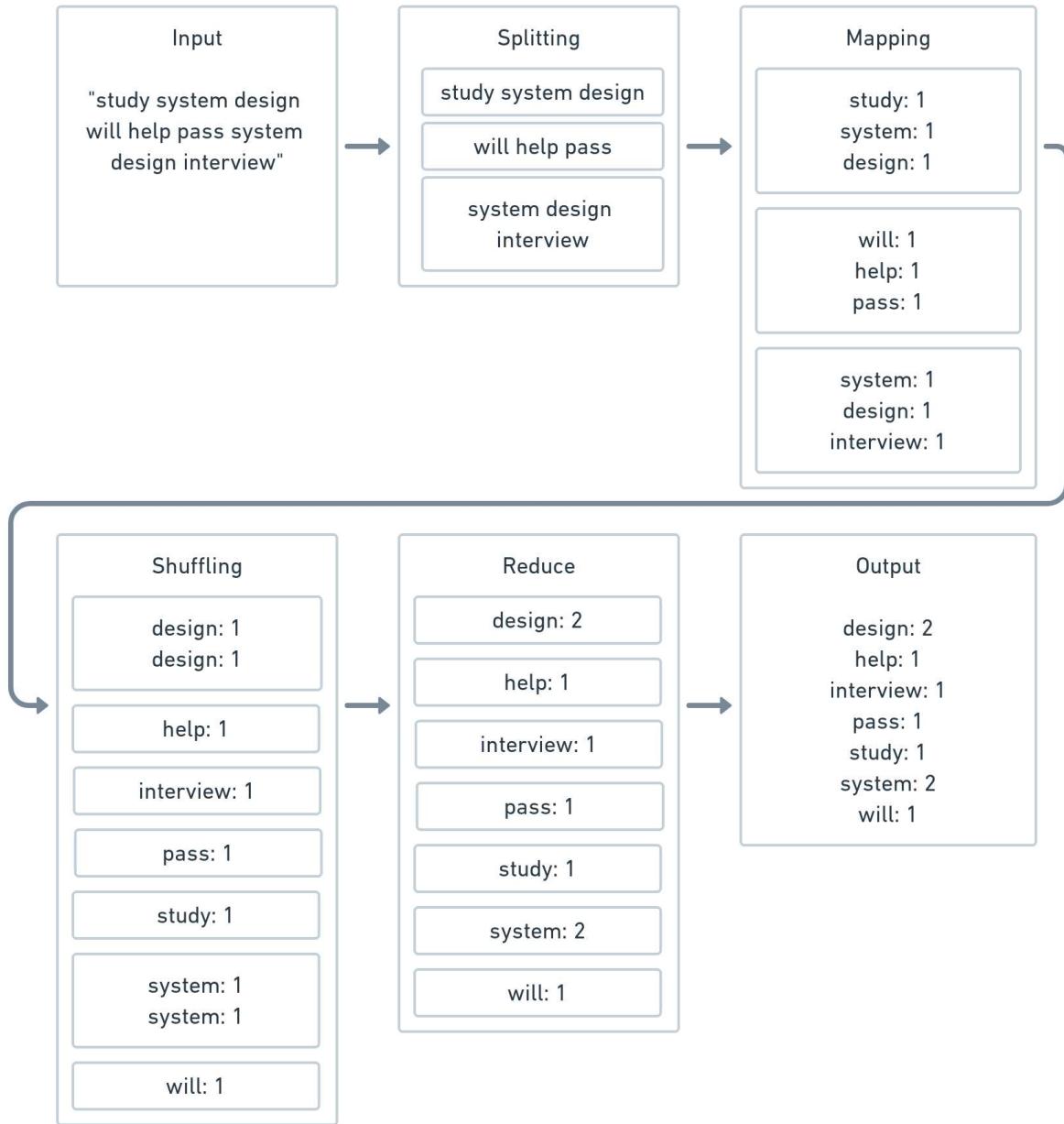


Here are some classic MapReduce examples:

Build Reverse Index for Search



Word Count for Document



Considerations

Here are some distributed system challenges and discussion points you can potentially bring up for your design during deep dives. Again, think about the implications of each challenge and offer some ways to remedy it.

1. What if the batch runs late?
2. What if the batch never runs?
3. What if the batch runs more than once?
4. What if the batch never finishes?

5. Do you have enough resources to run all the jobs? Some jobs, like transcoding, are very resource intensive. How will you prioritize the tasks?
6. For the same logical run, what if the previous run hasn't finished when the current run is supposed to run?

Stream Processing

Purpose

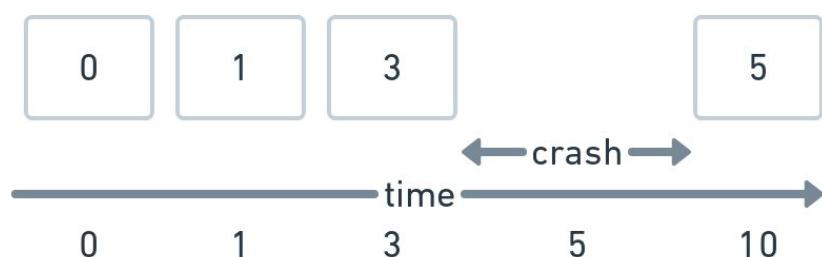
Data is unbounded in a stream processing architecture since data continuously comes in, and the system has to deal with it. The advantage of stream processing over batch is that the output will be much fresher since it's processing near real-time, but it also comes at the expense of complexities.

There are notions of event time and processing time in stream processing. Event time means when the event occurs, and processing time means when the system processes the event. Say you're designing a system to display metrics, and the system omits a metric that occurs at 10:00 AM and the system gets it at 10:01 AM, you need to be clear whether it should be event or processing time. Most likely, you want the event time.

In some streaming applications, event time does not need to be considered. For example, if you have a global counter that keeps track of the number of events since the beginning of time, the event time is irrelevant in this context. However, if you are keeping track of the number of events from one period to another, then event time is crucial because you need to ensure the event time belongs in that period.

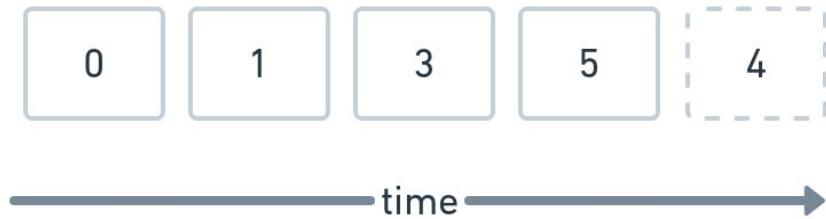
System is Down

Since streaming is near real-time, and batch processing is much less frequent, if the service is down for 10 minutes, it has a much more significant implication for a near real-time system. Also, when the system comes back up, you need to think about how the system will process the 10 minutes of unprocessed data.



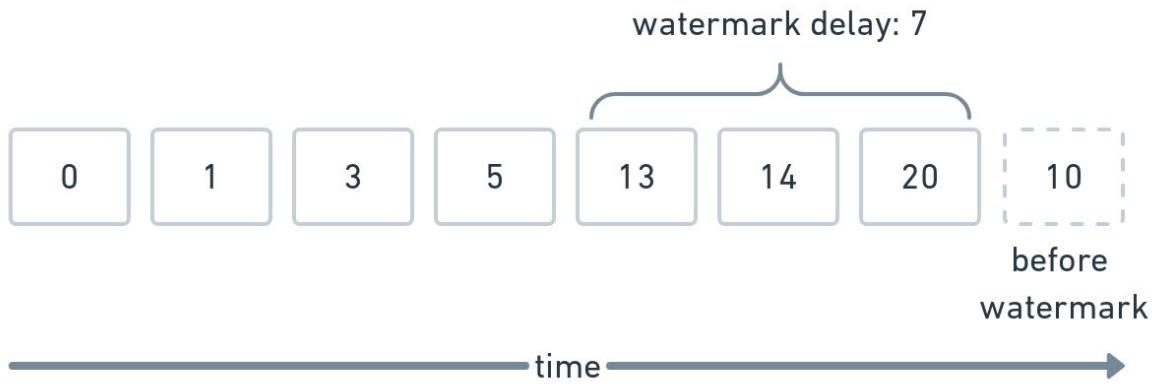
Late and Out of Order Events

In the real-life system, events come in late. Imagine you have a mobile phone that omits events, and the network is down. The mobile phone queues up the events, and when you reconnect to the internet, the events are late with respect to the processing time. On top of late events, since there's a clock skew (each machine has its clock) in a distributed system, you can't assume the events are ordered since it makes it difficult for you to assume you've seen all the events before a certain time.



Watermark

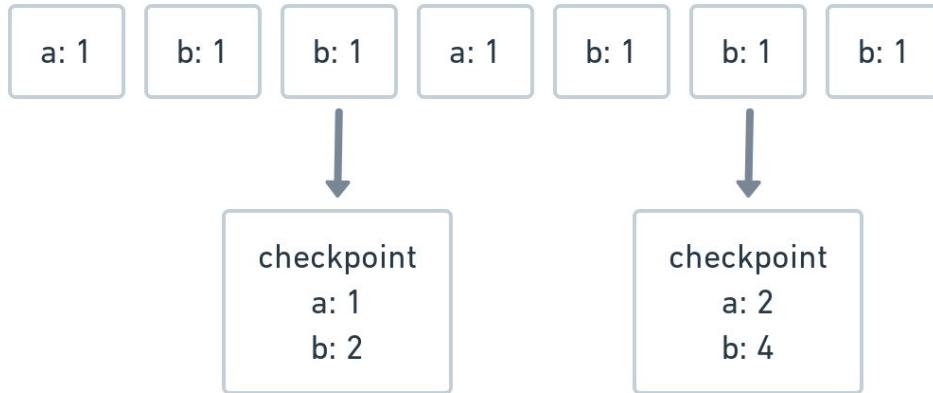
In stream processing, it's worth knowing about watermarks. The intuition behind the watermark is that you have a heuristic behind whether you have received enough data for a time-period to move on. For example, if you are collecting data from time_0 to time_1, due to late events, how do you know you have collected enough information about that interval such that you hope events between time_0 and time_1 won't ever come anymore? Assume you have a watermark delay of 7. If you see a time_20 event, you have a watermark of time_13. If you see events before time_13, you will consider it past the watermark and treat it accordingly, such as dropping the event. A trade-off you can discuss is the bigger the watermark, the more memory you have to hold to account for late data with the benefit of not dropping late events.



Now, what happens if another event belongs in time_0 to time_1? That depends on your application. Some common options are discarding it or updating the previous record. In an interview for a streaming service, this would be an interesting discussion point on the implication of your choice. Discarding is easy but may lead to an inaccuracy that's poor for user experience. On the other hand, updating records isn't as simple as just append-only applications and may require a separate pipeline for updating existing records.

Checkpointing

There is usually some intermediate data structure in streaming applications to keep track of the data processed so far. What if that host goes down? Processing failure is where checkpointing helps so you don't have to reprocess all the events from the beginning. The frequency of the checkpoint and how and what you persist in the checkpoints would be interesting deep dive discussion points in an interview. More frequent checkpointing means lower performance but faster failure recovery, and the reverse is true for less frequent checkpointing.



Batch Size

Even in stream processing, it doesn't mean you process event by event, which may kill the throughput of your system since there's additional overhead per event. For example, imagine for each event you need to do a network or disk IO. To do it on every event will cause IOs to become the bottleneck. Sometimes it's more efficient to micro-batch to make that more efficient. A trade-off discussion is the batch size. A bigger batch size causes delay but may have better throughput since there's less IO overhead per batch.



Lambda Architecture

Batch vs Stream Processing

If stream processing is near real-time and batch processing is periodic and delayed, it begs the question, why don't we always choose stream processing over batch processing? In modern day streaming processing, the line between streaming and batch is a bit murky. It's a spectrum of how much you batch before processing the data. It could be once a second or once a month. In an interview, make sure you understand your particular use case before picking the strategy.

Here are a couple of reasons why you may prefer batch over streaming:

Data is Enormous

Sometimes the amount of data you will be dealing with is so large that the system can't stream them in near real-time. One example would be to calculate a year's aggregation as opposed to a second's aggregation. You can't easily fit a year's data into memory.

Compute Intensive

If the calculation is compute-intensive, it could cause the streaming to backlog the data since the consumption isn't fast enough for the data production rate.

Complexity of Streaming

In batch processing, you're dealing with a bounded dataset. In streaming processing, the data is unbounded. So when events come late and out of order, it's difficult for the system to deal with those complexities in real-time. Also, if you have a streaming service, you will have some service-level objectives (SLO) about the freshness of the data, making it more complex to maintain.

Use Case

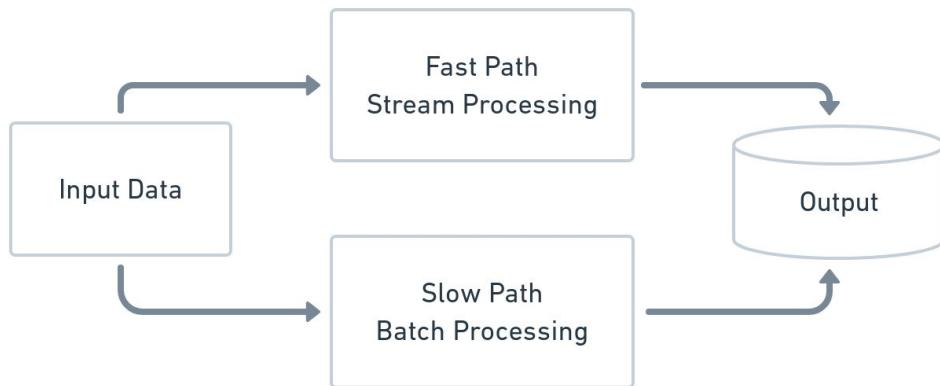
For some use cases, you just don't need the freshness streaming system can provide. For example, if you run payroll for companies once a day, there's

no point in streaming it since the payrolls might not even be ready, and the system should just run when the customers expect payroll to be run.

Lambda Architecture

Sometimes the application can not wait for a delay in processing for the end-user at the same time. Doing the processing accurately and consistently will be difficult with the existing infrastructure. There's lambda architecture with a fast lane and a slow lane. Fast lane tries to minimize latency at the sacrifice of completeness and accuracy like stream processing. Slow lane will have most, if not all the data available to compute a more accurate result for the end-user like batch processing. In practice, there's operational complexity to managing two similar systems.

In a system design interview, you can bring up lambda architecture if the interviewer challenges you about your stream processing and how a certain hypothetical setup (enormous data) could break your system, and you will use batch processing to produce the final accurate result.



Queue

Most likely, in asynchronous processing, you need some sort of queue to ensure the events are waiting in line to be processed. Having a queue is important because the processing speed can sometimes be slower than the producing speed. For example, if there's a thundering herd event, the event may cause the downstream to break or cause the events to be dropped.

Keep in mind that just because you have a queue doesn't make the flow asynchronous. The caller can wait for the result of the queue and make it synchronous. Depending on the queue type, waiting for the response of queue post-processing is unnecessary and slow.

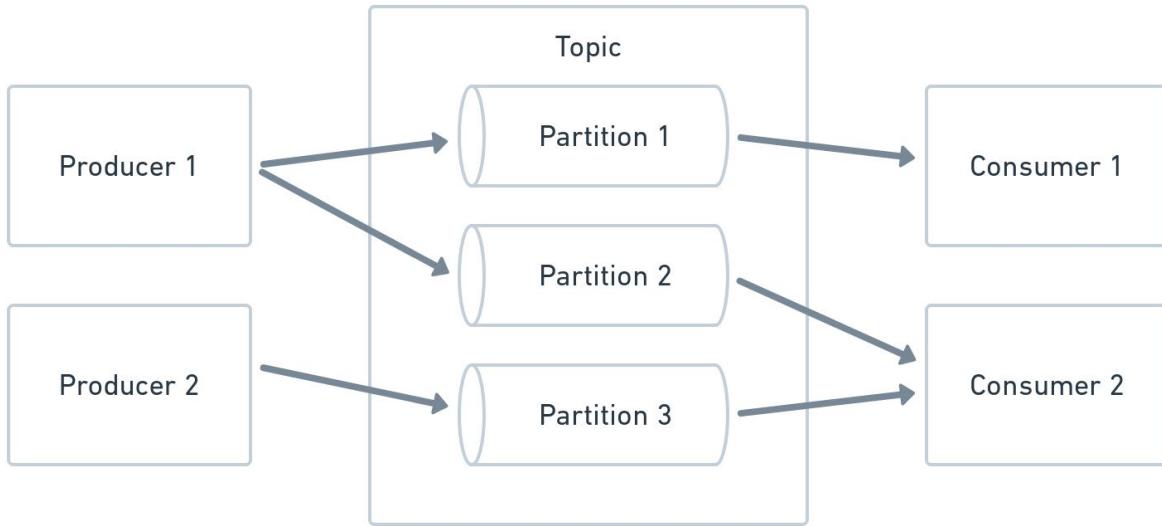
Here we will focus on when we need an asynchronous workflow, and how we can use a queue to help with reliability and durability. In an interview, going deeper about the queue of your choice and the trade-offs might be an interesting deep dive area.

Message Queue

In a message queue like Kafka, clients insert events as logs for a given topic. A topic is a category that you define. You can partition each Kafka topic into multiple partitions, and each partition maintains its message ordering. If you need global ordering, having multiple partitions will remove that global ordering guarantee. For each topic partition, there is a consumer. Each consumer maintains a durable offset as message batches are processed and can resume from the offset in case of failure.

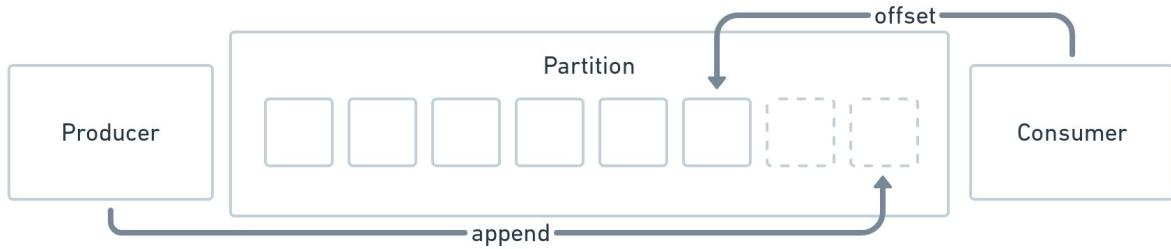
Also, Kafka event logs have retention periods for durability if some consumers still need them within that period. The advantage of a message queue is that it can handle high throughput by horizontally scaling and adding partitions. However, the downside is the complexity of sharding and durability. In practice, maintaining a durable message queue has a higher maintenance cost. It's unnecessarily complex for events like a ridesharing request where you only need to process it once. By the time you reprocess the event, the event is already outdated.

In a system design interview, if you have a system where it takes in a very high write rate of events like metrics and logs, the message queue might be a good candidate to hold the data while the downstream does aggregation. Generally, consumers are partition aware to allow the application to design for throughput for a message queue. This adds additional complexity for the application developer to think about which partitions to connect to.



When you do consider a message queue like Kafka, it's worth considering the following discussion points if it's relevant to your design interview question:

1. How do you scale horizontally? What is your sharding scheme and does it lead to hotspots? How do you query the result after the consumers finish processing? What's the replication scheme for the partitions? How does rebalancing work when you add or remove partitions? You can apply the same knowledge in the sharding and replications chapters.
2. Do you need global ordering? Can you sacrifice throughput by having one partition to guarantee global ordering?
3. What is the retention period for the partition? The longer you store, the more storage you will need, but the safer it is in case a consumer needs to replay from it. The default for Kafka is 7 days but you can use it as a trade-off discussion.

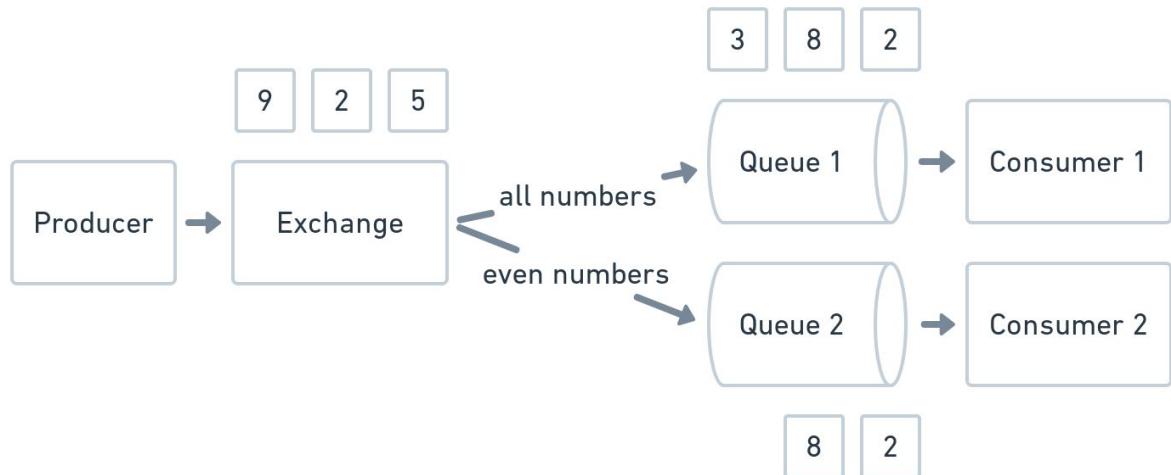


Each consumer maintains its offset per topic partition. In case of consumer failure in the process, the consumer can replay from the last offset.

Examples: Kafka, Amazon Kinesis

Publisher Subscriber

In a pub/sub model, the subscriber subscribes to events based on configuration. Once the producer publishes an event to a message exchange, the exchange will forward the event into each subscriber's queue based on their subscription configuration. Once the consumer successfully processes the event, the consumer will send an acknowledgment to the queue, and the queue will remove the event.



This model has the advantage that each subscriber can behave independently from other subscribers. Unlike a message queue like Kafka, there's no retention period for the message. Once the subscriber pulls the event out, the event is removed. Not having a retention period eliminates the complexity and maintenance of durability of the message queue. Once the queue removes the message, it's up to the subscriber to persist the

processed data. How you deal with the durability post queue processing is no longer the queue's responsibility.

Pub/sub is useful where you intend the event to be consumed and removed immediately. For example, if you're designing a ridesharing service that takes in a message request, that request should be processed as soon as possible and be done with. There's no need to replay it since 10 minutes later, that request is no longer relevant.

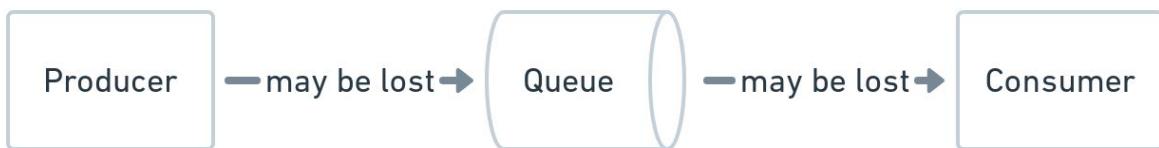
Examples: RabbitMQ, Amazon SQS

Delivery Semantics and Guarantees

Delivery semantics and guarantees are important topics to discuss in a system design interview because your selection can impact the performance and the user experience. For example, imagine you're designing a queue to process payments. Delivering more than once will cause an over-charge, and failing to deliver will result in undercharging.

At-Most-Once

As the name implies, the queue delivers the message at-most-once. You will choose this semantic when the system would rather lose a message than deliver the message twice.



Producer

This producer can achieve this by firing and forgetting into the queue's acknowledgment. If the message never made it to the consumer or the consumer fails to consume the event, the message is lost forever.

Consumer

On the consumer side, the consumer can poll the event and remove the message from the queue. In the event of consumer failure, the system loses the message forever.

Trade Off

At-most-once semantics results in better throughput because it has a lower overhead by firing and forgetting the events. The downside is you will occasionally lose events.

Use Cases

Metrics collection like server health status is usually a good use case where occasionally losing metrics data isn't the end of the world, since there is already a large volume. Driver location update in a ridesharing application is another possible use case. Just because you drop a driver's location at a given point in time doesn't mean the whole system will stop working. Instead, you will get another update in 5-10 seconds.

At-Least-Once

In at-least-once semantics, the queue may contain already processed events. Having already processed events can happen when the consumer fails to acknowledge the queue and remove the event. On subsequent retries, the consumer receives the same message again.



Producer

When the producer sends an event to the queue, if the producer doesn't acknowledge and resend the data, it could duplicate data in the queue.

Consumer

When the consumer polls from the queue and commits the change but fails to acknowledge the queue, the event will be processed again when the consumer retries.

Trade Off

At-least-once semantics guarantees the event won't be lost but may process the event multiple times. The throughput will be slightly worse than at-most-once because it requires an acknowledgment for both the producer and consumer.

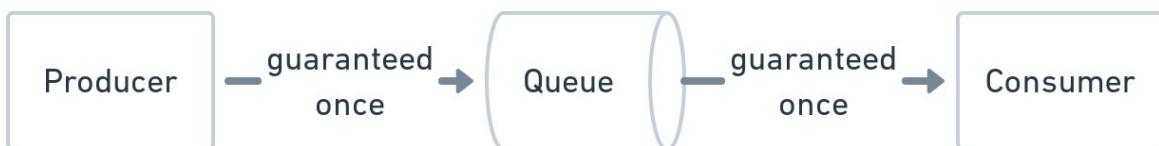
Use Cases

A periodic job that parses a file and saves it to the database. Even if the job runs twice, the same file will be parsed and overwrites the record by file_id. The reason is that this parsing and writing is idempotent based on file_id.

For notifications, sometimes it may be ok to have the occasional minor end-user annoyance of the same notification in favor of better throughput than exactly-once. In this use case, you don't even need to worry about idempotency if your product is ok with the occasional duplicate.

Exactly-Once

Exactly-once semantics ensure the message is processed exactly-once. It does so by having an idempotent key inside of the queue where if you try to insert multiple events into the queue, the queue will dedupe the messages based on the idempotent key. As a result, the throughput is much worse with the need to dedupe the idempotent key in favor of exactly-once guarantee.



For queues that don't support the exactly-once guarantee, you can use at-least-once delivery and handle the idempotency on the application level by having your data store for duplicate events.

Trade Off

The throughput will be the worst here because of the need for an idempotency check but with the simplicity of exact guarantee as most people expect a queue to behave.

Use Case

Suppose you need to send a payment request to a third party that does not handle idempotency, you need to ensure the same payment doesn't get charged twice.

If the downstream processing is costly to call and reprocess, you may want to ensure each event is only handled once instead of wasting resources to reprocess.

Custom Queue

Queues can take many forms. A queue doesn't have to be a pub/sub or message queue necessarily. Here are some examples:

Durable Priority Queue

You can implement a queue using a relational database table. For example, you might have a priority queue table of column `event_id` and `priority_number`. You index on the `priority_num` and fetch for the events with the highest priority number to process. Or you simply have a table of `event_id` and `status`, and you grab all the events in the pending stage.

Event	Priority
1	100
2	88
3	70

In-Memory FIFO Queue

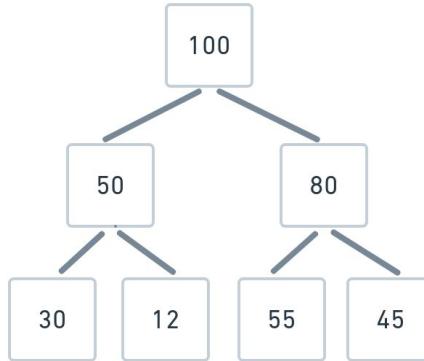
The queue can also be an in-memory Java concurrent queue that's on the application server. Just make sure in a system design interview, you develop a queue solution that makes the most sense with your application.



In-Memory Priority Queue

The following is the max heap data structure where the root has the event with the highest priority. In a system design interview, you need to worry about concurrent threads accessing the heap and the throughput implication.

Also, since it is in-memory, you need to worry about what happens when the queue goes down.



Conflict Resolution

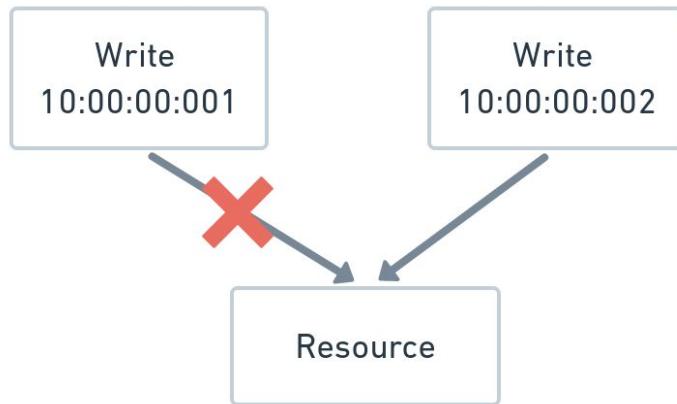
There may be multiple writers to the same key to different nodes simultaneously in leaderless replication or leader-leader replication. When the writes need to resolve the conflict, it's unclear how the system should resolve the conflicts.

The write conflict is similar to code branching, where two people are working on the same code, and when they both need to merge into the same branch, it's unclear how the engineers should resolve the conflict. There are many possible strategies, and in an interview setting, you need to ensure the conflict resolution strategy will continue to deliver a good end-user experience.

Here's a list of potential strategies:

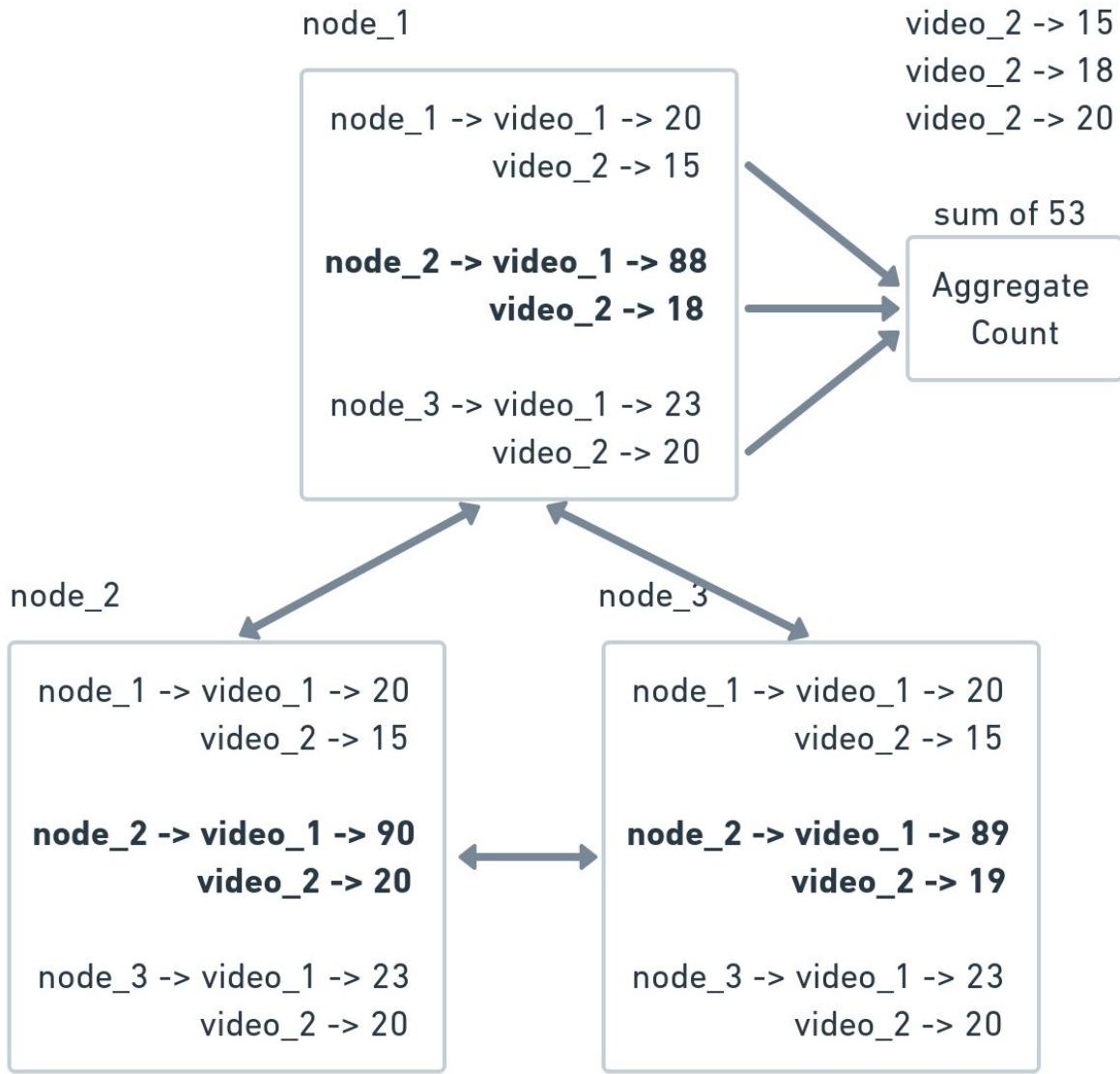
Last Write Wins

In last write wins (LWW), you can attach a timestamp to the data to resolve a conflict. The data that has the latest timestamp wins. LWW can be lossy, and due to clock skew, a later timestamp doesn't always mean it's happened later from the perspective of the same user. However, depending on your application, this may work most of the time, making it a practical and simple solution.



Conflict-Free Replicated Data Type (CRDT)

Instead of dealing with conflict, there are specific applications that can use CRDT. In CRDT, you can replicate data to each other. There's no need for coordination between the nodes because they are conflict-free. For example, assume the question is a global distributed counter x . Multiple nodes are taking in requests to increment and read x . It's difficult to determine what x should be when a node passes its x to another node. With CRDT, each node keeps track of its count along with other nodes' counts. For example, say there are 3 nodes, `node_1`, `node_2`, and `node_3`. When writes go to `node_1`, `node_1` increments its count. `node_1` can replicate its count to `node_2` and `node_3`, and they will update `node_1`'s count within its node. Each node just sums all the counts up to get the total count. Assuming an asynchronous replication, the count will be *eventually consistent*. Here's an example:

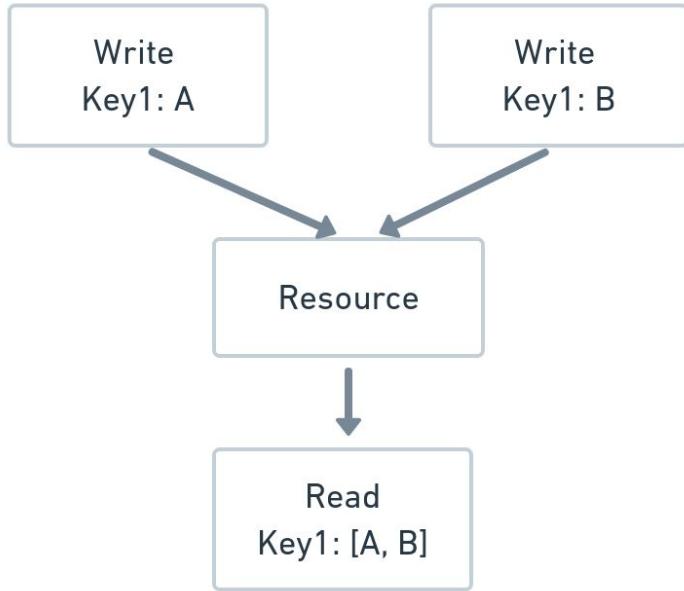


The upside is availability since any node can serve reads. The latency is better because you can get the count from a single node instead of a scatter-gather from all the nodes. The downside is the complexity of the broadcasting. CRDT has other applications as well, like resolving Google Docs operations. When asked about distributed counters and Google Docs, this might be useful in a system design interview.

Keep Records of the Conflict

One way to deal with the conflict is to not resolve and keep the conflicts persisted. When the application loads the data, the application will see the data that led to the conflict. For example, if node_1 has "animal": "dog" and node_2 has "animal": "cat," when the conflict is resolved, it has

“animal”: [“dog”, “cat”] and the application can decide what to do with this information. Keeping conflicting data comes at the expense of complexity for all the subsequent readers of this data. However, the system retains all the information and therefore is not lossy.



Custom Conflict Resolution

There are many conflict resolutions out there, and if it happens during the interview, you can think of some solutions that make sense for the question. For example, a software engineer has to resolve the conflict manually when there's a code conflict. It is difficult to resolve code automatically.

Security

Security is a broad topic that ranges from physical security to software security and is typically not covered much in a generalist system design interview. Even if it is covered, it's usually a test of knowledge rather than a test of problem-solving since they're generally not unique to the specific question, and you can apply the concepts to all questions. However, it is useful to consider the security of your end-user APIs if you have them since they are unique to your design.

We've put some security topics in the appendix if you wish to learn about them.

API Security Considerations

The user can perform a malicious act. In a system design interview, it's worth thinking about the worst things that can happen to the API and talk through how you would mitigate them. Here are some examples:

`transfer_money(amount, user_id, to_user_id)`

What if the client can specify any `user_id` and `to_user_id` to themselves?

What kind of validation should this have?

`upload_photo(user, photo_bytes)`

What happens if the user uploads some malicious bytes?

`request_ride(user, from, to)`

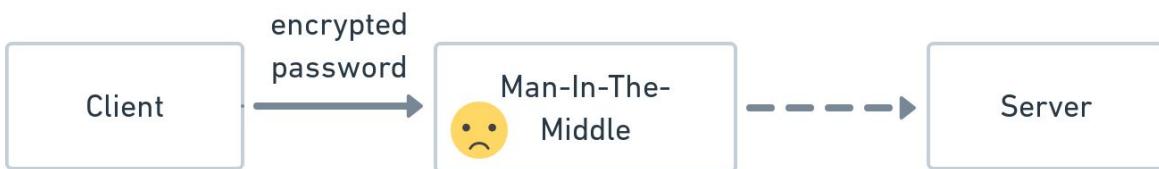
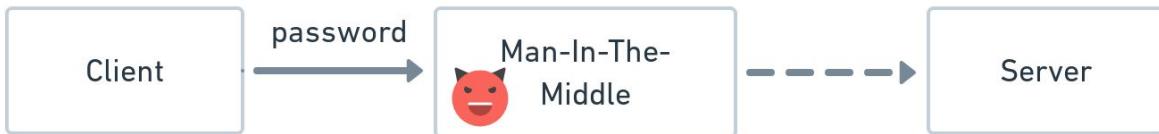
What happens if the user enters locations from other continents with no intention of getting picked up?

`place_order(user, item, quantity)`

What happens if the user enters a huge quantity? What if the user specifies an item that is no longer available?

Man in the Middle Attack

This usually isn't unique to any system design question but is a likely trivia you may be expected to know in an interview.

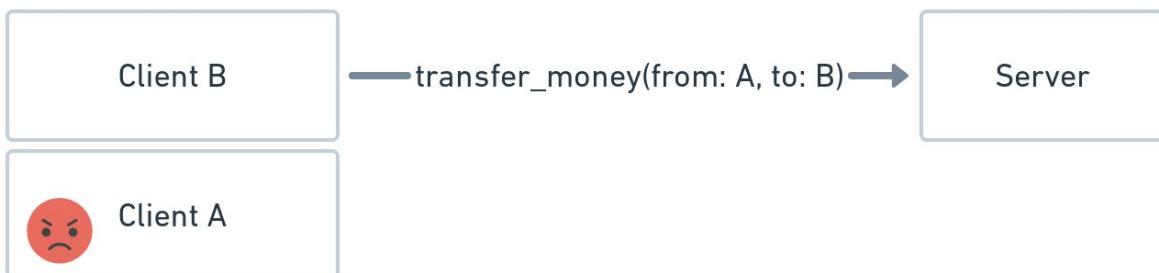


The idea is if someone is able to intercept your request, the man in the middle can capture important information like username, password, PII, etc. Man in the middle happens a lot with public wifi since anyone can receive wifi packets. The way to deal with it is through Transport Layer Security (TLS), where the packet is encrypted. The client can do a TLS key exchange to verify the server's certificate with a certificate authority. Subsequently, it uses that public key to encrypt the data, and the server will decrypt using their private key.

If you're interested in the exact flow, there are plenty of videos on the topic. Don't stress over the exact flow since it's very unlikely the interviewer will quiz you about the steps, just understand the problem TLS is trying to solve.

Authentication

Another common security topic that will likely come up is how the server knows if the client calling them is who they say they are. If the server trusts anyone, anyone can transfer money to themselves, request a ride for themselves, view personal photos of other people, etc. It's worth mentioning what would be the worst-case user story if authentication is compromised.



You can achieve identity by having the client log in via some protocol like OAuth, where the client maintains a token that identifies themselves. The system can periodically refresh the token in case the token gets compromised. The client passes the token in subsequent calls, so the server knows the requester is who they say they are.



Timeout

Timeout is super essential in distributed systems because you can never detect an actual failure. For example, imagine you have a service trying to request a resource from another service. If the service doesn't respond, it could result from network congestion, a slow server, slow server dependencies, etc. Since you may never get a response back, the client shouldn't be sitting there waiting. Otherwise, it will use up its resources unnecessarily by waiting. You can resolve this by setting a timeout to mark the request as a timeout failure. The client may decide to try again as a result.

The trade-off for timeout is that a longer timeout will waste more resources by having the client wait for a response, but if the result does come back, the client doesn't need to request it again. For example, imagine the processing time takes 1 minute, and the timeout is 50 seconds. The system just wasted 50 seconds. If the client had waited 10 more seconds, they could've gotten the result. Now it has to retry again and potentially lead to another timeout failure. However, if the service is truly unavailable, you will have only waited for 50 seconds instead of a longer timeout.

Here the client could also be the end-user. In an interview, it's worth talking about the user experience of how a timeout may affect your design. For example, if you set the timeout for too long in a ridesharing service, the end-user ends up waiting for a long time before the system tells them, "Please try again." Having the end-user wait a long time can be frustrating.

In an ideal world, the timeout should be as long as the request's processing time, but in reality, you never know how long it will take to get the final result back.

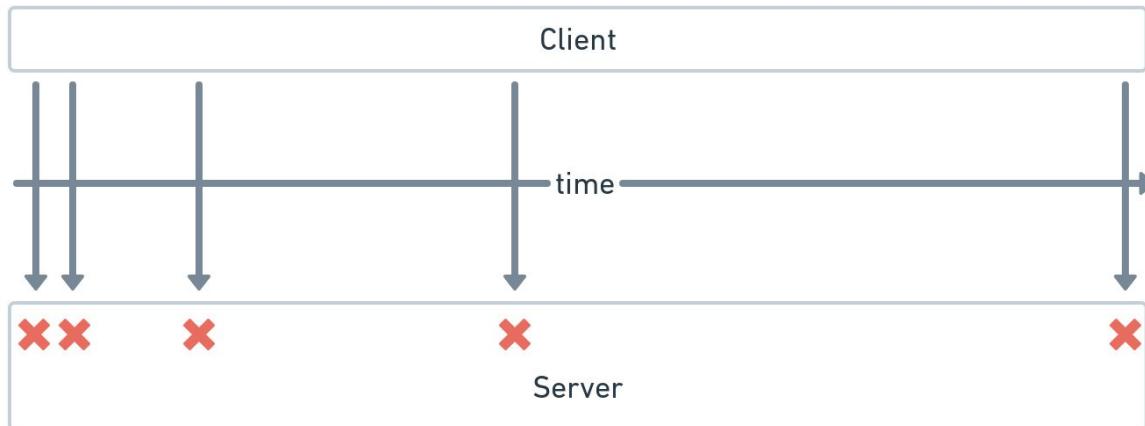
Timeout is also frequently used by service discovery to determine which hosts are still available and add and remove the physical machines from and to the shards. It's also helpful to determine if a leader is healthy and whether the system should elect a new leader. A longer timeout means it

will take longer to find out a node is unavailable, whereas a shorter timeout means the node is still available.

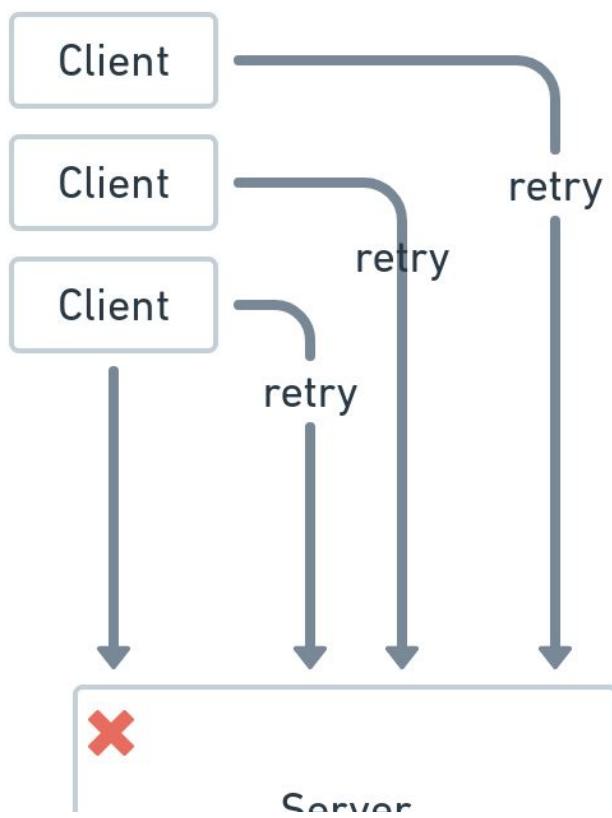
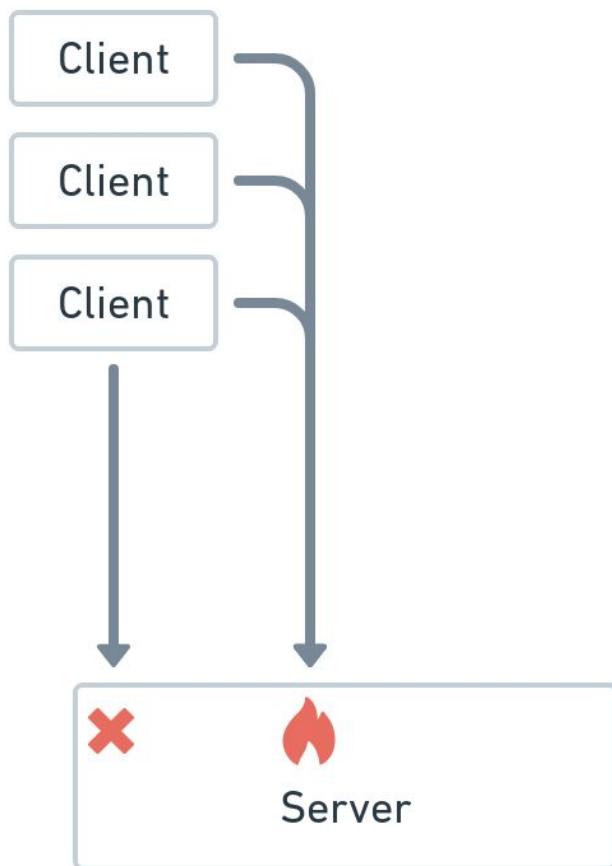
Timeout can be used in multiple use cases, so think critically about how the timeout duration impacts your design.

Exponential Backoff

In system design, the downstream could have temporary errors that could go away through time. It could be the server is busy due to a sudden unexpected traffic, it could be a bug that later gets rolled back, it could be network congestion that resulted in timeout.



The client may decide to retry on certain types of errors in anticipation it will be fixed on the next retry. However, a frequent retry could cause a snowball effect since there will be more requests backed up. Usually this is mitigated with exponential backoff with maximum retry. At some point, perhaps the error can not be fixed. After maximum retry is reached, most likely manual intervention is required as it signals a true downstream failure.



In a system design interview, it is helpful to discuss what happens if there's a temporary first or third party downstream failure. A bigger exponential backoff will cause delay in processing but eases the downstream by too many client retries. A smaller exponential backoff will get a quicker response if the service is back alive, but at the expense of more stress to the downstream service.

Buffering

When you are dealing with a throughput problem, you can consider buffering. When the end-user client passes data to the data store, there is overhead associated with each request. For example, each request needs to establish a TCP connection, load balancer, internal RPC calls, disk IO, etc. Each of them has non-trivial processing to it.



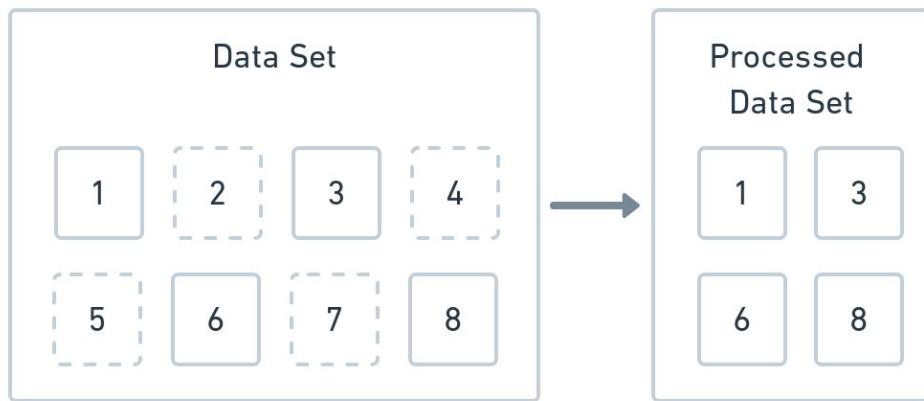
For example, imagine the client is omitting metrics data. Instead of sending data one at a time, it may wait for a second and collect hundreds of data points before sending it in.

The trade-off here is the longer you wait, the less overhead you incur per request. However, the downside to buffering is that it decreases freshness since it requires collecting more data first before sending it as a single request. Therefore, if freshness is an important non-functional requirement in a system design interview, you will want to set your buffering appropriately.

Sampling

Sometimes in a system design interview, you may agree with the interviewer that you don't need complete accuracy in favor of better performance. For example, reducing the amount of data processed can reduce the computation and storage needs at the expense of less accuracy.

In an interview, when discussing the bottleneck of high QPS or storage, try to think if there are areas where you can reduce the amount of data processed and storage at the expense of accuracy, but still provide a good user experience.



Reduce Storage

Imagine in a ridesharing service design you're designing the feature where you show the user where a vehicle has traveled during the ride. The driver's phone continues to send location updates every 5 seconds to a location service. You listen to that location update, but instead of processing and saving every update, you can just sample it for a fraction of the data. That way, you reduce to a fraction of QPS and storage. The downside might be that the final route won't have granular information about the ride, but that might be ok since the riders don't need a super detailed ride path.

Reduce Computation

To compute a recommendation, you may figure out what another similar user purchased to make the final recommendation for a given user. Instead of calculating it for every user, you may decide to reduce the search space

to make the computation less burdensome. Sampling may make sense if there is a large enough sample to derive a statistically significant result.

ID Generator

ID generation can be important during system design interviews for various reasons. Sometimes it's a system design question in itself. However, ID generators are more like tools, and you should be using the right generator for the correct use case. Here are some common ID generators.

UUID

In UUID Version 1, it uses MAC address and timestamp to generate effective uniqueness.

Pro

- Any server can independently generate a unique ID without any coordination.
- Effectively unique with the rare chance of duplicates.

Con

- Generated IDs are not ordered.
- 128-bit may be too big for an ID for some use cases.

Use Cases

- Pretty much anytime you want to generate a unique ID where ordering or 128-bit are non-issues.

Auto Increment

Databases like MySQL generate ID in auto-increment.

Pro

- Guarantees ordering and uniqueness.
- Simplicity to use a database to generate.

Con

- Not horizontally scalable since you only have one instance.
- Not fault-tolerant since you only have one instance to generate ID.

Use Cases

- Small scale applications.

Auto Increment Multiple Machines

Increase throughput with multiple databases. You can have a machine generate a subset of data. A famous example by Flickr is to have one server generate odd numbers, and the other server generates even numbers.

Pro

- Simple solution to add more machines.

Con

- It is not flexible. Imagine adding a third machine; you would need to reconfigure the servers to generate in multiples of 3.

Use Cases

- Medium scale applications where you need a quick solution.

Strongly Consistent and Fault Tolerant

Strongly consistent (linearizable) system like ZooKeeper can generate guaranteed monotonically increasing integer numbers using zxid.

Pro

- Ordered and unique.
- Fault Tolerant.
- 64-bit number.

Con

- Due to the leader-follower quorum to guarantee strongly consistent ordering, the throughput is lower.
- Complexity to maintain a ZooKeeper cluster.

Use Cases

- Fencing token for distributed lock.
- Snowflake machine ID generation when initially booted.

Distributed Roughly Sorted ID

There's a famous ID generator called Snowflakes created by Twitter.

Timestamp 41 bits	Machine ID 10 bits	Sequence 12 bits
----------------------	-----------------------	---------------------

Timestamp: In millisecond precision on the host generating the ID.

Machine ID: Assigned by ZooKeeper during machine bootup. Because it's 10 bits, it gives up to 1024 machines.

Sequence: The host generating the ID will increment the count every time an ID is populated. Since it's 12 bits, it goes up to 4096 numbers. So yes, it's still a potential problem if a machine gets more than 4096 within a millisecond.

Pro

- Roughly ordered and unique.
- Can handle high throughput.
- No need for machine coordination.
- 64-bit number.
- Can horizontally scale by adding more machines.

Con

- Still not perfectly ordered. If two different machines generate two IDs, it's unclear who was generated first.
- Complexity to maintain since Snowflake requires machines to do the work like a UUID library.

Use Case

- Tweeter wants their messages to be less than 128 bits that are roughly ordered with high throughput.
- Discord wants their messages table's cluster index to be unique and roughly ordered with high throughput.

Offline Generation

The algorithms mentioned above are all generated in real-time, hence the complexity of quickly calculating something unique, small, and ordered. Another way to handle it is to generate the IDs offline. The exact format and algorithm are up to the product use case.

Pro

- Latency is low for any custom ID sequence generation because numbers are already generated. For example, if you want short, ordered, or unique IDs.

Con

- Since IDs are generated offline, you don't know if you have generated enough. You will need a backup if you run out of IDs.
- If you over generate the IDs, you might be storing more ID than you need.

Use Case

- Perhaps for TinyURL you decided with the interviewer you need a shortened URL that is small, and you don't want to have the risk of multiple retry loops if a duplicate shortened URL is generated.

Custom ID

Some IDs don't need to be unique, ordered, or a specific size. ID can be String, integer, big integer, timestamp, hashed value, and it depends on your use case. In an interview, you need to clarify the requirement. For Snowflake IDs, you can configure the segments for your use case.

Compression

In a system design interview, you may face a bottleneck situation where too much data is passing through the wire and results in a bandwidth issue. In these situations, you can consider compression.

Compression is a very deep topic in itself and, by nature, very niche. Applications can apply different compression algorithms to different file types that lead to different compression efficiency. It is unlikely the interviewer will dig deep into the specifics unless you have experience in the domain. It's more important in an interview to articulate what you're trying to achieve with compression and what would be some of the considerations and implications for doing so.

Warning

Many people confuse compression, encoding, transcoding, and codec, so it's worth going over the basic definition for each.

Encoding

Encoding is the process of compressing a raw piece of data into an encoding format. An example would be from a RAW image to JPEG.

Transcoding

Transcoding takes an encoded format into another encoding format. For example, taking a JPEG image and converting it into different aspect ratios.

Compression

To reduce the size of a piece of information. You can conduct compression through encoding and transcoding.

Codec

Codec is the actual algorithm to compress and uncompress a piece of data. For example, H264 is a popular codec used for videos.

Here's a list of topics to consider:

Lossless vs Lossy

When you compress a file, it could be lossy compression. Lossy compression happens when you can never get the original file back. Lossless compression looks for file patterns to come up with another representation that requires less memory.

Lossless and lossy compression can co-exist. We can apply both compression techniques for different purposes. For example, an image can apply lossy compression permanently into JPEG that other software and users can use. The system can compress the image further by applying lossless encoding to be efficiently stored into databases, but software can not immediately use the format because it requires decompression.



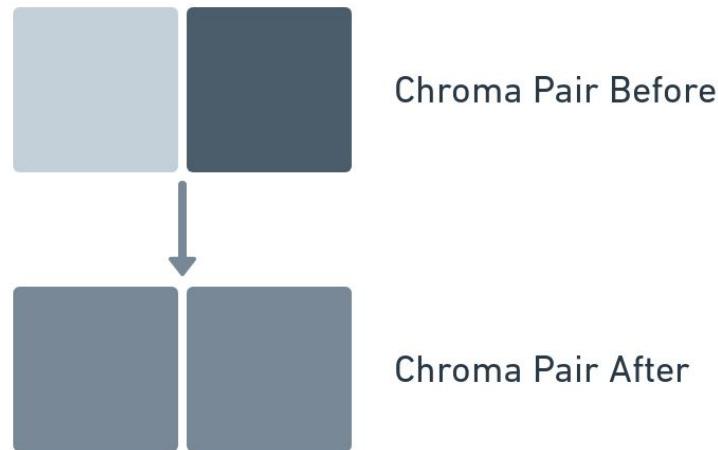
We will go over some compressions. Don't stress over the exact compression algorithms. The goal here is to build intuition, so the distinction between the two is clear.

Lossless Compression with Run-Length Encoding

Run-length encoding is a lossless compression algorithm. Instead of storing a bunch of characters, it compresses into a running length. For example, if you have AAAABBBBBBBBBBBBAAA, which takes 22 characters, you can store it as 4A15B3A which is significantly shorter. And you can always rebuild the original string from the compressed format. Of course, the efficiency depends on the pattern. If it is ABABAB, it'll be encoded as 1A1B1A1B1A1B, which might turn out to be worse. You can use RLE for images like JPEG.

Lossy Encoding with Chroma Subsampling

In chroma subsampling, you can model image tiles by combining luma and chroma, and luma information on the image is preserved while the chroma is compressed. For example, imagine the original image has a chroma pair. After compression, the adjacent pairs share the same color. It should be evident that you can't recreate the original color pair from a single color.



Compression Efficiency

Compression reduces the size, so it's important to figure out how much compression reduces the memory. The efficiency is measured by compression ratio:

$$\text{Compression Ratio} = \text{Uncompressed Size} / \text{Compressed Size}$$

If I compress a 10 MB file into 1 MB, the compression ratio will be 10. The higher the compression ratio the better. Lossless compression ratio for image is usually 2, lossy compression heavily depends on the encoding scheme. For example, the photos you take on an iOS smartphone are HEIF format, which is generally 2 MB. When you upload the photo to Instagram, it gets encoded into a 100 KB JPEG file to be viewed by your followers with a 20 compression ratio.

Quality of File

The “quality” in quality of a file heavily depends on the context. For images, videos, and audio, it is the end-user perception of quality. You trade off the quality of the file to reduce size in lossy compression. However, in practice, efficient lossy compression like JPEG has minimal impact on the

end-user perception for natural photographic images. However, JPEG isn't as good at compressing for synthetic images like web graphics. Usually, you can use png and vector graphics instead. While you don't need to be an expert in domain compression algorithms, you should touch on how you will monitor end-user experience to ensure the compression does not worsen their experiences.

Computing Time to Compress a File

We have discussed some compression algorithms, and as you can see, compression is ultimately about finding patterns and coming up with formats that data can represent in another format. Compressing takes computation time, especially for videos where there is a lot of data to be processed. Encoding some videos may take as long as the video itself. Generally, the more efficient the algorithm, the longer it takes to process due to the need to conduct a more thorough pattern discovery. Processing time has implications on whether the client or the servers can do compression. The challenge for backend compression is whether there are enough machines to process all the files since compression can be compute-heavy.

Compatibility of Devices

Some mobile clients and applications can only handle a certain type of file encoding. While the mapping of device type to encoding type is a niche topic, it's important to realize that the back-end needs to encode multiple types to support different users. Here are some examples:

Video	Can be encoded into H264 and VP9 and for each codec, it can be encoded into different resolutions 480p, 720p, and 1080p.
Image	Can be encoded into JPEG and can be transcoded into different aspect ratios and dimensions. Some common aspect ratios are 1:1, 3:2, 4:3, 16:9 and dimensions 1080 x 1080, 1280 × 720.
Audio	Can be encoded into MP3, OGG with different

qualities like 160 KBPS, 320 KBPS, etc.

Compressed File Usage

We have spent some time talking about how to compress files. In a system design interview, it's essential to know how to use compressed files.

Dimension and Aspect Ratio

In the last decade, the number of clients has increased with tablets, mobile phones, and the web. As a result, there are many screen sizes, aspect ratios, and resolutions for the viewport for each kind. Furthermore, the browser can be responsive to viewport size, creating the demand for different dimension and aspect ratios for videos and images.

Quality Encoding

Imagine a client doesn't have a good bandwidth network for image, video, and audio. For example, Facebook is trying to reach a global audience. Therefore, it needs to support users in low bandwidth areas as well. Different encodings can be used via adaptive bit rate (ABR) by dynamically adjusting the compression quality to better fit bandwidth availability.

Pass Only Needed Data

In a system design interview, you may run into the bottleneck of passing too much data over the wire. You can consider a tactic to only give what you need to pass.

Filtering

Sometimes the amount of data from one server to another server may be unnecessarily huge. For example, suppose you're designing an e-commerce platform and just need a field for a product in a product catalogue with a bunch of metadata. In that case, each product catalogue has a bunch of metadata. Instead of passing the whole metadata dump, the client can specify just the field to look for and reduce bandwidth through the wire. The trade-off is better bandwidth at the expense of the complexity of the client to specify the filtering scheme.

Pass Chunk Delta with Rsync

Instead of passing the whole objects, depending on your use case, there may be an opportunity to pass the delta or incremental change. Again, there are many algorithms for this depending on the use case, but here's one example you may have used before with rsync:

Each 2,048-byte chunk is calculated using a hash function like MD5 into 128 bits which is 16 bytes—significantly less data to pass. This is how it works:

Step 1

Compare the first chunk's checksum to see if they're the same, if they're the same, move on.

New File

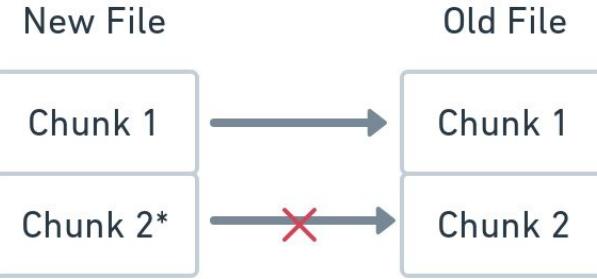
Old File

Chunk 1

Chunk 1

Step 2

Detect that the new chunk is different.

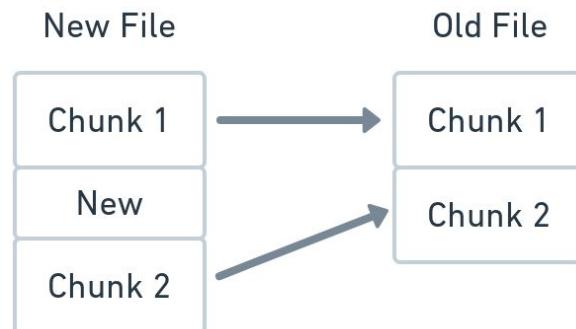


Step 3

Calculate the next hash by moving a byte forward at the head and tail.



In the above diagram, to calculate chunk 2, you remove 1 and add 2 and you don't need to recalculate all the blocks in “-”. This is called rolling hash.



When a new chunk is detected like the above, only the new chunk is passed. And if it's the same block, you move on to the next block.

Passing what you need might sound like a no-brainer. However, remember there's an overhead of passing the checksum to the server to compare the

checksum. For example, if you try to copy a file over the network with rsync, you're introducing a lot of overhead when you can straight up copy the file over. Use rsync when the files are similar. If the files are completely different, it's better to pass the new bytes since there's no point in comparing the files.

Fail to Open

In an interview during the deep dive session, while you're discussing possible failure scenarios, most people will talk about the redundancy of another storage or service to take over through election or manual intervention. However, those still take time and don't always result in a smooth automatic failover. There can be creative approaches to provide a backup experience to be perceived as available for the end-user. In an interview, you can come up with options and just describe the trade-offs, but make sure the selected one still shows a good user experience. Here are some examples:

Faulty Payment Processor

When you're designing an e-commerce checkout process, if the payment processor is down, instead of dropping the order and resulting in revenue loss, you may let the orders through and collect the money later. The trade-off is the risk of not collecting the money later due to fraud or expired payment to benefit much better checkout availability.

Faulty Storage

When a storage node is down, it's possible to store the data in another node as a temporary holding space. Once the server is back up, the node that temporarily holds the data will transfer the data to the original node. This mechanism happens in Cassandra hinted handoff and consistent hashing where another node takes the write on behalf of the original node. Hinted hand-off comes at the expense of inconsistency for Cassandra and additional load for consistency hashing.

Faulty Price Estimate

When you're designing a ridesharing service for an estimated price and the price estimate calculator is down, you may decide to call another backup service for similar historical data or just calculate the baseline price without the surge multiplier. The trade-off is availability at the expense of revenue loss due to the lack of a surge multiplier.

Distributed Transaction

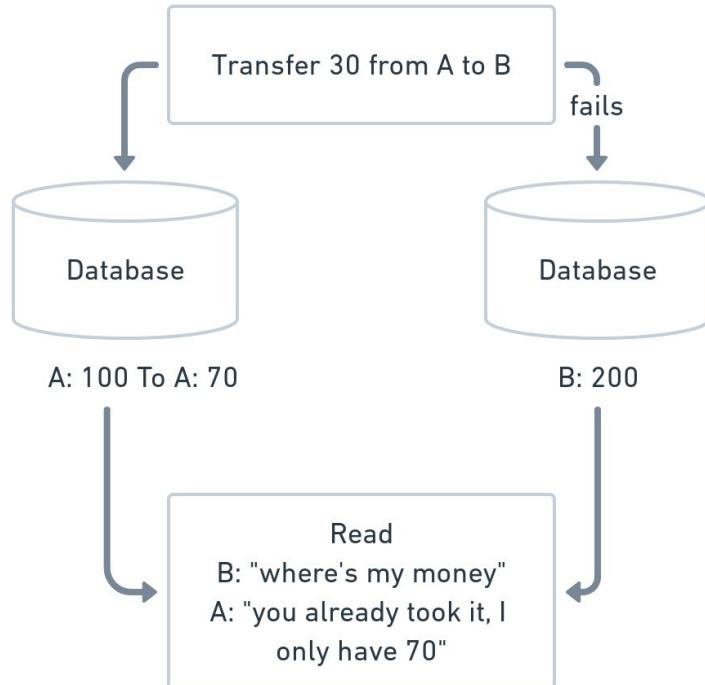
A distributed transaction is a transaction that involves more than one data source where the complexity comes when a subset of the data sources successfully commits. At the same time, the other subset fails to commit. This complexity comes up in an interview more often than you might think. While you don't need to jump into the complexity during high-level design immediately, it's something that the interviewer may ask you about or something that you may want to address. Here are some very common examples in a system design interview:

Money Transfer

```
def send_money(from_user, to_user, amount)
    from_user_database = get_database(from_user)
    to_user_database = get_database(to_user)

    to_user_database.add_amount(amount)
    from_user_database.remove_amount(amount)
```

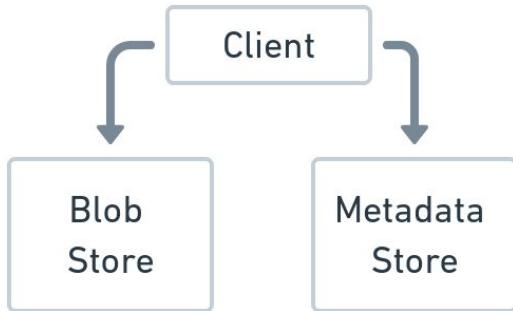
Imagine you are doing a code review, and you come across this simplified code. What would you be worried about? This call is problematic when `add_amount` is committed successfully and `remove_amount` fails to commit. Perhaps the `from_user` no longer has enough balance, or the `from_user_database` is just unavailable now. You will be left in an inconsistent state where the money has already been removed from the `to_user_database`.



Usually, a system can write to multiple sources with 2-phase commit (2PC) with a prepare and a commit phase. The idea for 2PC is that once both sources decide they are ok with the commit, the coordinator will continue to push the 2nd phase commit until they go through. If there's only 1 phase, if one commits and the other one decides it can't commit, the system will be in an inconsistent state. The challenge of 2PC is that if the coordinator is down, the service becomes unavailable.

Blob Storage and Metadata Storage

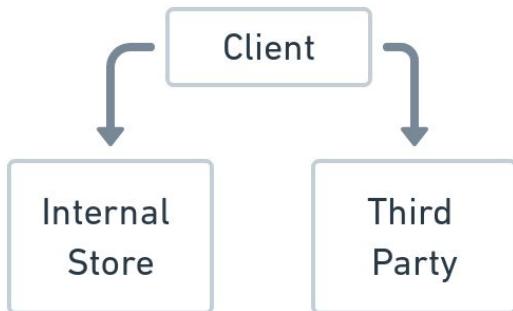
Imagine the scenario where you upload a photo and metadata. If you persist the metadata first, then the photo, what happens if the photo commit fails? You will have a metadata row with no photo blob. Of course, you can also have a 2-phase commit coordinator, but that increases the complexity and throughput, and sometimes you're not in control of the service like Amazon S3.



You can solve the issue by persisting to the blob store first to get the URL then store the URL in the metadata. Then, if the save to the metadata store fails, you're left with an unreferenced blob. You can have a background cleanup job to clear the unreferenced blob. Or you can just leave the unreferenced blob alone at the expense of more data and cost with less complexity without the background job.

Third Party Data Source

In some designs, you will have a service where you don't have control over the API contract and design, and you need to make a distributed transaction to them.



Database and Queue

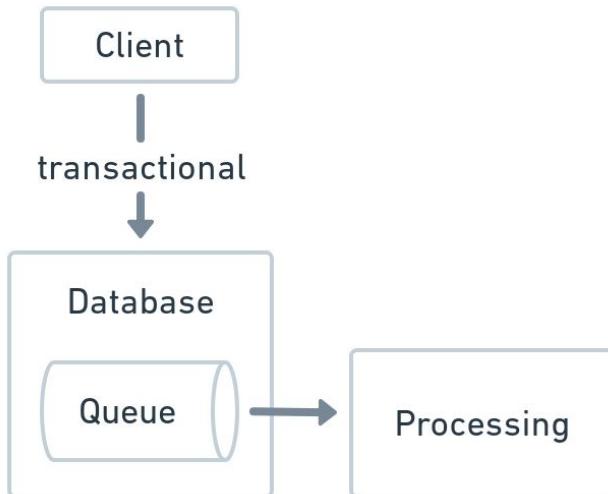
In many system design questions, you may write to a database first before omitting an event to a queue to be processed asynchronously. The complexity comes when the insertion into the queue fails, and the event could be lost if not handled appropriately.

For example, when you design an e-commerce checkout pipeline, you may create a record for the order placed and involve an asynchronous workflow to fulfill that order. However, if the insertion into the queue fails, that order

may just be stuck without ever getting fulfilled. Here are a couple of options:

Solution 1: Database Queue

Some databases support a post-processing queue where the query to save the record and insert it into the queue is transactional. Sometimes this isn't an option if the database doesn't come with queue support.

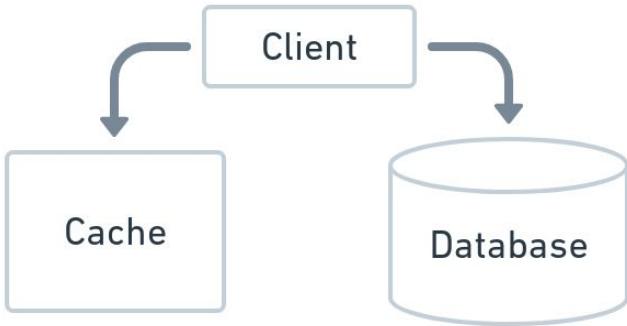


Solution 2: Let It Be

The failure rate into the queue may be so low that occasional event loss is acceptable. Perhaps you have a background job that occasionally checks for unprocessed records. The solution depends on whether data loss is acceptable. Think about what the end-user will experience with the occasional data loss. For example, if you have a checkout pipeline and the queue sends an email to notify them of the success, an occasional email loss might be acceptable.

Cache and Storage Update

When you think about write-through cache where the request to update a record updates the cache and the store at the same time, you can guess there will be an issue where one of the two fails. It is potentially worse if the cache is updated but the underlying store isn't, as the cache will be serving a value that wasn't persisted to disk. That's why write-through cache can be quite challenging.



Abstract Design Choices

We have discussed some system design scenarios that could potentially come up and some of the common ways to deal with them. More abstractly, you can think about the following choices as potential design and for each design, think about the implication for the end-user. For example, imagine you have Service A and Service B.

Option 1: Call A then B.

Option 2: Call B then A.

Option 3: Call A and B concurrently.

Option 4: Call A and B with a coordinator.

Option 5: Make A and B transactional.

Each of them has its pros and cons. For example, if asked about the distributed transaction complexity in an interview, you can bring up 2 to 3 ideas and go over them with the interviewer. For each option, just think about what happens if there's a partial failure.

Cold Storage

Data will continue to grow as long as you have users, and as the amount of data grows within the storage, the performance will degrade through time, and eventually, storage will run out of memory. Fortunately for most applications, recent data is usually accessed more frequently than past data. You can take advantage of that fact by moving data from hot storage to cold storage. Since the performance for cold storage isn't as performant and available, it's usually cheaper to maintain, although there's the complexity of moving data from hot to cold storage.

If you are in a situation where the interviewer asks you what would happen if the data continues to grow, you can think about separating infrequently accessed data into cold storage. Make sure after you move the data, the performance constraints are still satisfied.

Networking

Networking is a very broad and deep topic. Fortunately, unless you're interviewing for a networking specialist role, there shouldn't be too much networking-specific trivia they will ask you. However, it is still essential to learn about some core networking concepts because they are essential in providing the right level of details regarding scalability.

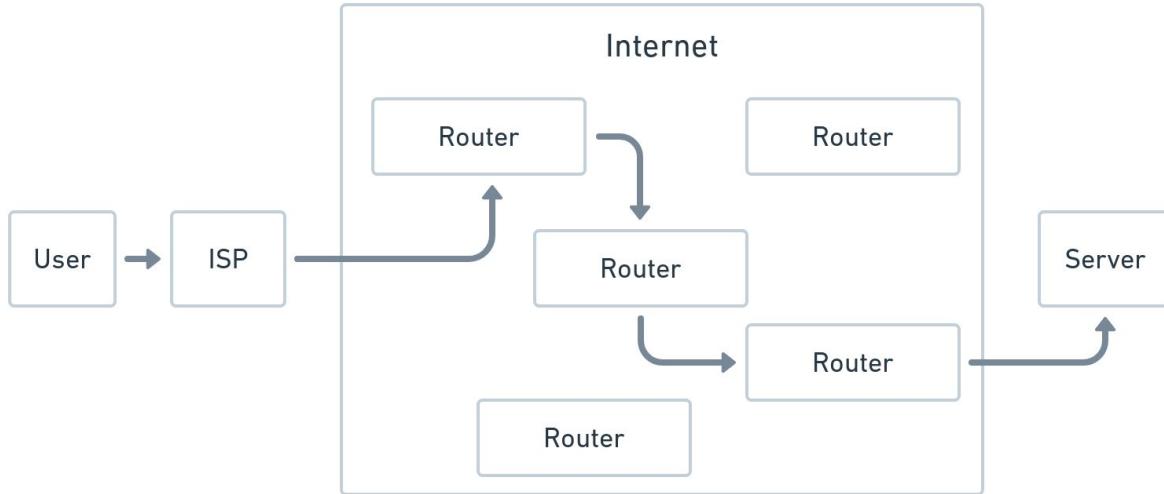
IP and Port

When you have a device or a server, it has an IP as the home address of the internet. There can be many port entry points for a given device IP that listens to requests. When you make a request, you can specify an IP and a port to route the request to.

Reminder

You might wonder why you never specify an IP or port, but you're still able to load <https://www.google.com>. When you enter <https://www.google.com>, the browser will ask DNS servers which IP the domain google.com belongs to. Since the protocol is HTTPS, the server uses the default port 443, so a user doesn't have to specify it. Another example is HTTP's (without the s for secure) default port is 8080.

When you request through a browser, it typically goes to the ISP and gets routed through a series of routers (also called the internet) before reaching the server IP. When a server IP needs to send a notification back to the client, it goes through the internet and eventually back to the client. A TCP connection essentially consists of `client_ip`, `client_port`, `server_ip`, `server_port`.



Why is knowing IP, port, and router important in a system design interview?

Geo-Sharding

During the deep dive design, you might decide to geo-sharding the service. When multiple data centers are taking in the same request for a server, it's important to know how it gets routed to the most appropriate one.

Data Center Failover

During system design deep dive, you might want to improve your system's availability by having other data centers take in requests when the main data center for a given user is down. It's important to know how the request doesn't continue to hit the server that is down.

CDN

Like geo-sharding, for a given CDN domain, how do you route to the closest CDN for the best performance?

Describing How the Internet Works

Sometimes the interviewer may ask you to describe how the internet works. If there's a basic networking knowledge gap, the interviewer can tell you are hand-wavy about some of the important details. For example:

“The client will route to the closest CDN for best performance.”

“When the data center is down, the request can be routed to the other one.”

While these phrases are acceptable, sometimes the candidate trivializes the complexity of routing requests through the internet.

WebSocket

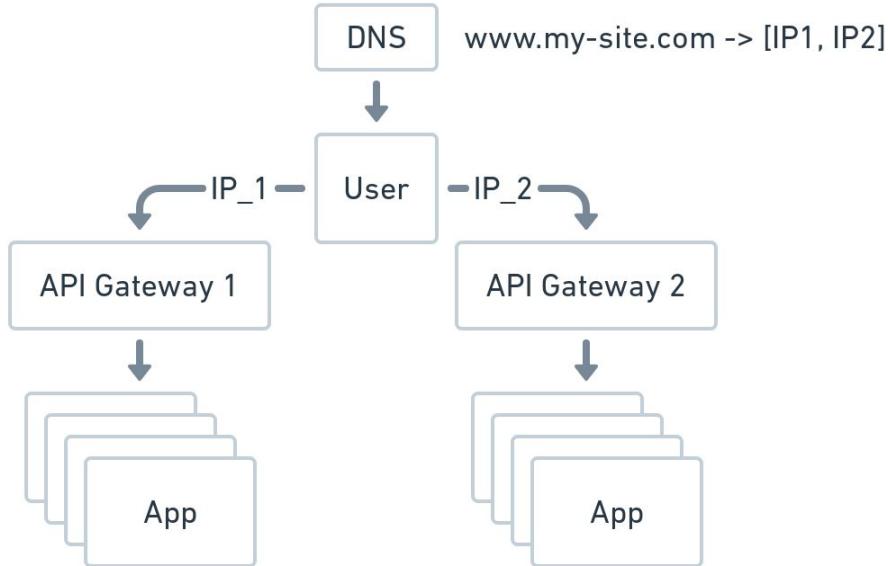
When a client establishes a WebSocket connection with the server, how do they communicate with each other? What happens if the WebSocket connection to the server goes down?

Domain Name System (DNS)

Domain Name System (DNS) is a service that converts domain to IP. Domains like google.com are human-readable, but you need an IP to route a request to a Google-owned server. The browser will ask a DNS server for an IP and cache it in the browser with a TTL.

DNS usually doesn't come up much since it's not unique to any questions. However, it's still worth knowing how the internet works if questions like, "Tell me what happens when you enter www.google.com," or, "The website is down, can you please debug," come up.

One thing worth knowing is DNS round robin, where you load balance via IP when there are multiple IPs for a given service to improve redundancy and availability. However, this can be problematic since many browsers cache the DNS, and the system doesn't have direct control over how the client manages the cache. However, it's still worth mentioning as a possibility when talking about the fault-tolerance of a region.



How to Route to the Closest Region

In a system design interview, you will sometimes want to discuss how to improve the availability and performance of your system. One way to do this is to have geo-distributed edges and servers. There are usually two things you want to know:

1. How a request routes to the nearest data center.
2. What happens if a region is down, and how does the system route to the appropriate data center or edge to serve the traffic?

You can apply the geo-distributed concept on edge servers, CDN, data centers, etc.

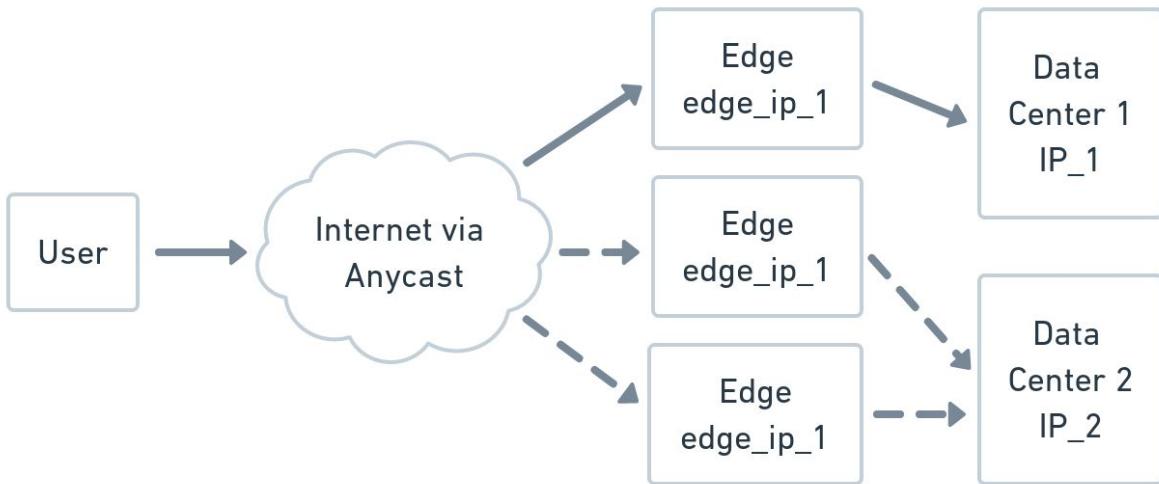
Option 1: Use DNS Routing

The responsibility of a DNS server is to return the IP to route your request to. For example, a DNS server can look at the client's IP and return the client's closest IP to hit. Or, if a server IP is down, you can configure the DNS to hit another IP address even if it's further away.

The advantage is improved performance with an IP that is closest to the user and improved availability to have another IP server as a failover. The disadvantage is the complexity of DNS logic to return the most optimal IP.

Even more importantly, clients usually cache the domain to IP due to performance needs with TTL. So if an IP is down, the cached IP browser may continue to hit the failed server, and the server has no control over the client-side browser cache.

Option 2: Use Edge Router



Each server still has the concept of static IP but instead has edge routers that have the same edge IP for anycast. In an anycast, multiple edges can be assigned the same IP.

The question is, how do we know which edge to hit? That's where anycast comes in. When a request goes through the internet and through the routers, each router will select the most optimal router to forward to for a given IP. It's almost like the shortest distance graph question where each node knows the most efficient edge to forward to. Once the request hits the edge router, the edge router will forward the request to the most optimal data center to serve the request. The edge router knows which data center is the most optimal because it continuously monitors the health of the data centers, and the closer and healthier the data center, the better. Also, the network between the edge servers and data center is usually company-owned, so it's a lot more efficient than the internet.

OSI Model

If you want to learn more about the networking stack, it might be helpful to know how to create the packet with each stack.

For a system design interview, you don't need to know the trivia of what is in a packet. What's more important is knowing what commonly mentioned terms belong in which layer. In a generalist interview, layer 7 and layer 4 are the most important ones to be familiar with.

For example, you shouldn't think TCP and HTTP are the same things. They're different things and solve different problems. Here are the layers that are worth knowing:

Layer 7: Application

In a system design interview, you should just need to know about HTTP and DNS. Of course, you may occasionally need SMTP, FTP, SSH, and SOAP 1 but don't worry about them too deeply since they're usually not used much for most of the popular interview questions. But, again, unless you have prior expertise on the topics, you shouldn't be expected to know the deep details of them.

For HTTP, it's worth knowing about the common verbs and the common status code and when to use them. Don't overly index those details unless the interviewer asks for them; you need to prioritize your discussion points.

Layer 4: Transport Layer

The transport layer is the layer that is responsible for node to node communication that deals with control flow. The two most important protocols are TCP and UDP.

User Datagram Protocol (UDP)

In a UDP protocol, a datagram is sent from the sender to one to many recipients. Unlike TCP, it does not require an acknowledgment and does not guarantee the ordering of datagrams. Unlike TCP, It is also capable of broadcasting to many recipients. In a system design interview, you may consider UDP for:

1. Video streaming.
2. Voice and video chat.
3. Heartbeat to a server.

4. Broadcasting to many recipients.

UDP is preferred over TCP when you need a more performant protocol without all the overhead that TCP adds. The downside is decreased reliability, like packet loss and ordering. However, if your application can tolerate the downside, you can choose UDP for your solution in the interview.

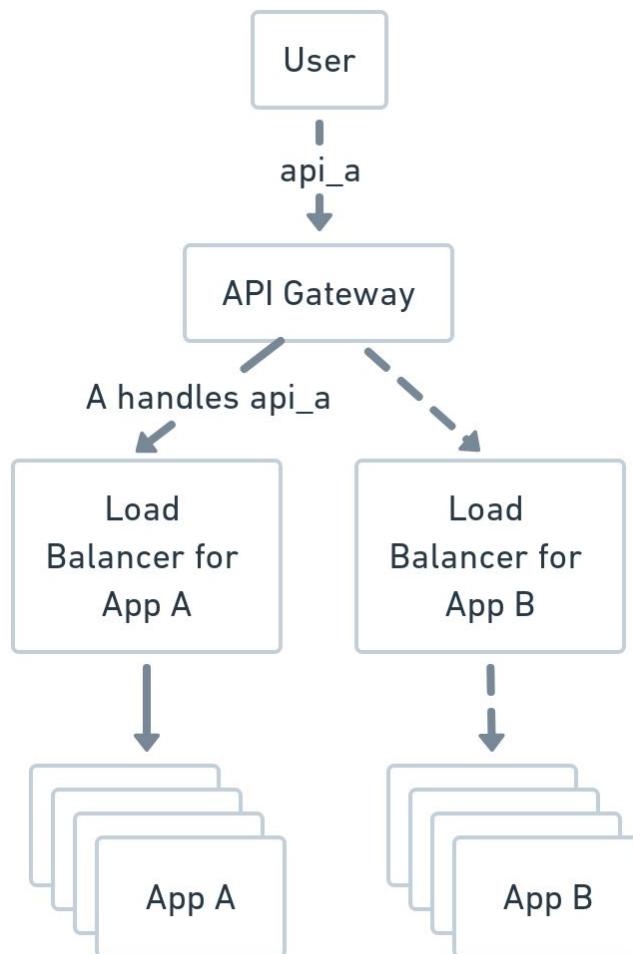
Transmission Control Protocol (TCP)

In a TCP protocol, the connection establishes sequence numbers in a TCP handshake to keep the packets in order. There will be an acknowledgment for every request, and the protocol will ensure the packets are in order. For TCP, it can only handle one on one connection. Therefore, most web applications will use TCP that can not handle packet loss. Use TCP when you need reliability to transfer data. In a system design interview, most connections will be TCP unless the use case can handle the cons of UDP.

API Gateway

API gateway is the entry point for APIs. One of its functions is to forward a given API to the appropriate microservice to process the API. Since API gateway is the entry point, it has an IP where the callers hit. The API gateway is also the main reverse proxy behind most of the internal microservices.

In an interview, if you have multiple microservices, it might be helpful to have an API gateway diagram to show clarity. Since API gateway is the main reverse proxy, if the interviewer asks you to implement a feature that fronts all the backends like rate limiting, IP blocklist, and TLS termination, API gateway is a possible place to implement them.

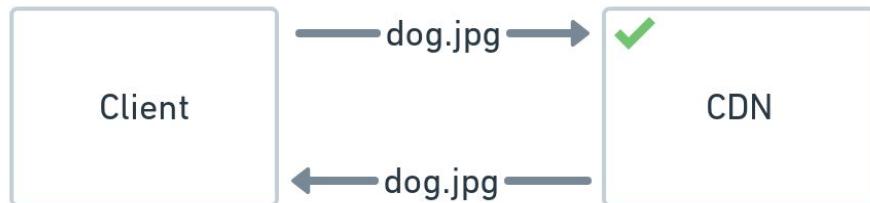


Content Delivery Network

A content delivery network (CDN) is a group of geographically distributed nodes which deliver content efficiently. You can cache static contents such as images, videos, and static files. The access pattern is similar to a read-through cache. The request for the static files will be routed to the most optimal CDN using anycast. If the file doesn't exist, it will be requested from the server and stored in the CDN.



If the content already exists in the CDN, the CDN will return the file to the client.



Some of the reasons to use a CDN are to:

Improve Latency

Because the content is closer to the user, the architecture will reduce the latency.

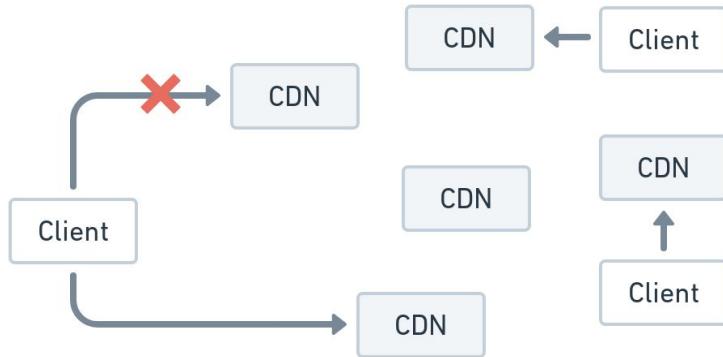
Reduce Bandwidth

Because a CDN now serves the contents, it significantly reduces the amount of data that goes through the main servers and routers. Reducing the bandwidth to the origin server can be cost-saving because there's less distance to transfer the bytes to the client.

Better Availability with Redundancy

You can distribute a CDN in multiple geographical locations with multiple redundancies. If a request fails to one of the CDNs, you can route to another

CDN.



CDN Considerations

While a CDN offers great benefits, it can also be quite complicated. When you need to deep dive into CDNs during the interview, don't trivialize them. Bring up some potential challenges and how they relate to the question you're solving. Here are some things that are worth considering:

1. We are caching static data in a CDN, and CDNs cost money. What should we store?
2. What happens if there's a CDN cache miss? How should the request be routed?
3. Since there are many CDN nodes, how do you keep the network updated?
4. How do you prepopulate the static content in a CDN?
5. If a CDN node fails, what should happen?
6. Does data differ between different regional CDNs?
7. Since CDNs are like a cache, what are the cache considerations we should think about?

These are some questions you should consider to show more depth to CDNs on top of just "CDNs are closer to the user, so it's great." The answer to the questions heavily depends on your application. For example, Netflix tries to predict what a region may watch and warm up the regional CDN, how often do they run to warm up the CDN? It opens up another layer of complexity that may be interesting to talk about in an interview. By coming up with essential discussion points, it shows the interviewer you have depth in your knowledge.

Monitoring

The system can predict and auto-adjust quickly to the changing requirements in the ideal world without any intervention. However, the reality is never the same as the ideal. Here are some metrics you should consider:

Latency

Latency might be what the end-user experiences, and it's important to check if the experience has been deteriorating. If the latency spikes, you want to check if there's a spike in downstream dependencies. If not, you should figure out if the service in question has issues.

QPS

Your user base can be growing, so it's useful to monitor QPS to ensure you have enough capacity to handle future traffic. Also, it's important to check if rogue internal systems are making multiple inefficient queries.

Error Rate

When engineers deploy new features, you want to monitor specific or overall error rates to ensure the system is reliable. If there's a spike in the error rate, on-call will need to look into the issue. In a system design interview, if time allows, you might want to articulate the important metrics you should track.

Storage

Similar to QPS, as users grow, the system will store the data at a faster rate. Throughout time, more and more data will just grow. It's essential to know the forecast so databases don't run out of space.

Metrics Count

In a system design interview, it's important to specify what kind of metrics you will be looking out for in a system design interview. You want to monitor the events you expect to happen. For example, if you have a cron job, you should monitor that the cron job has run, then trigger an alarm if it doesn't after some time. For e-commerce, If there's a sudden drop in orders

processed, it's worth looking into the reason. Another example is a ridesharing application. You want to monitor the backlog of the request queue. If the queue is getting full, that means a lot of customers are waiting for rides.

Full-Text Search

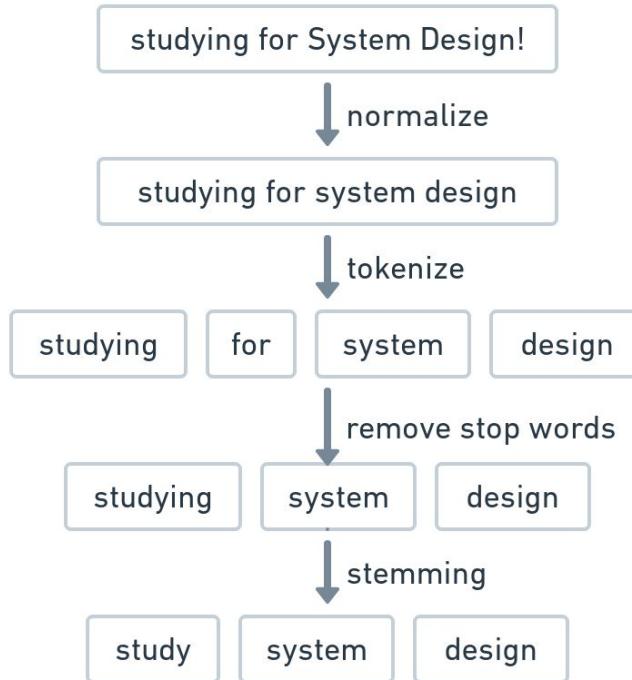
Full-Text Search is a pretty popular feature that is relevant to a lot of system design questions. For example, searching for images with labels, tweets with text, and chat history with messages. Sometimes, the search function may be a full interview question in itself.

As far as search infrastructure, it heavily depends on the requirements. Based on freshness, you may decide to index every day or every minute. Based on the search latency, you may decide to cache in favor of latency over complexity.

Search is a very broad area, and there could be specific natural language processing for specific languages, but it's useful to know some of the common steps. Here we will cover the common techniques which should be sufficient for interviews, unless you specialize in that field.

Text to Token Path

Before a text document can be indexed, we need to tokenize the string.



Normalize Step

Change uppercase to lowercase and remove special characters.

Tokenize Step

Split the full text into tokens.

Remove Stop Words Step

Words like “the” and “for” exist in all documents, so they’re not particularly useful.

Stemming

Converting and reducing words into their word stem, like “studying” to “study,” for example.

Reminder

Those are the common steps for a full-text document and are very NLP-specific. Depending on your interview question, you may need to add or remove some steps. For example, you may care about the case sensitivity of the non-stem words. The goal here is to have some high-level intuitions

so you can ask more informed domain-specific questions initially, but don't feel like you must memorize them or spend more time on NLP unless you're interviewing for that domain. For example, stemming is specific to English and not to other languages.

Indexing

For each token, the database will store a list of documents called a posting. The token to the posting list is also known as the reverse index.



With the reverse index, the search query can efficiently query for the tokens. If the token is “design,” you’re able to get document 1 and 3 efficiently. The posting list is sorted by specific attributes such as an ID or ranking score.

N-Gram

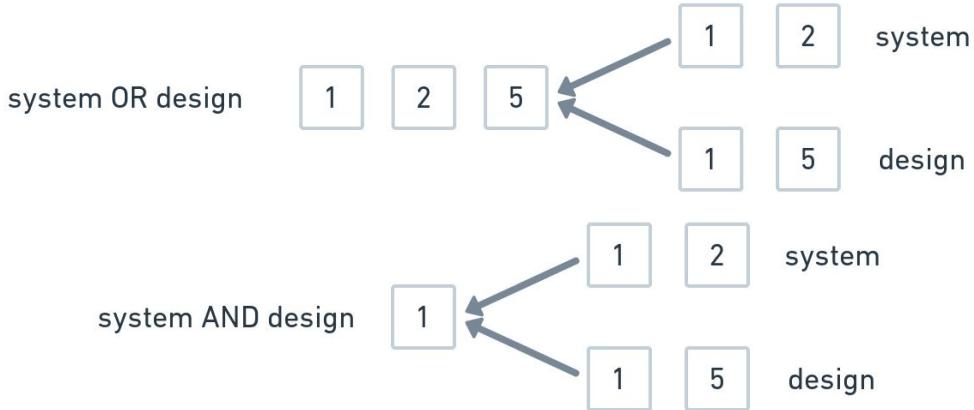
Sometimes we may be interested in querying adjacent words since they have meaning together, such as “system design” and “San Francisco.” Here’s an example of 2-gram:



When the user searches for “system design,” we’re able to get document 1 efficiently. The trade-off is extra storage since there are more keys to store.

Search Query

During the interview, it's worth discussing what kind of operations you would like to support. For example, "system design" might mean "system design," "system AND design," "system OR design," wildcard, NOT, etc. Therefore, you want to ask the interviewer what kind of query expressions you want to support.



Like the examples above, you can merge documents using the merge k list algorithm since the posting lists are sorted already.

Ranking

Sometimes, the system can return many results to the user, and it's worth discussing what kind of ranking algorithm you would like to use. Since the posting list is sorted, that sorting key can be the ranking factor. Otherwise, you may need to sort it post-retrieval. If you have a ranking score that changes through time, you need to think about what happens to the ranking changes as time passes. A famous TF-IDF algorithm is often used for document search, as discussed in the Appendix under Advanced Concepts.

Data Source and Indexing Pipeline

In an interview, it's important to talk about the data sources that you will be using to index. It's important to know how frequently the data comes in, what the structure is like, and where it's stored. After identifying the sources, you want to discuss the freshness of the index to determine if you need a stream processing, batch, or lambda architecture, as discussed in the asynchronous processing chapter. For batch processing, the MapReduce programming model is a classic way to build reverse indices.

Service Discovery and Request Routing

Within an internal distributed system, you will sometimes need to determine which node to route a request to. There are two aspects to think about:

Load Balancing

Use a load balancer when you need to distribute the load to a set of servers. There are various load balancing algorithms. Either you can round robin or monitor the hosts for health (CPU, memory, bandwidth) and forward it to the most underutilized one. Unless the load balancer significantly impacts your end design you don't need to spend too much time here, since it's generic to all questions.

Shard Discovery

If you shard your database, you will need to know which shard to route to. Shard is usually a logical concept since there can be many nodes within a shard.

A shard discovery tells you which shard a request belongs to and the shard node to forward the request to. Sometimes it can return multiple just in case one is down.

```
get_node(request_attribute) → (node_id, private_ip)  
get_node(request_attribute) → (node_id, [private_ip])
```

For example, if you shard by user_id, you can pass the user_id, and the shard discovery service will find the shard node to forward to.

Usually, the mapping is stored in a data store like a ZooKeeper. For example, assume the key goes from 1 to 100, and there are 3 shards and 3 physical nodes.

Key	Shard	Nodes
1-33	1	(node_1, ip_1), (node_2, ip_2)

34-66	2	(node_2, ip_2), (node_3, ip_3)
67-100	3	(node_1, ip_1), (node_3, ip_3)

Given this store, there are 2 reasonable approaches:

Option 1: Clients Call A Central Service

For a request, the client will hit a partition-aware service like a ZooKeeper itself. The advantage is simplicity since you only have a central service that manages the mapping. The downside could be additional latency to make an interprocess call from the client to a separate service. Also, having a central service makes it difficult to scale since all requests will need to hit the service.

Option 2: Client is Node Aware

Instead of calling a central server every time, each client can maintain the node it needs to send to. It does this by requesting for the node when the service starts up. The advantage is improved latency since you don't need to call the shard discovery service every time, and the disadvantage is the complexity of updating the client mapping every time the configuration is changed. Also, the consistency may suffer if the push to the client upon config update is delayed.

In a system design interview, this may be an interesting discussion if your system is latency-sensitive. You can consider option 2 to improve that latency at the cost of occasionally hitting the wrong node if the client mapping isn't successfully updated.

Product Solutions

Sometimes we tend to be too technical solution-focused and forget that you can solve some problems by taking a step back and thinking about other product options. In a system design interview, not all solutions must be purely technical. It's worth thinking about the fundamental user problem and coming up with alternative solutions, including product design.

Location with Poor Reception

For example, imagine you're designing a ridesharing service and the interviewer asks you how you would deal with drivers in areas with poor reception. An engineering focused solution would be to focus on getting the driver's heartbeat to remove the driver from the system as if they're offline. This isn't wrong, but it lacks user empathy.

Instead, you should be empathetic towards the driver user, as they could become very frustrated if they accidentally go into an area with poor reception. In this case, in addition to the heartbeat, you can think of product features such as letting the drivers know which geo regions tend to have poor reception, to advise them not to go there if they're waiting for a ride.

Complicated User Experience

Using the ridesharing service example again, let's say you initially design the requirement such that for a given ride request, you will fan-out to 3 different drivers to allow them to accept the ride. Letting the driver accept the ride brings up all sorts of edge cases where some drivers' acceptance will fail because another driver accepted before. Also, it locks up 3 drivers for every request, and you might deplete the pool of drivers.

Instead of trying to get it to work, you might take a step back to redesign it so the backend will automatically pick a driver, and the driver must take the ride. If the driver doesn't want it for whatever reason, they can cancel and turn off the app. While the solution isn't directly technical, you considered the technical complexity and decided on another user experience that is equally good, if not better, with a much simpler technical design. User

empathy and product sense become even more important as you become more senior.

Chapter 6:

System Design Questions

This section will go over some of the most popular system design interview questions asked by top companies. What's important here is not to memorize the solution, but to look at some examples of how to impress the interviewer by associating the proposed options and final recommendation to the assumptions and requirements.

There are many different ways to design a system design question, and no sources, including this book, should be the authoritative source. Even though there will be many similar patterns, what's important is your ability to convince the interviewer by providing reasonable designs and trade-offs.

As you practice coming up with your solutions during mock interviews, try to think critically about the solutions and run them by people with experience to gather feedback. You will learn better this way.

There will be more deep dive discussion points in the examples than you can cover in a real interview, so don't worry about covering everything. Usually, a good 2 or 3 solid discussion points are good enough in an interview. More discussion points are covered here, to help you learn through examples.

Two people can get an offer from top companies even if they have completely disagreeing designs, because their assumptions may be different. For example, suppose you self-doubt your design often during self-study because other sources have different solutions. In that case, you will struggle in an interview when the interviewer challenges you with another potential solution if you are shaky with the fundamentals. Learn the fundamentals, and you will be flexible in solving any question!

Ridesharing Service

Interviewer:

Please design a ridesharing service like Uber and Lyft.

Step 1: Gather Requirements

Functional Requirement

Candidate:

Do you mean riders can request rides by ridesharing service, and a driver will eventually pick them up, correct?

Interviewer:

That's correct.

Candidate:

Ok, so the purpose of the product is to solve the pain point of going from one location to another quickly and at a reasonable price. Here are the features I can think of:

1. Match riders with drivers.
2. Display an estimated cost for the rider.
3. When the user signs in, they see a list of nearby drivers.
4. Support multiple rider users into a single car like Uber Pool.
5. Drivers and riders can cancel rides.

Is there anything specific or any other requirements you would like me to focus on?

Interviewer:

In the interest of time, let's just focus on matching the riders with the drivers. We can worry about other features if we have time.

Non-Functional Requirement

Candidate:

Ok great. Let's talk about some of the non-functional requirements. How many users are we working with, and how are they distributed across the world?

Interviewer:

Let's assume we have 100 million daily active users (DAU) scattered around the world. Most users are in North America and Europe.

Candidate:

Ok thanks, are there any query patterns I would be aware of? Should I worry about potential bursty and cyclical requests? For example, I expect more users to be using the service on a Friday night or after a major concert or sporting event.

Interviewer:

That's a great question. Those are traffic patterns we should keep in mind.

Candidate:

I am going to assume that we need the drivers' location to match the riders and drivers. I'm going to assume we optimize for the linear distance. Clearly, there are many edge cases in the real world, like if there's a river between the driver and the rider. But in the interest of time for this session, that's the assumption I will be making. Are you ok with that assumption?

Interviewer:

That's fine with me.

Candidate:

I will assume the wait time for a ride should be minimized. However, it is acceptable if the rider waits for 30 seconds or more.

Interviewer:

That sounds reasonable to me, it's acceptable to have an occasional long tail of waiting time if we have driver supply problems, but for the majority of the cases, we should strive to minimize if driver supply isn't an issue.

Candidate:

Ok great, how important is the accuracy of the driver location data? I'm going to assume it's important within the order of 10 seconds delay since the distance traveled in 10 seconds won't have a critical impact on the final result. Worst case scenario, the user waits ten more seconds. Also, I'm going to assume availability to request rides is very important here because it's frustrating not to be able to get a ride.

Interviewer:

Both driver location accuracy and availability sound reasonable.

Step 2: Define API

Candidate:

Ok, now that we have the requirements, let me define the APIs to make sure we are on the same page:

Rider API

```
request_ride(user_id, from_location, to_location) → status  
matched_ride(driver_info)
```

Driver API

```
update_location(user_id, current_location) → status
```

Location is defined as a tuple of longitude and latitude. For the `request_ride` API, a success status response means the server has taken the request and is in the process of finding a ride.

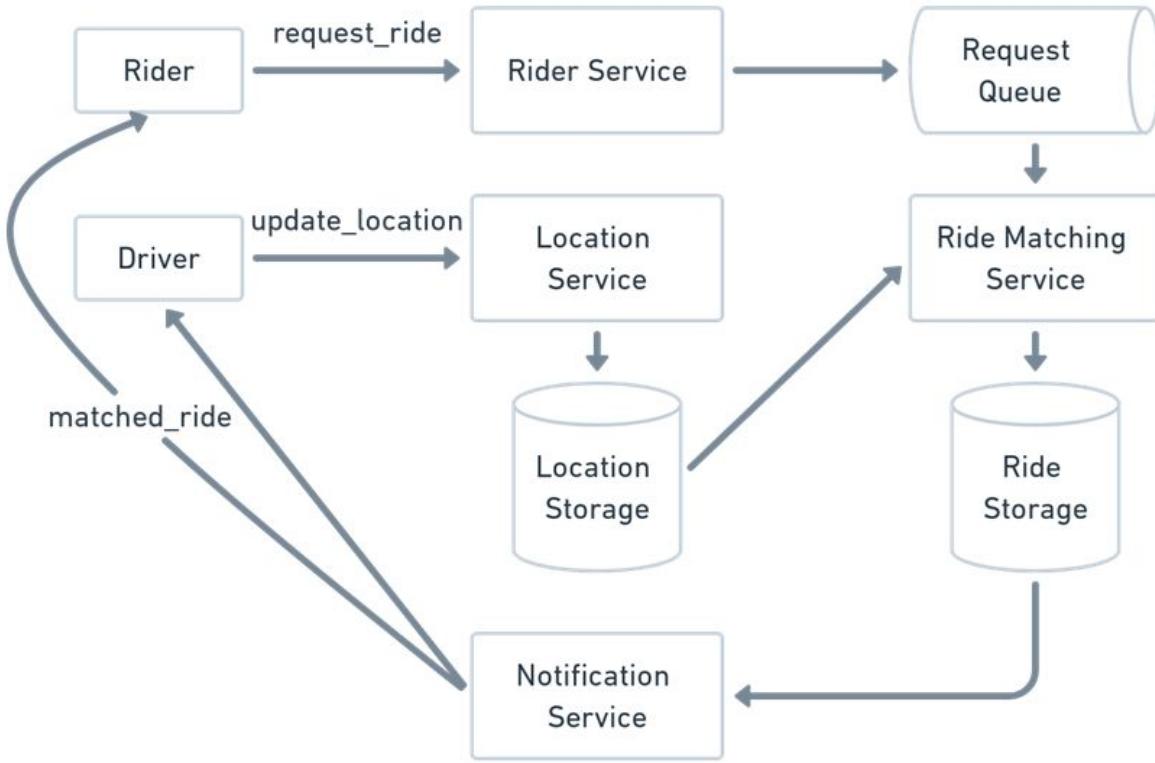
`matched_ride` is a callback function when the server finds a ride and notifies the client of the `driver_info`.

`update_location` is the driver periodically sending their location to the server. The frequency will be an interesting one to talk about.

Step 3: Define High-Level Diagram

Candidate:

Now that I've defined the core end-user APIs needed for the requirement, let me draw the high-level diagram to fulfill the APIs. I'm going to assume we need some sort of queue to handle the bursty traffic, but we can come back to this.



Step 4: Schema and Data Structures

Candidate:

So here is my high-level diagram. It looks like there are two storages and a queue. Let me quickly go over the logical schema for those.

Request Queue

For the request queue, each event can have the `rider_id` and the `from_location` to pick up the rider.

Rider Id	From Location
----------	---------------

Ride Table

Once a rider is matched with a driver, a record is persisted into the ride table. Each rider and driver can only have one active ride.

Ride Id	Rider Id	Driver Id	From Location	To Location	Status
---------	----------	-----------	---------------	-------------	--------

Location Storage

For each driver, a current location is stored. However, searching for a list of drivers for a requester's location will be slow with a full table scan. Let me come back to this optimization.

Driver Id	Current Location
-----------	------------------

Step 5: Summarize End to End Flow

Candidate:

Now that I have the API, high-level diagram, and schema, let me walk through the API flows to ensure I didn't miss anything.

For request_ride, the rider will hit the rider service first and immediately forward that request to the request queue. The ride matching service will periodically pull from the request queue and ride from the location storage for drivers to match with. Once a rider is matched with a driver, the ride record gets persisted into the ride storage. The matched ride event gets broadcasted out through the notification service to the driver and rider.

For update_location, the driver periodically sends in location information to the location service, and the location data gets persisted into the location storage. The frequency is a concern to be, I can deep dive into this later.

I think I have a working solution. I have a list of topics I want to talk about, but before we go deeper, do you have any questions regarding my current design?

Interviewer:

This seems reasonable. Let's proceed with what you think is important to address.

Step 6: Deep Dives

Candidate:

Ok, thanks, so here is a list of topics that I think are worthy of addressing:

1. Driver location update
2. Location update failure scenario
3. Location storage search
4. Concurrency: What happens if multiple concurrent requests get the same list of drivers from the location storage?
5. Bursty request after some popular concert or sporting events.

I think these are the critical ones. Is there one that you would like me to cover or any other ones not listed here?

Interviewer:

The list looks reasonable. Go ahead with whichever one you think is good to talk about.

Driver Location Update

Candidate:

Ok, let's talk about driver location updates. In the beginning, we are assuming a driver location update every 10 seconds. Let's do some quick math to figure out the scale. Let's assume we have 10% of the DAU for drivers, which is 10 million. Each driver updates every 10 seconds, so that's 1,000,000 QPS. If we write directly to disk or SSD, it will break, so we need to shard up to 100-2000+ instances, assuming a database happens 500 QPS and SSD handles 10,000 QPS.

Option 1: Put a Queue in Front of Database

To deal with the high write QPS, we can put a queue in front of the database since queues can handle millions of QPS. We can do micro-batches to increase QPS into the database. The upside of this is the durability of the location data at the cost of many more machines and complexity with an additional queue.

Option 2: Write to Location Cache

Instead of writing to a database, we can write to a location cache. In the best case, a cache can handle 50,000 QPS+, which doesn't require as many instances as disk and SSD. The latency is faster for a location update. The downside is the durability of the location will suffer if the cache goes down.

Option 3: Lower the Update Frequency

Every 10 seconds is nice, but making it every 20 seconds wouldn't be the end of the world. In the worst case, the potentially matched driver is driving away from the rider. It would just take 10 more seconds to come back.

Conclusion

Since the driver location is updated every 10 seconds, even if we lose an update, it will be fine because the next update will make the lost location obsolete. So we can go option 2 with a non-durable store in favor of better throughput. We can also choose number 3 at the same time and monitor the quality of matched rides since we're decreasing the update frequency. We don't really need a queue since we expect the cache cluster to handle the throughput already.

Interviewer:

Ok, I think that makes sense, but can you go into more details about your cache architecture and how that relates to failure scenarios?

Location Storage Failure Scenario

Candidate:

Yeah, that's a good point. Let's discuss what happens if the cache goes down.

Option 1: Leader-Follower Replication

If the leader goes down, we will have to pick another follower to become the leader of the write requests. The read from the matching engine can be done from any read replicas since location accuracy is a strong constraint.

Option 2: Leaderless Replication

We will perform a quorum write and read from a cluster of nodes using leaderless replication. If one of the nodes goes down, we can continue to do

quorum write and read.

Conclusion

Since option 1 requires an election to pick the newest leader, the location service will be temporarily unavailable for a location update. For option 2, the advantage is that if any of the nodes fails, the write and read can continue at the expense of an outdated read since the replica with the latest update may be down. Since the consistency of driver location isn't critically important, we favor availability over consistency for driver location use cases to ensure riders can always request rides and drivers can always update their locations.

Interviewer:

Ok, that sounds reasonable. Can you go into more detail about the match-making service?

Location Storage Search

Candidate:

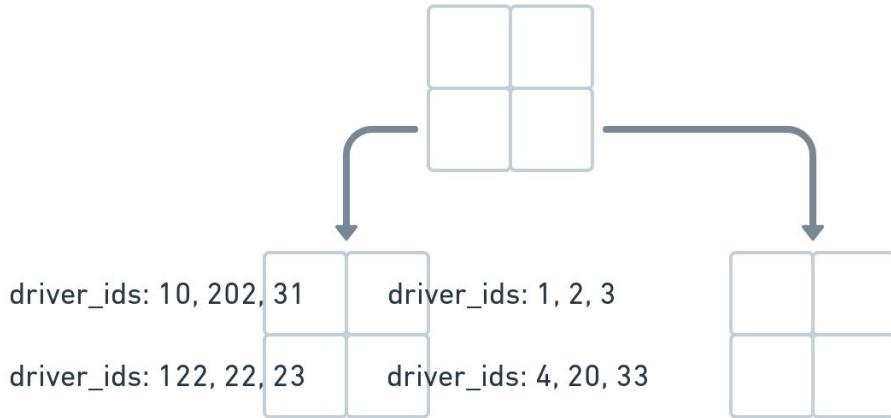
When the matching engine needs to pick a driver for a rider, it needs to read the location storage for potential candidates for a given requested location.

Option 1: Conduct a Full Table Scan

For a given requested location, scan through each driver's location and shorten it by the distance. Then, grab the driver with the shortest distance.

Option 2: Index by Location ID / QuadTree

Create an in-memory database quadtree to scope down the number of prospective drivers efficiently. Each quadtree box will have a list of driver IDs. When a driver pushes a new data, the driver table is updated, and if the quadtree node changed for the driver, it will be removed from the previous node first before inserting into a new node.



Conclusion

Clearly, we need some sort of indexing to narrow down on the search space, so I would pick option 2. However, in real life, there are specialized data structures like Google S2. For now, we'll keep it simple with a quadtree. Are you ok with that?

Interviewer:

Ok, that sounds fine to me.

Match Making Concurrency

Candidate:

Another topic I want to bring up is concurrency. As you can see from the high-level diagram, multiple concurrent requests happen simultaneously to fetch potentially the same list of drivers. For example, if you and I are standing next to each other and requesting a ride simultaneously, the same driver should be the “best” driver for both of us, but only one of us can get that driver.

Option 1: Serially

The ride matching service can process one request at a time. The issue is the overheard reading from the request queue, location storage IO, and ride storage IO will all add up.

Option 2: Serially Batch

Instead of matching one request at a time, we can dispatch a batch of ride requests against a batch of drivers and resolve in a batch of rider to driver

matching. This has a better throughput than option 1 and solves the concurrency issue.

Option 3: Pessimistic Locking

Whenever a request comes, you can lock up a quadtree quadrant and let all the other requests wait. The throughput issue is just as bad as option 1 since each request needs the lock to be released for a given location.

Option 4: Optimistic Locking

Whenever a driver is selected for a rider, do a conditional commit to the rider table and if the driver is already taken, fail this request. The failure bubbles up to the end-user such that the end-user has to retry. It's a poor user experience to have to keep retrying in a highly concurrent environment. Even if the internal system automatically retries, there could be a lot of retries in a highly concurrent environment, leading to a high waiting time. The advantage is no requests are blocking each other.

Conclusion

Option 1 and 3 are out of the question since the throughput just doesn't cut it. Option 4 is a possibility but may lead to a poor user experience with a lot of retries and waiting time. Option 2 seems the most reasonable with a dispatcher system, we need to tune the batch to ensure it can handle the throughput and the workers can handle the computation of a batch of riders and drivers.

Interviewer:

Ok, that sounds fine to me. At the beginning of the interview, you mentioned bursty requests. How would you handle those?

Bursty Requests

Candidate:

Let's do some calculations on the request_ride QPS. 100 million DAU, assume each active user requests 2 rides a day, and we will assume a multiple of 10 during peak time.

$$10^8 \times 2 \times 10^1 = 2 \times 10^9 \text{ QPD}$$

$$2 \times 10^9 / 10^5 = 2 \times 10^4 \text{ QPS}$$

20,000 QPS for a complex calculation like ride matching isn't trivial.

First of all, I've already proposed using a queue that can handle the high write request. I just need to make sure the queue doesn't get backlogged, so I need to monitor for queue spillover. Secondly, I addressed the issue of concurrency in a highly concurrent environment. However, processing the matching serially with batching still doesn't seem achievable with 20,000 QPS. Therefore, I am going to consider sharding to share-nothing shards so each shard can have its dispatching.

Option 1: Shard Based on Locality

Let's say we divide the world up into 4 parent shards. Within each parent shard, there will be sub-shards where each sub-shard is processed serially. The purpose of the parent shard is so they live on the same machine. The purpose of the subshard is to have parallelization of dispatching.

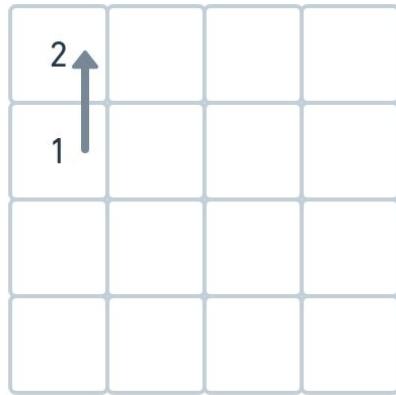
1	1	2	2
1	1	2	2
3	3	4	4
3	3	4	4

The advantage of this sharding is that if a shard needs to talk to another shard, it is more likely they're already on the same host and doesn't request a machine hop. For example, if there are no cars in one of the sub shards, the sub shard can forward that request to another adjacent sub shard. However, the downside is potential hotspots. For example, if all the 1s are San Francisco, shard 1 will be hot.

Option 2: Random Shard

Instead of having the notion of parent shard and sub shard, make everything into sub shards. If there are no drivers in a sub shard, forward that request

into an adjacent sub shard that may live on a separate machine.



Conclusion

Both options allow us to sub-shard to increase the throughput. Although it's nice to have option 1 where machines are more likely to be next to each other, given latency isn't a huge issue and the probability of needing to reach into another sub shard is low due to low driver supply, I would go with option 2 to avoid the risk of a big hotspot.

Interviewer:

Ok, I see where you're going with this. Unfortunately, we are out of time.
Thank you for your time!

Top Watched YouTube Video

Interviewer:

Please design a system to calculate the top YouTube videos.

Step 1: Gather Requirements

Functional Requirement

Candidate:

Who are the users, and how will they be consuming the output? What is the pain point we're trying to solve for them? Also, what kind of data sources do we have to build the top videos? And is the top attribute based on the video itself or the metadata of the video?

Interviewer:

Those are good questions. For now, we'll only consider the video itself, don't worry about video categories. The value-add for top video is to allow the user to discover something interesting to watch now. You just need to design a global top list to be displayed as part of their web or mobile client.

Candidate:

That sounds good. Should I provide a different top list for different time windows, and what does the UI look like? Is there a specific algorithm I should use to determine the top?

Interviewer:

As far as the UI is concerned, there's a tab called top video. There are sub-tabs for each time window of interest within the tab, like daily, last hour, and real-time. I will leave it up to you for the algorithm. It will look roughly like this:

YouTube		
Real-Time	Hourly	Daily
1. System Design Interview Fundamentals 2. Superbowl Playoff Replay 3. Gangnam Style 4. Let's Get in FANG 5. COVID		

Candidate:

Ok, for now, I will assume a video with the greatest count within a time window determines the winner.

Non-Functional Requirement

Candidate:

How many users should I assume for YouTube, and how are they distributed across the world? Is there a distribution of videos that users tend to watch? How many unique videos are there?

Interviewer:

Let's assume there are 50 million DAU across the whole world. For the videos, it's a long tail distribution where a subset of videos makes up the majority of the volume. Let's assume there are 10 billion unique videos on YouTube.

Candidate:

How important is the accuracy of the output? What if it is slightly incorrect? Also, what is the freshness we are hoping to achieve? Does it have to be sub-seconds, or is waiting for a couple of minutes acceptable?

Interviewer:

Accuracy and freshness are both important because we want the users to get the most accurate and latest data, however, feel free to make some trade-offs if you need to.

Candidate:

Ok, that sounds good, and I'm going to assume the query to fetch for top videos should be as low as possible, and the data should be durable to be fetched by all the users in the future. Furthermore, I'm going to assume the data source is from the events omitted directly from the watch event instead of from a previously generated log. Is that fine?

Interviewer:

Sounds good to me.

Step 2: Define API

Candidate:

Ok, based on the requirements, here are the APIs I think I will need:

`watch_video(user_id, video_id) → status`
`get_top_videos(k, start_time, end_time) → [video_info]`

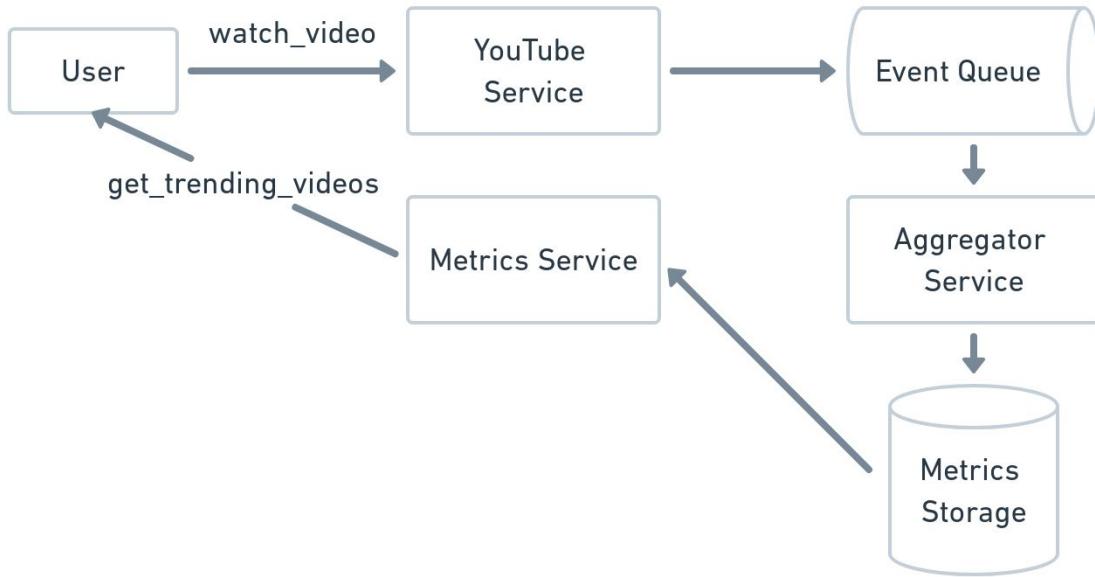
For `watch_video` the API is called when the user watches a video. For now, we won't worry about things like deduping the count if they refresh the page.

For `get_top_video`, this is an API to fetch the top K videos for a time range. For now, I'll assume K can only go up to 100, and the time is in minute level granularity. The output is a `video_info` object with `video_id` and some metadata to be displayed.

Step 3: Define High-Level Diagram

Candidate:

I'm going to draw the diagram for the write and read path. I'm going to assume we need a queue since the aggregation pipeline is asynchronous by nature. The aggregator service can be a pass-through and store every event, but that might be too much data and will be compute-intensive on read. So instead, we will take events from the event queue and roll them up. The aggregation algorithm and size of the rollup could be an interesting discussion for later.



Step 4: Schema and Data Structures

Candidate:

So here is my high-level diagram. It looks like there is one storage and a queue. So let me quickly go over the logical schema for those.

Event Queue

Whenever a user watches a video, we omit an event of `video_id` and the timestamp associated with it. So let me come back and think about if it makes more sense to be a client or server-generated timestamp.

Video Id	Timestamp
----------	-----------

Metrics Storage

For now, I will assume we roll the metrics up to a minute granularity by `video_id` and create a minute metrics table. If we need other more or less granular rollups to optimize for other reads, we can further discuss the need and potential schema.

Minute	Video ID	Count
--------	----------	-------

To get the top K for a given minute, we need to fetch all the minute records and sort them by count, which will be expensive, and that's something we can discuss later.

Step 5: Summarize End to End Flow

Now that we have the high-level diagram and schemas, let me walk through the flow to make sure I have all the components.

For `watch_video`, when a user watches a video, it will call the YouTube service, and the service will emit an event to the event queue. Because there might be many metrics and the flow is asynchronous, we added a queue. Storing every event will be data-intensive and slow to aggregate on read. We can try to materialize by rolling up to the metrics storage.

For `get_top_videos`, it will call the metrics service, which is just a light layer to call the metrics storage for the top videos.

I have a couple of things I would like to discuss further, but am I missing anything that you would like to cover?

Interviewer:

No, you covered the main critical paths, but I agree. It sounds like there are a couple of places for further discussions.

Step 6: Deep Dives

Candidate:

Great, so here are some that I've gathered:

1. Client versus server-side generated timestamp.
2. Even with a rollup, the read might be slow since there may be many unique videos.
3. Size of rollup.
4. More detail about the aggregation algorithm.
5. Address any scalability issues for the queue, aggregator, and database.

Generally, these are the discussion points I would like to cover. Is there anything else you would like me to address?

Interviewer:

That sounds like a reasonable list. Please go ahead.

Client versus Server Timestamp

When we aggregate data, we need a timestamp. The timestamp can be generated from multiple places.

Option 1: Client Generated Timestamp

The web or mobile client will generate the event time and pass it to the server for processing. The advantage of client-generated timestamps is the accuracy of the event time since the client generates it. If the device is offline, it will still capture the event. However, the risks for client-side timestamp are client device clock skew and maliciously generated timestamps.

Option 2: YouTube Service Generated Timestamp

Instead of generating it on the client, we can generate it on the server-side. The issue is if there's a network delay or the client is watching the video offline, the timestamp won't be as accurate. However, there's more consistency with the clock with a server-generated timestamp.

Conclusion

Since we're not supporting the offline use case, the timestamp accuracy doesn't need to be 100%. Furthermore, we don't expect the metric collection to be significantly backlogged such that the client-side events are delayed. Let's go with option 2 for now because of those reasons. In the future, we can capture client event time, client event sent time, and server received time. You can use the difference between client sent and server received time to approximate the client clock skew and adjust for the delta on the server. For malicious timestamps, we can monitor for outlier timestamps.

Aggregation Service Data Structure

Candidate:

Now, let's dig into the aggregator service. The aggregator service will periodically fetch for a batch of events and roll them up into minute segments. The logical schema that is stored in-memory is:

Minute	Video ID	Count
--------	----------	-------

After a minute segment is ready to be flushed to disk, we need to fetch the top K videos.

Option 1: Sort the List

Sorting the list will result in $N \log N$.

Option 2: Use a Min Heap of Size K

Using Min Heap will result in $K \log K$.

Conclusion

Given K is significantly smaller than N, let's go with option 2.

Aggregation Service Scaling Deep Dive

Candidate:

Let's do some math to figure out potential bottlenecks. Let's assume there are 100 million unique videos expected per minute, and we anticipate to buffer for 20 minutes since sometimes stream processing can go down, and there may be a backlog to build. Assume each column is 8 bytes for a total of approximately 30 bytes with 3 columns: minute, video_id, and count.

$$1 \times 10^8 \times 30 \times 20 = 6 \times 10^{10}$$

Which is 60 GB of memory conservatively, which should fit into a modern machine in-memory just fine.

Interviewer:

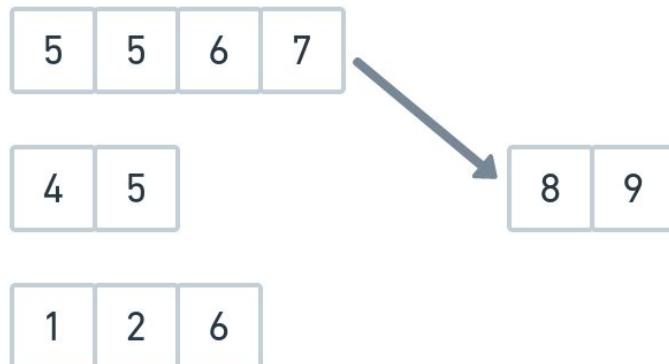
Ok, that seems reasonable. But for now, let's assume we only have a 16 GB machine and plan for more than 100 million unique videos. What would you do differently?

Candidate:

Ok, basically the problem is it won't fit into memory. So let me think of a couple of options.

Option 1: Shard into Multiple Aggregator Services

Each aggregator service shard will maintain a mutually exclusive count table. We can shard by hash using `video_id` using consistent hashing since we don't need to perform a range query. Once a minute window is ready to be materialized, apply merge K sorted list algorithm:



The downside is the complexity of dealing with a cluster of computing nodes that need to be coordinated. For example, you need another layer to coordinate the flushing of the same window across all the shards. Also, the additional overhead will result in additional latency as well. And since there's a long tail distribution, some videos will likely be hot and lead to hotspots.

Option 2: Randomly Insert into Aggregator Service

Instead of looking at `video_id` and deciding on the shard, just randomly insert events into an aggregator. After aggregation per shard, there will be a coordinator to merge the results. This has the benefit of random distribution, so it won't lead to hotspots. The downside is you need to scatter-gather a `video_id` count by searching all the nodes.

Option 3: Use a Probabilistic Data Structure

You can use a probabilistic data structure like Count-Min Sketch where the trade-off is more space for a more accurate count. Although the Count-Min

Sketch table stores the approximate count, you will still have to go through the videos and create a min-heap to get the top K.

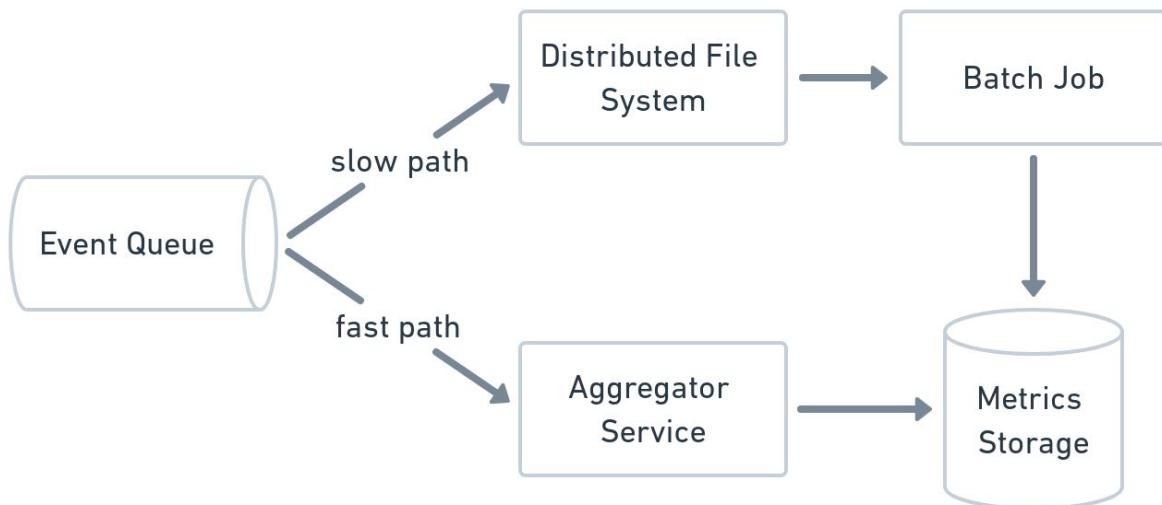
Warning

Advanced data structures like Count-Min Sketch aren't always required for system design interviews. It would be unreasonable for interviewers to expect candidates to know all the advanced data structures in the respective fields. However, if you do mention it, it may give a good impression. Just make sure you know the internals very well.

Sometimes the interviewers themselves won't know about it. You need to read the room and not dig yourself a hole by spending 5 minutes explaining how Count-Min Sketch works. You can dismiss it by saying, "Count-Min Sketch will help reduce space at the expense of accuracy. However, for now, can I assume we want strong accuracy and can't fit into memory, so we need to shard with option 1 or 2?"

Option 4: Have a Batch Pipeline to Catch Up

Since option 2 will result in some data inaccuracy due to the inability to handle the storage and high throughput to calculate the count ranking of millions of videos, we can have a slower but more accurate pipeline to account for that. This is also known as lambda architecture.



We can use the MapReduce programming paradigm to achieve this. We can dig deeper later if we have time. The downside of lambda architecture is the complexity of maintaining two similar systems.

Conclusion

I would pick option 2 since we have some room to sacrifice accuracy in favor of getting the data fast. And I would consider option 3 if we need an eventually accurate result for the end-user.

Interviewer:

Yeah, that sounds reasonable to me.

Metrics Storage Deep Dive

Candidate:

Ok, now that we've talked about the aggregation service, let's talk a bit more about the storage layer and how to satisfy the hourly and daily aggregation. So far, we're only storing the top K per minute interval. This will clearly work for minute aggregation but becomes problematic if we aggregate 60 minute-intervals to calculate for an hour interval since the winner count may not exist in every minute interval. It gets even worse with daily intervals.

Option 1: Aggregate it Anyway

One obvious option is just to aggregate the 60 minute-intervals anyway. The upside is simplicity; the data is already available. The downside is data inaccuracy since each minute interview doesn't have the full count list.

Option 2: Include Some Buffer

Instead of storing the top K, store K + buffer. Assume the buffer size is 500 and K is 100. We will store the top 600 per min-interval. This is less likely to be lossy at the expense of more data to store and process. Even then, it still can be lossy for a longer time horizon.

Option 3: Keep In Memory for Hour Data in Streaming

So far, we are only processing for minute-interval. We can process it for hour-interval as well. The upside is the hour interval data will be persisted

with accuracy. However, the memory issue addressed above becomes even more problematic due to the need to store for an hour. We can again consider using Count-Min Sketch for hourly data to reduce the amount of space needed. I would create an hourly table to store the result.

Hour	Video ID	Count
------	----------	-------

Option 4: Hourly Batch Job

Instead of depending on min-interval, depend on the hourly batch job to compute for the result. The downside is batch jobs are usually less reliable to get the job done on time since there's a lot of data to be processed.

Conclusion

I would consider option 3 since it's the best effort for accuracy, yet you get the result in a more timely manner. I would still consider option 4 to override the inaccurate result from option 3, if accuracy matters.

Aggregation Service Failures

Interviewer:

Ok, I think that sounds reasonable. Can you describe to me the failure scenarios of your architecture?

Candidate:

Sure. Clearly, any part of the system can potentially fail. Generally speaking, for stateless application servers, I would put a load balancer in front of them. However, the one that worries me the most is the aggregator service. If the aggregator service goes down, I will need to build the aggregations from scratch. Therefore, I would use periodic checkpointing and rely on a durable queue to replay from the offset associated with the checkpoint.

Here are a couple of options:

Option 1: Checkpoint on Local Disk

Whenever I need to create a checkpoint, I write to the local disk, and in case there's a streaming process level failure, it rebuilds from the local disk. The

advantage is the checkpoint and reconstruction will be fast. However, in the event of a correlated instance-level failure, both disk and in-memory aggregation are lost. Also, you need to worry about if the instance has enough disk memory to hold for the long term.

Option 2: Checkpoint on a Distributed Store

Instead of writing to a local disk, write them to a distributed data store. The upside is the ability to scale and it lessens the chance of a big failure impact due to correlated failure. However, the network bandwidth overhead will slow down the checkpoint and reconstruction.

Option 3: Write to Local Disk and Distributed Store

I can write to the local disk more frequently if the in-process streaming fails while less frequently dumping the data to a distributed store in case of a correlated failure.

Conclusion

I would choose option 3, although we need to be careful with data loss if the backup is asynchronous and with backup delay. That is probably acceptable since losing some data is acceptable here. Generally, I would toy around the checkpoint frequency where more frequent checkpointing will result in a faster recovery time but slower processing time.

Late Events

Interviewer:

Ok, that sounds like a fine tradeoff. We have time for one more discussion. As you mentioned, what happens when a late event comes in?

Candidate:

Good question. I would have a watermark to indicate all events before that watermark should be treated separately. Let me think about the length of the watermark.

Option 1: Longer Watermark Delay

I can have a long watermark delay to account for events that come way late. The upside is better accuracy, but I would be holding more data in stream

processing.

Option 2: Shorter Watermark Delay

The shorter the delay, the sooner I can finalize the result, and the less data I have to hold in-memory. But if late events come, I will need to handle them separately.

Conclusion

Given data accuracy has some leeway, I would prefer option 2. Also, I don't anticipate too many late events since the assumption here is to use server-generated timestamps as the internal network is more reliable.

Post Watermark Processing

Candidate:

Also, I need to think about what to do when a late event comes.

Option 1: Discard It

If an event past watermark comes along, I will just discard it. The obvious con for this is that it is lossy. The pro is simplicity, as it is easier to just ignore it.

Option 2: Try to Modify the Existing Metrics Storage

If a late event comes in, I may insert it into another queue for modification processing. However, this comes with a lot of complexity to build another pipeline for modification.

Conclusion

Since accuracy has some leeway and we already have an option to create a slow path with batch processing, this will eventually be resolved. So we don't need to build a separate pipeline for modification.

Interviewer:

Sounds good, we are out of time. Thanks!

Emoji Broadcasting

Interviewer:

Please design emoji broadcasting for Facebook live. When people are watching a Facebook live stream, people can express their reactions using emojis. When the emoji is pressed, it should get fanned out to all the stream watchers. How would you design something like this?

Step 1: Gather Requirements

Functional Requirement

Candidate:

Can you describe to me the user experience? Can I assume the purpose of this product is to allow the users to feel the general mood of the watchers, so they feel more engaged? Is there a predefined set of emojis we're working with? Should I worry about who can watch the stream? Any special features for VIP watchers or anything like that? Should I worry about offline replays?

Interviewer:

The general experience is the emoji icon temporarily appears on each watcher's viewport and disappears. Yes, the emojis try to display a collective emotional experience to the streaming experience. Assume the emojis are predefined. For now, assume anybody can watch the stream, and everybody is treated the same. You don't need to support offline replays.

Non-Functional Requirement

Candidate:

How many daily active users are there, and what can I assume about the distribution of the user base? Should I worry about celebrity users or very popular streams? For example, what's the maximum number of people a stream can allow?

Interviewer:

Good questions. Let's design for 100 million daily active users for Facebook Live, and users are distributed worldwide. Yes, you should account for celebrity users and popular streams. You should design streams to support tens of millions of users per stream in the case of a really popular stream.

Candidate:

Sounds good. What's the accuracy requirement for this design? I would imagine in a very popular stream, a flood of hundreds of thousands of concurrent emojis would freeze up the devices, and even if the devices can handle it, it would cover up the user's viewport.

Interviewer:

That's a good observation. Feel free to play around with what you think is a good user experience in case of excessive emojis.

Candidate:

What about the freshness of the emoji? I would expect the emoji to be delivered as soon as possible because an out-of-context emoji would be a confusing user experience.

Interviewer:

I agree.

Candidate:

That brings up another requirement about consistency. If I press the emoji at a stream's specific frame, it would be challenging to display that specific time frame since other users may be ahead or behind the stream. So for now, can I assume I don't need to coordinate the emoji and just send it as fast as possible?

Interviewer:

Thanks for bringing that up. Yes you don't need to worry about it.

Candidate:

Since we don't need to support replay, and it seems like we have some flexibility for losing some emojis, I'm going to assume durability isn't important at all. But we should strive to deliver as many emojis as we reasonably can.

Interviewer:

Ideally, we want to capture all the emojis to have all the data points to design for what we should ultimately show to the end-users. However, an occasional loss is acceptable.

Step 2: Define API

Candidate:

Ok, based on the requirements, here are the APIs I think I will need:

watch_stream(user_id, stream_id) → websocket_address
send_emoji(user_id, stream_id, emoji_id) → status
display_emoji(emojis)

watch_stream is the API called when a user clicks into a stream. It will redirect the client to connect to a WebSocket instance.

connect_websocket connects the client with a WebSocket instance.

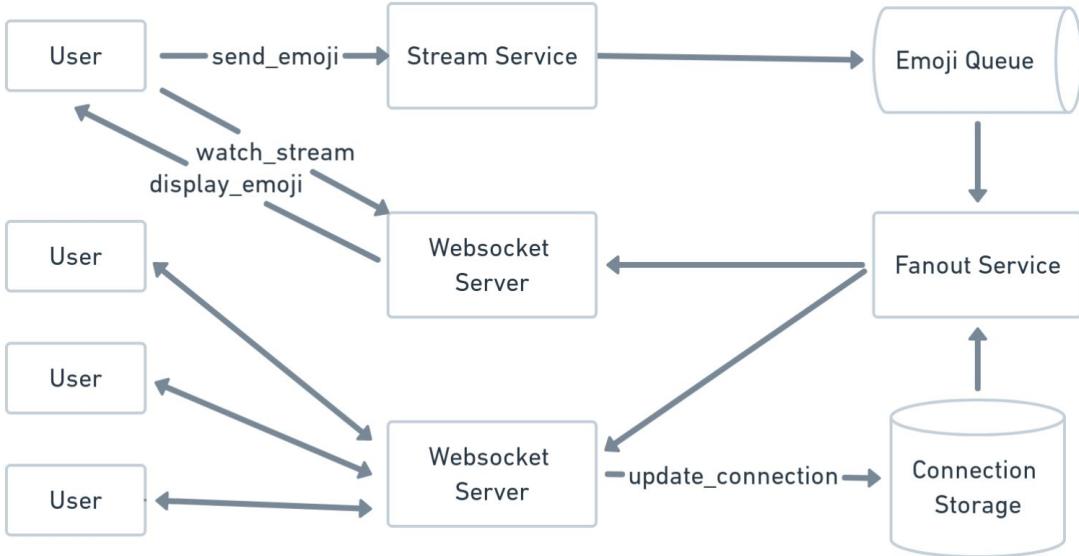
send_emoji is used when a user is in a stream, and they express an emoji. Maybe we should think about a timestamp here if the request is delayed and the emoji is outdated and should just be dropped. For now, we will push whatever the client gives us. A status success means the server has successfully received the message and is in the process of fanning the emoji out.

display_emoji is a callback function on the client-side to be displayed on the viewer's viewport.

Step 3: Define High-Level Diagram

Candidate:

Ok, let me draw the high-level diagrams to satisfy the APIs we just described. I'm going to assume we need a queue for send_emoji because that is by nature asynchronous, and the write throughput should be high. We'll validate it in the deep dive.



Users are connected to the WebSocket servers to stream the videos. For now, we'll use WebSocket unidirectionally from the server to the user. For `send_emoji`, it can go through a streaming service to be fanned out. When users are connected, it updates the connection storage.

Step 4: Schema and Data Structures

Candidate:

So that is my high-level diagram, I can add more details into the schemas for emoji queue, connection storage, and WebSocket server.

Emoji Queue

For now, we will assume we don't need to display who the emoji is from. So all we need to know is which stream ID the emoji is going to.

Stream Id	Emoji Id
-----------	----------

Connection Storage

The fan-out service needs to know which WebSocket servers to forward the emoji to based on the `stream_id`.

`stream_id` → [websocket_server_id]

WebSocket Server

Since a WebSocket connection is behind a stateful TCP connection, the lifecycle of the connection is tied to the physical machine. Each WebSocket server will maintain a mapping of a stream_id to a list of connections.

stream_id → [connection]

A connection object is basically the IP and port to forward the request to.

Step 5: Summarize End to End Flow

Candidate:

There are a couple of flows to talk about.

When the end-user calls watch_stream, it sends a notification to the WebSocket server to include the connection in stream_id → [connection] on the WebSocket server and the WebSocket server will update the connection storage to ensure the websocket_server_id is included in that stream. When the user disconnects from the WebSocket server or stops watching the stream, the connection is removed from stream_id → [connection] and if the list is empty, send a request to remove websocket_server_id for the stream_id in the connection storage.

When a send_emoji is called, it goes to the queue and to the fan-out service. The fan-out service looks at the connection storage for a list of websocket_server_id for a given stream_id to forward the request to. On a WebSocket server, it looks at all the connections to forward the emoji to for a given stream_id.

Step 6: Deep Dives

Candidate:

Ok, so now that I've described the flows, we have a working solution. Here is a list of topics I want to address:

1. The fan-out factor is huge when there are a lot of watchers in a stream.
2. Connection storage complexity.
3. Scale for the global customer base.

4. WebSocket node failure.
5. Persistent connection state.
6. Store emoji for replay.

I feel like the fan-out factor is the most important one to address, should I go ahead?

Interviewer:

Yes, please.

Fan-out Factor

Candidate:

As discussed in the nonfunctional requirement, we can anticipate tens of millions of concurrent watchers for a given stream. That means in the worst case if all the watchers press an emoji at the same time, we will need to fan-out tens of millions of emojis to tens of millions of users! That's a huge thundering herd problem with the potential to hit north of a million QPS at the same for a short duration if there's something eventful for a given stream like Michael Jordan making a buzzer-beater shot.

Option 1: Just Let It Fan-Out

We will need to scale the stream service and emoji queue to handle 10 million QPS by sharding. The fan-out service can do micro-batches but there may still be a significant amount of backlog leading to delay in emoji broadcasting. For each micro-batch, a lot of emojis will need to be fanned out. Even if we make the system work, the amount of emojis a user receives will be overwhelming for the device. Even if the device works, 10 million emojis on a user's screen is not particularly helpful.

Option 2: Let the Client-Side Sample

Since accuracy becomes less and less important when the number of users goes up, as there will be a lot of overlapping emojis, you can have the client sample based on the number of concurrent watchers. For example, you can have a mathematical function such that for a 10 Million concurrent stream, you may decide that on each click, there's a probability of 0.01% of going through. Even with something as low as 0.01%, there will still be plenty of

emojis to be processed. If there are only 10 concurrent watchers, it may have 99.9% of going through. That way, the expected emoji traffic is more predictable per stream and it significantly decreases the QPS while still providing a good user experience.

Option 3: Let Fan-Out Service Sample

Fan-out service can conduct micro-batches every second to get an aggregated count per emoji for a given stream.

Stream Id	Emoji Id	Count
-----------	----------	-------

When the count is high, (say more than 20 is considered high), instead of showing 20+ emojis within a second, work with the UI team to show an “emoji confetti” that signals a lot of people pressed it, instead of flooding the screen with a bunch of the same emoji. So a count of 20 will show the same emoji confetti with a count of 1,000.

Conclusion

I would consider a combination of options 2 and 3. Option 2 to reduce the throughput to stream service and emoji, and option 3 to reduce the fan-out done by the WebSocket servers and improve user experience.

Connection Storage Complexity

Candidate:

One of the challenges I glossed over is the challenge behind fanning out to the WebSockets from the fan-out service. Here are some options:

Option 1: Always Fan-out to all the WebSocket Servers

The current design is to update the connection storage as connections are added and removed. Let's consider another alternative: what if we always fan-out to all the WebSocket servers? The advantage is simplicity; you don't need to update the connection storage as connections are added and just forward the emojis to all the WebSocket servers known to be available. But the cost of forwarding could be excessive if there are streams with very few watchers. For example, if there's a stream with 1 watcher and there are

1,000 WebSocket servers, for every emoji, you have to forward to 1,000 servers.

Another consideration is the fan-out factor. Let me do some calculations:

Assume 30% of the 100 million DAU are active during peak time, so that's 30 million concurrent WebSocket connections we need to maintain. Assuming we configure each server to handle 250,000 connections, we will need 120 WebSocket machines. For each emoji, we will need to fan-out to 60 WebSocket servers. If only 1 server needs it, that's a huge fan-out factor. As mentioned in the nonfunctional requirement, it's a long tail of videos so the majority of the videos won't have too many watchers.

Option 2: Use the Connection Storage

With the current design, as mentioned in option 1, the challenge is to have to update the connection storage as connections are added and removed, and depending on how frequently connections are opened and closed, the QPS may be high enough to worry about scaling for connection storage. Also, there's a slight overhead on read for `webSocket_server_ids` whenever the fan-out server causes a slight delay.

Conclusion

Given the big fan-out factor and the tail distribution of the videos, I believe it's worth the complexity to maintain a connection storage to reduce the load on the WebSocket servers. If only 1-3 servers need to handle it as opposed to 120 servers, that's a huge saving at the cost of maintaining the connection storage.

Interviewer:

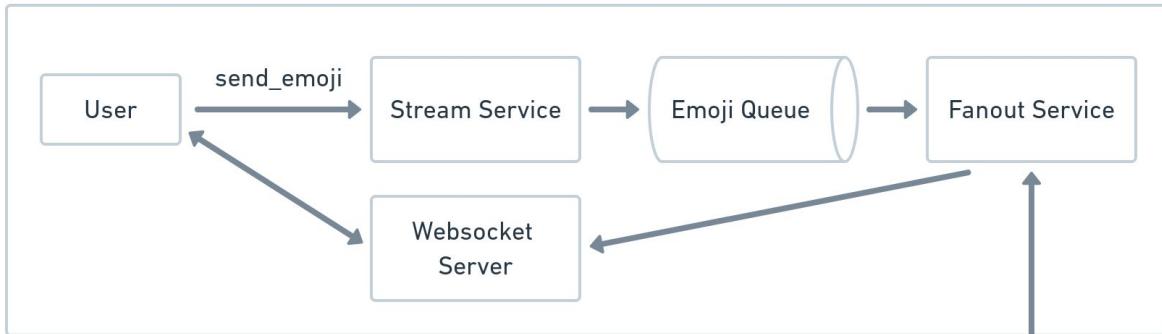
Ok, that seems fine to me based on the assumptions we're making. What about a globally distributed user base? It looks like your design is just for a local region, so what if I'm watching from another region?

Globally Distributed User Case

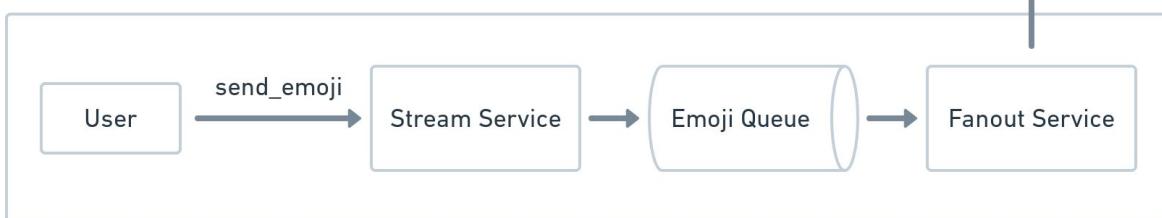
Candidate:

That's a good question, let me just draw out what it would look like first.

Region 1



Region 2



Basically for a region, a similar pattern needs to emerge where a region needs to know which other regions are also watching the same stream. I think we have similar options. By the way, how many regions can I assume we have?

Interviewer:

Let's assume 3.

Candidate:

Ok, here are the options:

Option 1: Forward the Emoji to All the Regions

For every emoji request, I will forward the emoji to all 3 regions. The downside is if a region doesn't have any users watching that stream, then that forwarding is wasted. The solution is again much simpler without a region connection store.

Option 2: Have a Global Region Connection Store

We will need a region connection store in each region that stores:

stream_id → [region_id]

Whenever we update the connection storage, we also need to consider if we need to broadcast the change to the other regions. When the fan-out service gets an emoji request, it looks for its local websocket_server_ids to broadcast to as well as other regions to forward the request to.

Conclusion

For regional design, I would actually prefer option 1 to always broadcast instead of maintaining a global connection store. The reason is that the fan-out factor is 3 compared to 120. The complexity of fanning out local updates can be unreliable and expensive to solve for a small fan-out. Most importantly, the probability of reaching a user watching a specific stream is higher within a region than a server. The reason is because there are more users in a region than a server, since a region consists of many servers. Maintaining a global connection store is significantly more complicated than a regional connection store.

Connection Store Design

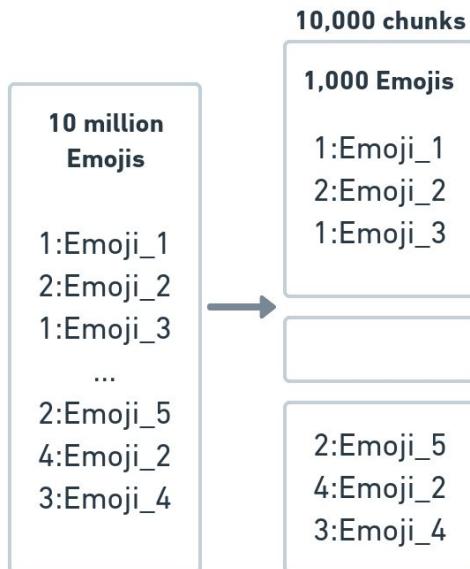
Interviewer:

Ok, that makes sense, it is interesting how two similar partners have totally different outcomes. Can you go a little bit deeper into the connection store? You mentioned how to update the store but not the read path.

Candidate:

Sure, that's a good point. I am also worried about the read path because of the potential thundering herd issue with 10 million emojis at peak for the emoji queue. Let's dig into the details a bit.

Let's assume we batch 1,000 emojis, that would be 100,000 chunks. For each chunk, we batch all the keys together to make one call to the data store which makes it 10,000 QPS. This is the read throughput at the absolute peak, which can be risky.



Here are some options:

Option 1: Use a Durable Store

We can consider using a durable store since the nature of the schema is key-value, it will be more efficient if we are able to fetch a list of `websocket_server_ids` per key lookup as opposed to row oriented, where the `websocket_server_id` could be stored across different rows. I believe we should have some sort of durability because if the store goes down, we need to rebuild it from the local WebSocket connection store which can be expensive with 120+ servers, plus the complexity to coordinate that effort.

Read latency shouldn't be that big of a deal since we're comparing 5ms disk read to a fraction of a millisecond which is trivial for a fan-out like this. However, it's difficult to support 10,000 QPS on disk so we may incur the additional complexity to shard.

Option 2: Use a Cache

Caches should be able to handle the 10,000 QPS and the schema is a key-value which means we'll get the list of `websocket_server_ids` blob efficiently. The downside is the complexity to rebuild the cache if it goes down and the complexity to invalidate as data changes. For this particular option, I will consider a periodic backup so in the case of failure, we can

use a potentially stale backup. The downside to that is inconsistency if the backup didn't capture the latest information. The implication is some users may not get the emojis and the server may be sending emojis to users who are disconnected. This seems like an acceptable experience since I am assuming we have some leeway for accuracy as long as we back up frequently.

Option 3: Use Both Durable Stores With Read-Through Cache

Instead of using a cache cluster with periodic backup, I can use the database as the source of truth and use a read-through cache to handle the writes. The complexity is the invalidation when the database is updated. Sometimes during invalidation with idempotent delete, there may be occasional cache misses with read-through cache leading to unpredictable latency upon connection update.

Conclusion

Personally, I see options 2 and 3 as solid options. Option 1 has too much complexity with the need to shard to scale for QPS. Option 3 has better consistency but option 2 is simpler with just a single cache cluster with backup at the cost of inconsistency. I still prefer option 2 as the connection is something that is frequently changed and is inconsistent anyway because of disconnections.

Interviewer:

Ok, personally I prefer option 3 but I think your justifications make sense since it's just a difference in opinion on how much consistency matters here.

Load Balance WebSocket

Candidate:

Right, we can always be incremental, start with option 2 and monitor the cost of inconsistency and incorporate a database layer if we need to. There's one thing we haven't discussed, and that's how we're going to load balance the WebSockets. Do you mind if we briefly touch on that?

Interviewer:

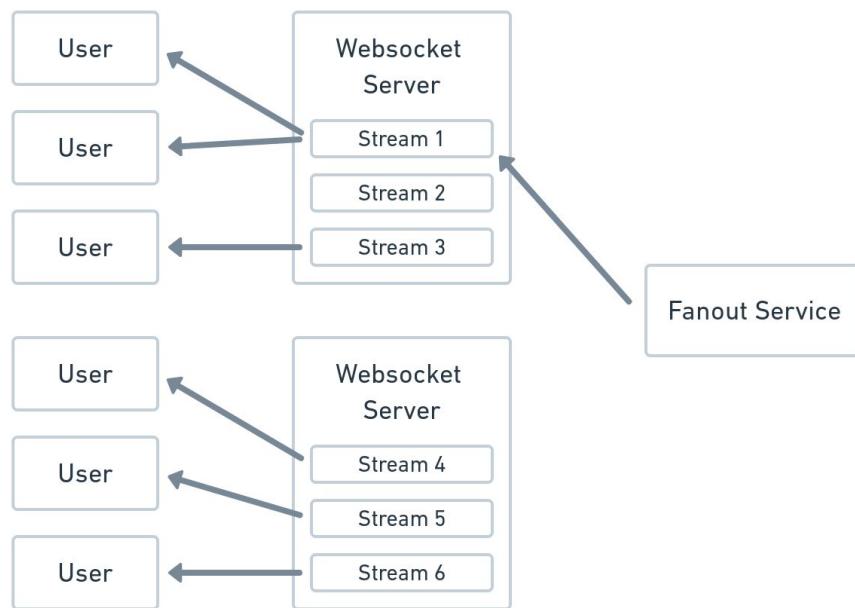
Sure, that sounds important.

Candidate:

As I'm designing for the connection store, I noticed in the worst case, an emoji needs to be broadcasted out to all the WebSocket servers for a given stream if there's at least one user watching that stream per WebSocket server. Here are some options:

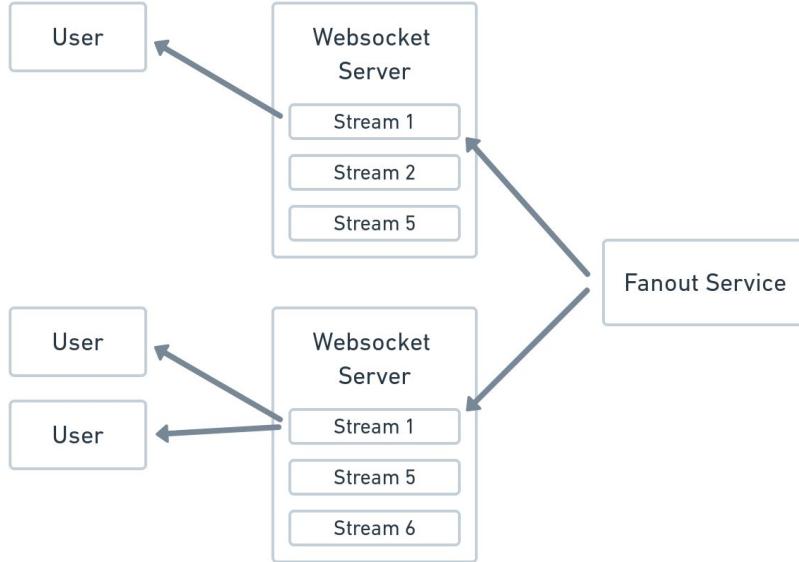
Option 1: Assign Base on Stream ID

Each stream can only be hosted on a particular WebSocket. The advantage of doing so is when the fan-out service only needs to forward the request to one WebSocket server. The disadvantage is the connections are now stateful, which is harder to scale out. Also, we are going to have a hotspot if a stream ID is very hot so we need to further shard a stream into micro-shards such that each micro-shard hosts the same stream. We will need to broadcast to all the micro-shards during fan-out for that stream.



Option 2: Round Robin

Instead of assigning to a particular WebSocket server, when a user connects to a particular stream, we round robin them to the next available WebSocket server.



The advantage is that the WebSocket server is stateless and can scale by adding more servers. However, during the fan-out, we need to fan-out to all the WebSocket servers based on the connection store if the same stream is hosted on multiple WebSocket servers.

Conclusion

It looks like my original proposal to round robin was good enough already, but it was worth going through the options. For option 1, given most of our bottleneck will be the popular videos and we have to fan-out any way to multiple micro-shards, it doesn't seem like a stateful proposal adds much value. If we are designing for a smaller scale where there aren't millions of users that require multiple WebSocket servers per stream, then I think option 1 may be preferable without the complexity of the connection store, since we know which hosts the stream is hosted on.

Interviewer:

Great, before we wrap up, imagine we want to support stream replays. How would you design for that?

Stream Replay

Candidate:

That's an interesting design question. The notion of time becomes important since we need to map an emoji to a particular point in time during a stream. There are a couple of options:

Option 1: Store the Emoji Timestamp Using the Client Clock

For each emoji clicked by the user, pass the relative timestamp of the stream and store every single click by the user. When a user watches a replay, use the relative timestamp to display the emoji at the appropriate time in the stream. The complexity is the need sample as this could result in a bad user experience with flooding emojis, just like in the live stream. But the upside is the emoji relative timestamp will be stored for exactly the frame the user intended to click on.

Option 2: Store the Emoji Timestamp Using Stream Service Clock

Instead of using the client's clock, use the server's clock when the emoji is received. This solution has all the cons of option 1 but at least it can prevent malicious users from inserting bad timestamps.

Option 3: Use Fan-Out Service Clock

Instead of building it from scratch again with the raw emojis, leverage the processing that has already been done and let the fan-out service persist the list of emojis it intends to broadcast out per stream with a schema like the following:

(stream_id, time_slice) → [emoji_id]

The fan-out service will have to coordinate with the streaming service to come up with a best-effort on which relative timestamp frame the emojis are for. The downside is that it may be inconsistent with the users' intention. The pro is it reuses much of the existing infrastructure.

Interviewer:

Great, that sounds similar to what we do in production. We're out of time, thanks for your time.

Instagam

Interviewer:

Please design an Instagram application where users can post a photo with a description. Users can follow other users and see a list of Instagram feeds from people they follow.

Step 1: Gather Requirements

Functional Requirement

Candidate:

Ok, so I'm going to assume the product exists to allow people to share their lives with the public to connect with and follow others and to get inspiration, and the more relevant the posts, the better the engagement and user experience. From a reader's perspective, I will assume they can scroll through a list-style feed ranked by some score. From the poster's perspective, they can post a photo with the description. Do I have the correct understanding of the product?

Interviewer:

Yup, that sounds good to me.

Candidate:

Ok, I have more questions about the feature:

1. Should I worry about letting the users post multiple photos?
2. How are the scores calculated? I would assume attributes like affinity with the follower, freshness, and engagement scores are important, like how Facebook uses Edgerank?
3. Is there a limit of how many feeds they can see?
4. Is there a size limit to the photo or the type of photo we support?
5. Do we support both mobile and web?

Interviewer:

Those are good questions. Here are the answers:

1. Just worry about one photo per post for now.
2. Those are really good signals. For now, let's just keep it simple and use the timestamp as the ranking factor. Don't worry about EdgeRank for now.
3. Let's cap it to 500 feeds now since we don't expect users to scroll through that many.
4. Let's cap it to 2 MB.
5. Yes, please support both.

Candidate:

Thanks for answering those questions. I have a couple more questions regarding the scale and performance constraints.

Non-Functional Requirement

Candidate:

How many users should I assume for Instagram, and how are they distributed across the world? Is there a distribution of followers like celebrities? What's the maximum follower list an user can have?

Interviewer:

Let's assume there are 500 million daily active users across the whole world. The distribution of followers is a long tail where there are a few celebrities most people follow. Let's say we allow people to follow up to 1,000 users.

Candidate:

How important is the accuracy of the feeds, meaning is it still a good experience if we miss occasional feeds? I would think accuracy is important, but I want to confirm. What is the freshness requirement? When a user posts a feed, how quickly should it show up?

Interviewer:

As you said, the accuracy of the feeds is very important since a missing feed will result in users feeling like they missed out when all their friends are talking about a particular feed. Regarding freshness, it is important, but

you have some flexibility since a delay of a couple of minutes won't immediately lead to a poor user experience.

Candidate:

Ok, I'm going to assume the durability of the posts is very important since losing posts will lead to a poor experience on both sides. It's part of their life event.

Interviewer:

That's right. I'm sure you wouldn't want to lose your childhood photos; they're precious and important.

Candidate:

Ok, before I proceed, I will assume latency should be relatively low with p99 at 200 ms. Is that a fair assumption?

Interviewer:

Yes, it's the first impression of the product. It needs to be fast.

Step 2: Define API

Candidate:

Yup, I would be pretty frustrated if it loaded too slowly. Now I have my questions answered, let's focus on the APIs needed to finish the design.

post_feed(user_id, photo_byte, description) → status

read_feeds(user_id, offset) → [feed]

load_image(user_id, photo_url) → photo_byte

Regarding followers, I'm just going to assume the table already exists. I'll add more details later if need be.

For post_feed, the user passes in the photo_byte and the description. Regarding the post's timestamp, I'll just assume it's server-generated since it's an online operation, and we don't need to deal with all the complexity of client-side clock skew.

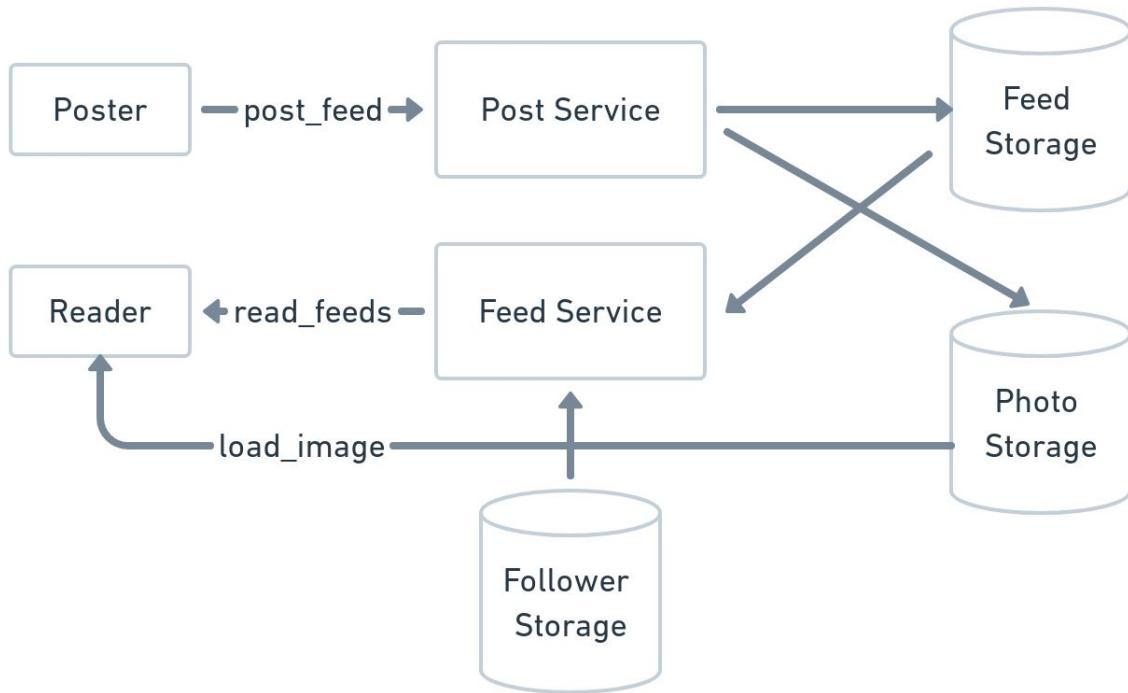
For `read_feeds`, the feed object is a `photo_url` and the description. I will need to consider pagination here since showing 500 feeds all at the same time will cause a lot of unnecessary loads since users don't need to see all the feeds at the same time, and most users will never scroll beyond 20-30 feeds. The offset is just used for pagination to specify the offset of the paginated chunk. For now, I will assume the paginated page size is fixed so the client doesn't have to pass the page size.

For `load_image`, it calls a photo service to fetch for the blob with the `photo_url`.

Step 3: Define High-Level Diagram

Candidate:

Now that I have the APIs let me think through the high-level diagram that completes the flow for the APIs.



Warning

Since this is a very popular question, most people immediately jump into the fan-in, fan-out, and hybrid discussion like a prophet without even

identifying the problem you're trying to solve. Premature optimizations will make you seem unnatural and memorized.

Remember, start simple, identify the problem, then come up with options and trade-offs.

Step 4: Schema and Data Structures

Candidate:

So this is my high-level diagram. Let me add more detail about the schemas needed.

Feed Storage

Each feed belongs to a user and has a photo_url with a description. Since we're going with one photo per feed, this is fine for now but will need to refactor if we support multiple photos.

Feed Id	User Id	Photo Url	Description
---------	---------	-----------	-------------

Photo Storage

For each photo URL, there's a photo blob. The relationship can be stored using a blob store. There's some optimization for bandwidth and latency via CDN we can discuss later if we have more time.

Photo Url	Photo blob
-----------	------------

Follower Storage

This is a many-to-many table for followers and followees.

Follower Id	Followee Id
-------------	-------------

Step 5: Summarize End to End Flow

Candidate:

Now that we have two main APIs, let's go through how each one works.

For post_feed, it passes the bytes and description to the post service. The post service will try to commit the photo bytes into the photo storage then get the photo_url in return. Then it will commit the photo_url and description into the feed storage.

On read, the reader will call the feed service, and the feed service will read a list of followers. For each follower, we will call the feed storage for a list of feeds. Then the feed service will need to merge the feeds and serve them to the reader. This retrieval pattern looks like an expensive operation to achieve 200 ms. Let's discuss this more later.

Once the reader gets a list of feeds, for each feed's photo_url, it will fetch the photo_bytes from the photo storage.

Step 6: Deep Dives

Candidate:

Now that I've summarized the flows, here is a list of topics I would like to cover:

1. Optimize for the feed read query.
2. Scale for the read query for globally distributed users.
3. Distributed transaction of the blob store and feed store.
4. Users with poor bandwidth.
5. Reduce bandwidth for the photos.

I think optimizing for the feed read query is critically important. Are you ok with me starting there first?

Interviewer:

That sounds fine to me.

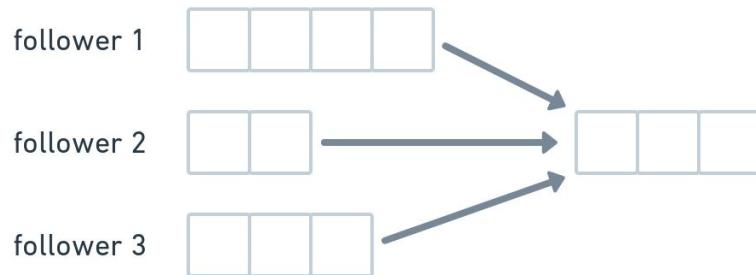
Optimize for the Feed Read Query

Candidate:

As described, the requirement is to achieve 200 ms for p99. I feel like the current design will not satisfy that. Let me think it through.

Option 1: Fan-Out

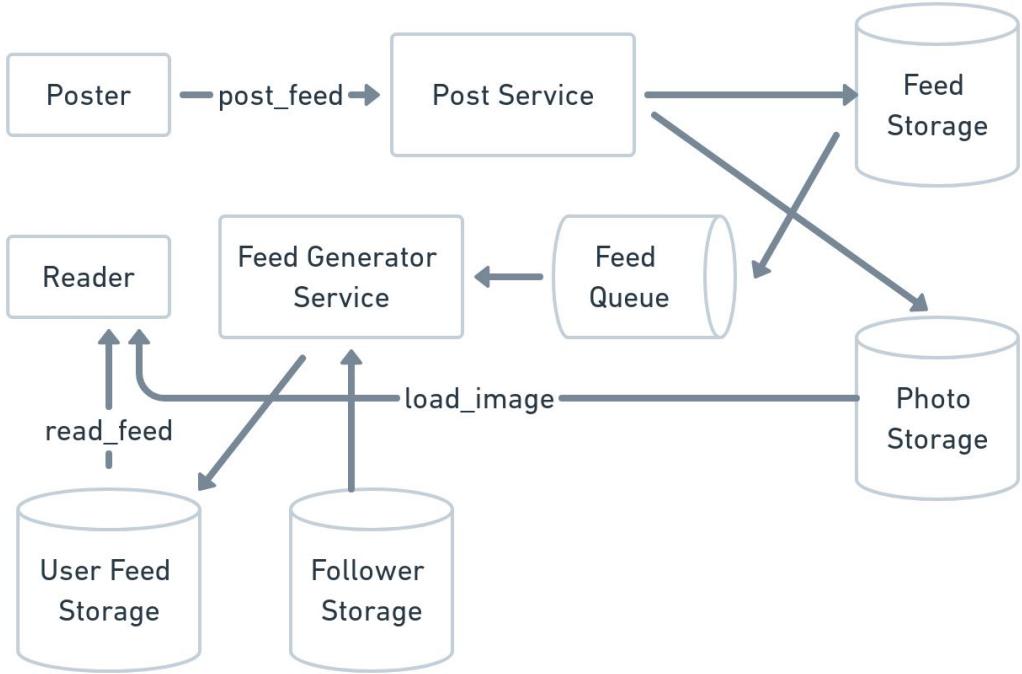
Let's assume the worst case where a user follows 1,000 users. To generate the top 500 feeds, I need to fetch 500 posts per user because all the top 500 feeds can come from one user. We can decrease the number of feeds fetched per user, but it is lossy and still expensive. For now, we'll assume the feed table is indexed by timestamp per user. Assume we fetch 500 posts per user sorted by timestamp, k-list merge them until we have the top 500 posts.



The amount of disk IO needed to fetch for 1,000 keys even if we batch them would be too expensive. Also, for each operation, we need to keep $500 \times 1,000 = 500,000$ posts in memory to k-list merge them into the top 500. The upside is it's simple since we don't need to precompute any result.

Option 2: Compute on Write

Instead of doing the expensive operation per read, we can compute the result on write such that each user has their custom list of recommended Instagram feeds.



User Feed Storage

`UserId → [Feed]`

Every time a user makes a post, it is committed to the feed storage, and the feed storage sends an event to the feed queue. The feed generator service processes it by looking for all followees and fans that out to them. The advantage of doing so is the speed of a key-value lookup is extremely fast. The downside is the fan-out factor can be huge if the poster has a lot of followers. Imagine the poster has 10 million followers. That is 10 million key updates!

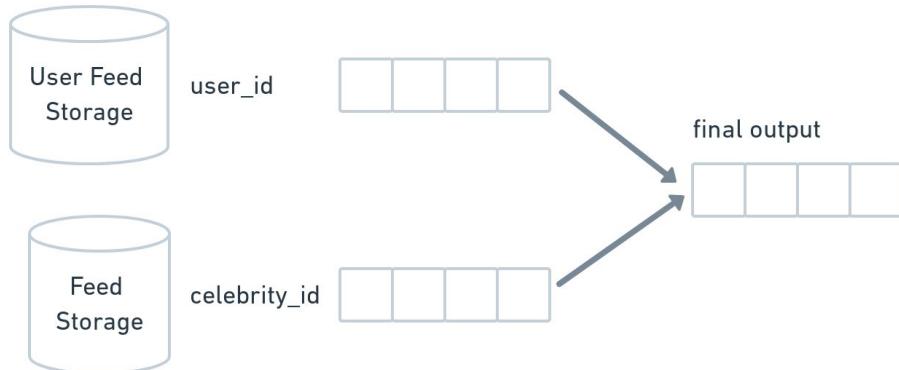
Also, this compute-on-write pattern will get more complicated if the outcome depends on more attributes. For example, if the affinity score changes, the ranking could change, and the user feed list needs to be updated. If the ranking score decays over time, the listing also needs to change. Even now, if a user stops following a celebrity, the feed list needs to be updated.

Option 3: Periodic Update

Another possibility to prevent the fan-out and changing dependencies is to compute the list periodically. The advantage is fast read by key-value lookup per user. The biggest advantage is we don't need to worry about updating the list when any of the list dependencies change. The big downside for this approach is that computing the suggestions for all 500 million users every couple of minutes will be extremely expensive, especially if the users aren't active. Imagine a user with one follower. The more frequently we run, the fresher the data at the expense of more computational power needed. So if we don't run frequently enough, we run into the risk of a stale list.

Option 4: Hybrid Between Fan-Out and Fan-In

Since fanning out is huge for celebrities and the cost of fanning in is huge for non-celebrities because there are many users to fetch, we can combine the two where we fan-out for non-celebrity and fan-in for celebrity.



Since there won't be too many celebrities, the fan-in won't be that compute-intensive. The major downside to this approach is complexity. For example, you need to keep track of the celebrity list dynamically based on count or config. And if it's dynamic, there's the possibility of hovering around the threshold, causing duplicate or missing feeds. Imagine someone is a celebrity, so it's not in the user feed storage. Then the celebrity becomes a non-celebrity. The previous celebrity feeds will all be missing and require a backfill. Imagine a user is a non-celebrity and becomes a celebrity, there will be duplicate posts since it's in the user feed storage and retrieved along during the fan-in from the feed table.

Conclusion

I would choose option 4 as it satisfies the requirement while not wasting a lot of resources for computation. For the celebrity definition, I would hire a small operation team to monitor the threshold and remove the duplicate and backfill as necessary, since they don't happen often. Eventually we can automate those operations.

Interviewer:

Oh cool, in fact we're actually working on automating that process right now, and you brought up some good pointers to think about. Let's talk a bit more about the duplicate use case when a non-celebrity becomes a celebrity. How would you handle it?

Duplicate Posts

Candidate:

Let me think of a couple of options.

Option 1: Do Nothing

One option is to allow the duplicate posts to happen from the user feed storage and fan-in from the feed table. A user becoming a celebrity is extremely rare, and an occasional duplicate post won't be a terrible user experience. We can just not worry about it.

Option 2: Dedupe in Aggregation with K-List Merge

When we combine the results from the user feed storage and feed table, we can dedupe using k-list merge where posts with the same timestamp are adjacent. The challenge of deduping with k-list merge is that even though both lists are sorted by timestamp, there can be other posts with the timestamp, so the same post might not be adjacent.

Option 3: Dedupe in Aggregation with HashSet

When we combine the user feed storage and feed table results, we can dedupe using a HashSet. The challenge for this is the memory requirement to keep all the posts.

Conclusion

Given the complexity and investment with options 2 and 3 and how rare the celebrity promotion event is, I would just go with option 1. Duplicate posts may be slightly confusing for a second for the user, but I don't think it's something that will cause an angry and frustrated user.

Global User Base

Candidate:

It is an interesting question indeed. Now another concern I have is how global user distribution impacts latency. The write fan-out to all the user feed storage across all the data centers shouldn't be a big concern since a slight delay of freshness is acceptable. However, read latency needs to be fast, so I want to dig into that a little bit more.

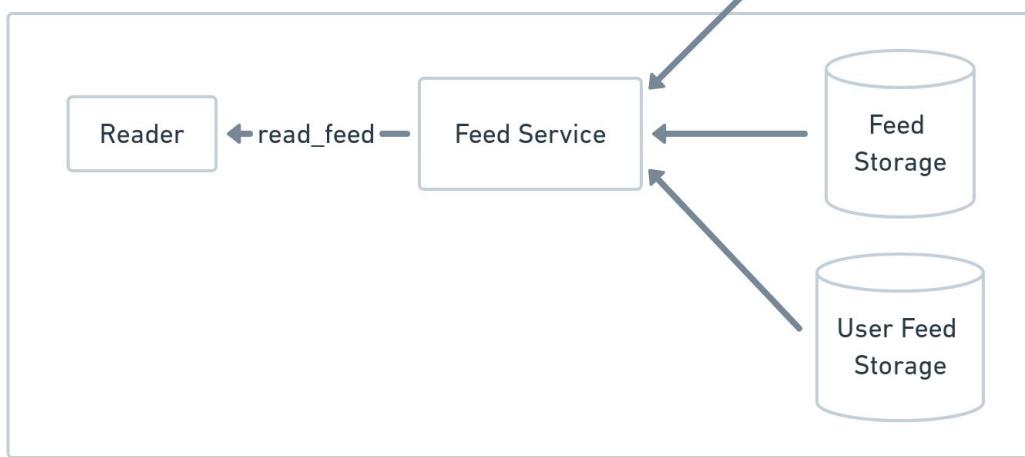
Option 1: Cross-Regional Read

One solution is to use a cross-regional read for the regional celebrity shard. Since we target 200 ms for p99, cross-globe network latency is around 150-200 ms, making this very unstable. This latency can be even worse when more regions are involved. The upside for this solution is simplicity since we can just do a cross-region read, and consistency will be stronger.

Region 1



Region 2



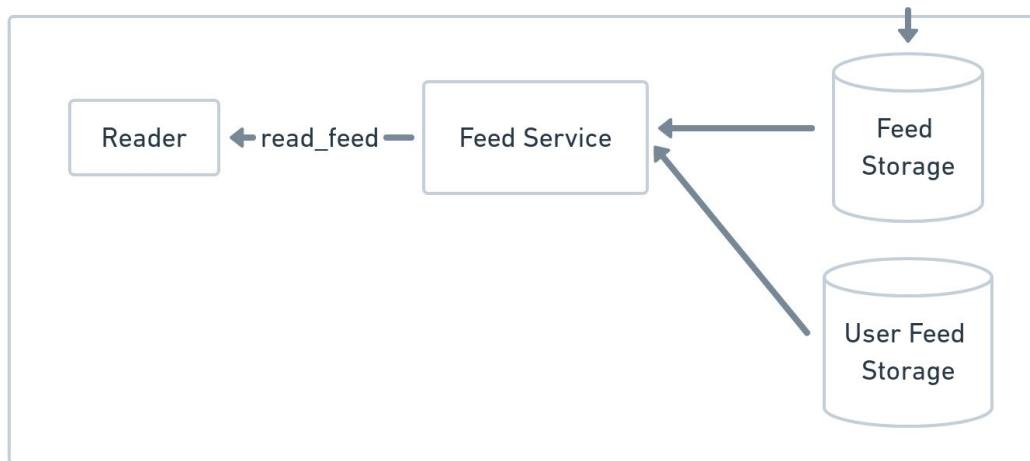
Option 2: Cross Region Replication

Instead of reading from other regions, we can replicate the feed storage from one region to another.

Region 1



Region 2



Upon reading, the user will fetch both user feed storage and feed storage from their local region. Global replication will speed up the read query.

The downsides are the possible delay if we assume an asynchronous replication. As part of the non-functional, we agree a slight delay is acceptable, so this solution is acceptable as asynchronous replication is fine.

There will be extra storage to store all other data centers' feeds. Regarding extra storage, we are already building redundancy for the feed data to improve durability, which we discussed is important. However, we need to replicate it to all the regions, so it might be slightly more than we need if we were just to do it for durability.

Conclusion

Based on the strong requirement of read latency and durability, I feel more comfortable with the second approach with the additional complexity to

deal with replication.

Distributed Transaction

Candidate:

We glossed over the distributed transaction nature of writing to the blob store and the feed storage. If one of the two transactions goes down, it will lead to inconsistencies.

Option 1: Write to Photo Storage First, Then to Feed Storage

For this option, I would write to the photo storage first to get the photo_url and use that photo_url to pass to the feed storage to be persisted. The problem is when the feed storage transaction fails, and then the photo_url becomes an unreferenced photo. To deal with unreferenced photos, we need a background job to look for unreferenced photos and delete them if we care enough to reduce the storage. The pro is that the sequential 2 step transaction is reliable. The con is the complexity of dealing with unreferenced URLs.

Option 2: 2 Phase Commit (2PC)

Assuming we control the feed storage and photo storage, we can potentially design for a 2 phase commit. If both storages agree to the transaction, we will continue to push the commit phase through. That way, it's all or nothing, and there won't be unreferenced photos. The downside is the complexity of 2PC, especially in the event of a real failure with one of the storages. In addition, the throughput will be worse with 2PC.

Conclusion

I would go with option 1 since option 2 is pretty complex for minimal gain. It's possible the feed storage failure doesn't happen that often and the cost of storing the unreferenced photo isn't that high. We can for sure monitor and build the background job if we need to.

Interviewer:

Sounds fine, now, back to the feed. Imagine you're on the feed team, and the Ads department wants to work with you to incorporate ads into the feeds. What would you think about it?

Feed Ads

Candidate:

That is pretty interesting. What kind of ads are these? How do we determine what ads to show for a user?

Interviewer:

Imagine the Ads team just started, and you have control over the ads API contract. There's an inventory of ads to be shown for advertisers, and we will show the users ads when they view their feeds. Feel free to make your assumptions for this interview.

Candidate:

Let me think about a couple of options.

Option 1: Periodic Refresh of Ads

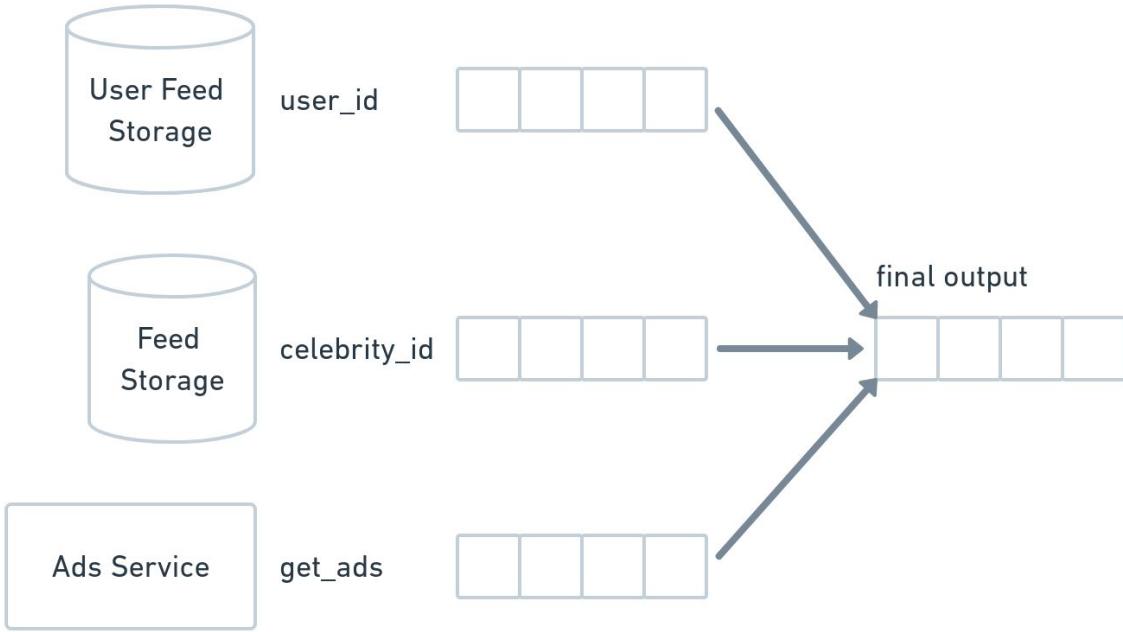
We can have a periodic job in the background to populate the ads into the user feed storage, so reading from the user feed storage will be fast. It is difficult to predict the impression users will have for an ad by prematurely materializing the ads into the user feed store. If an advertiser only wants 10 impressions and we fan it out to 10 users, the 10 users may never come online again.

Option 2: Fan-In on Read

The most straight forward option is to directly read for the ads service with the following API:

`get_ads(user_id) → [ad]`

The advantage is simplicity, and you don't need to worry about materializing an ad that will go stale by keeping ads as part of the user feed storage. The challenge for this design is to ensure we have low latency. As we add more services, latency will increase.



Conclusion

This one seems straightforward to pick option 2. However, the conversation I would have with the ads team is to establish a latency SLO and establish a process for more internal customers to display on the feeds to ensure the latency is low enough for the end-user. If the Ads team cannot achieve that SLO, we may need to consider some variant of option 1.

Warning

This design may seem trivial by doing a fan-in, but the purpose is for the candidate to identify the importance of having low latency for the service and the risk of building up over time. Also, it's a chance to show leadership with SLO since that's what usually happens in real-life situations.

Poor Bandwidth

Interviewer:

That seems fine to me. I'm on the Ads team, and we had a long conversation about establishing that SLO. Ok, now Instagram has a global user base, and some users don't have the luxury of high bandwidth. How do you deal with that situation?

Candidate:

Sure, that sounds challenging. Here are a couple of options I can think of:

Option 1: Lossy Client-Side Compression

I am not a compression expert, but I know we can apply some less efficient compression algorithms to reduce the resolution to reduce the bandwidth needed to upload the photo. The client-side compression shouldn't be too compute-intensive, but the image quality will suffer. Since Instagram is a photo-based product, sacrificing photo quality should be carefully considered.

Option 2: Ensure the Upload is Idempotent

Photo upload can be all or nothing, meaning if the upload fails, you must re-upload again. Instead, we can assign the upload an ID where if the client gets disconnected and connected again, they can resume the upload from the file offset they left over. The pro is we will eventually finish uploading, the con is additional complexity, and the upload may take a long time.

Option 3: Limit the Size of File That Users Can Upload

Instead of letting the users upload a large file, we can restrict them to uploading a fraction of the maximum size. For example, if the maximum is 2MB, we might tell them no more than 500 KB. The advantage is the system has fewer bytes to pass, but the disadvantage is that it can be a frustrating user experience as they have to modify the photos before uploading.

Option 4: Create a Point of Presence Near Them

If we create an edge server near them, the distance the bytes have to travel will be less. Once the bytes hit the edge server, they will be within Instagram's internal network, which is more reliable and efficient. This isn't always possible since there will be many users scattered around the world, but it could be a long-term solution with something like Facebook Aquila. Let's just hope it works this time.

Conclusion

I would prefer option 2 with a warning to them that the amount of memory they're trying to upload might take a while instead of prohibiting them and allowing them the option to lower the quality if they wish. We should definitely consider option 4 in the long term, but we're not there yet.

Interviewer:

Great, that sounds great, thanks! We are out of time.

Distributed Counter

Interviewer:

Please design a service that keeps track of how many times users viewed a YouTube video.

Step 1: Gather Requirements

Functional Requirement

Candidate:

What is the purpose of having this view count? Is it to make a recommendation or just to give the user some sense of video popularity? Is it to pay the content creator? Should we worry about different time buckets of the count? If a user views the same video twice, should that be counted twice or once? Or should the view be counted again after some time?

Interviewer:

For this session, the purpose of this view count is just to inform the user about the popularity of the video based on the total count. There's no financial tie to the number. We don't need to worry about different time slices. Just worry about the total count. If the user views the video twice, we will count it twice. If we have more time in the end, we can talk about some changes counting twice will bring.

Candidate:

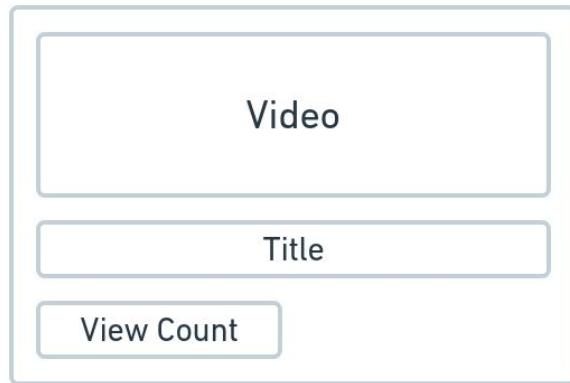
Ok, thanks. How should the view count be shown to the end-user?

Interviewer:

For now, the only place you will see the number is when a video page is loaded.

Candidate:

Ok, I'm going to assume it looks something like this:



Non-Functional Requirement

Candidate:

How many daily active users are there, and what can I assume about the distribution of the user base? What's the distribution of videos watched by the users? Are there videos that we can assume are popular? How many videos does a user watch per day?

Interviewer:

Let's assume there are 500 million daily active users across the whole world. Let's assume a long tail distribution, meaning some videos are very popular and there are many unpopular ones. You can assume each user views 10 videos a day.

Candidate:

Ok, great. What is the accuracy requirement of this number? Does the number have to be truly reflective of the actual view count? And a question related to that, does the number have to be updated in real-time?

Interviewer:

The number should be as close to accuracy as possible, but you get some leeway. This number may have some financial ties in the future, so we should take that into consideration. For now, freshness isn't that important. As long as it's within an hour or two, it is acceptable.

Candidate:

Thanks, I know accuracy is important, but what about consistency? If two users were to look at the numbers simultaneously, should the numbers be as close to the same as possible?

Interviewer:

What do you think?

Candidate:

Well, I think it's unlikely two users will communicate to each other about the exact view count of the same video. Also, there's no causal ordering to the two users, meaning it's practically impossible to tell who watched it before the other. So I don't think consistency matters much here. What about the availability of the metrics collection service and the durability of the view count?

Interviewer:

Availability is extremely important since we don't want this count service to prevent the page load. The durability of the view count is also very important.

Candidate:

Ok, finally, I'm going to assume the latency has to be relatively low because the page load for the video should be fast, like 300 ms p99.

Interviewer:

That's a fair assumption.

Step 2: Define API

Candidate:

Ok, based on the requirement, here are the APIs:

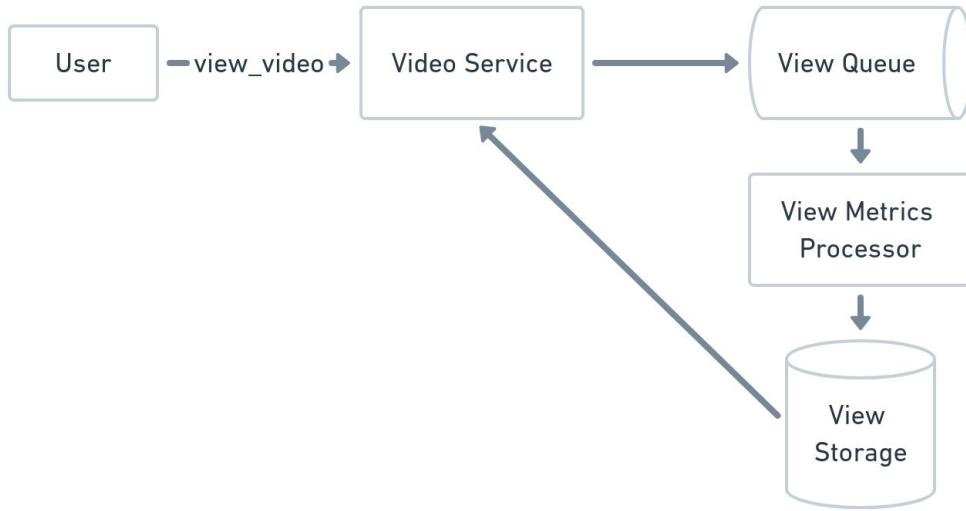
`view_video(video_id) → count`

`view_video` is the API to load the video page. For now, we'll abstract it to just an API to get the count, but when you watch a video, you are also incrementing the count at the same time.

Step 3: Define High-Level Diagram

Candidate:

The API is quite simple since we are only interested in omitting the metrics. Here's the high-level diagram.



It is pretty straightforward to insert the metrics into the queue since I expect a high write QPS. The metrics are processed and saved to view storage. At the same time, the video service reads from the view storage for the latest count.

Step 4: Schema and Data Structures

Candidate:

Let me logically define the schemas I need:

View Queue

Video Id

Yeah, you just need the video ID in the queue since we don't need to aggregate by user or time.

View Storage

Video Id	Count
----------	-------

We just need to store the count per video ID. Quite straightforward.

Step 5: Summarize End to End Flow

Candidate:

Okay, the high-level diagram and the schemas are very straightforward, but let me just make sure I have all the right components by going through the flow.

When the user visits the video page, we will call the view_video API and an event is fired to the view queue. The view metrics processor will process the view metrics and store them in the view storage. At the same time, the video service will fetch the count from view storage. Instead of waiting for the increment write to complete, the video service can just increment a count by one and return that value to the user.

Step 6: Deep Dives

Candidate:

The flow is quite straightforward, but there are some challenges to scaling for this pipeline:

1. Scale for the write throughput.
2. Scale for the read throughput.
3. Idempotency of the metrics.
4. How do you shard? What about a super hot video?
5. Storage choice?

My intuition tells me the write throughput is going to be technically challenging. Let's dig deeper there.

Scale for the Write Throughput - Delay Data

Since you told me there are 500 million DAU and each views 10 videos a day, I'll assume a peak factor of 10.

$$QPD = 5 \times 10^8 \times 10 \times 10 = 5 \times 10^{10}$$

$$QPS = 5 \times 10^5 = 500,000$$

The 500,000 QPS is for both the read and write which is quite significant. Also, since there's a long tail of video popularity, some videos are going to be extremely hot. Since the bottleneck is going to be caused by some hot

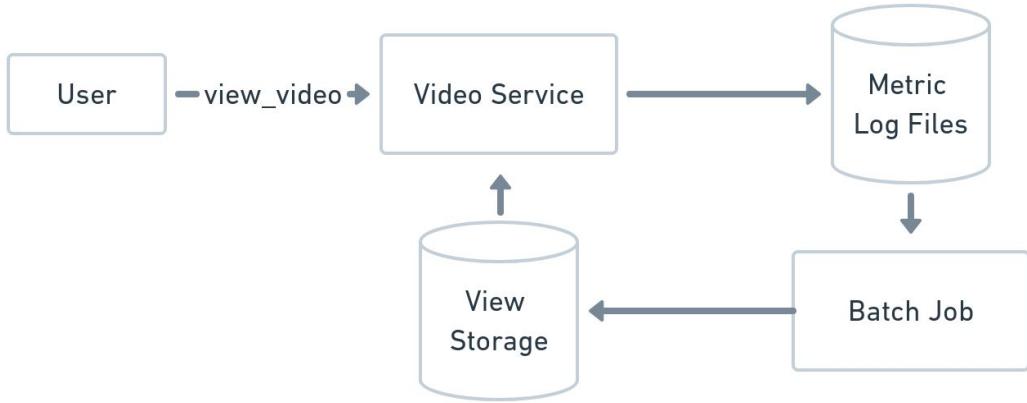
videos, I'm going to zoom in on the counter of a particular video. Let's assume the popular videos get 0.1% of the traffic at 500 QPS during peak time, which is significant for a shared key.

Option 1: Stream Processing

There will be contention for a shared row key in stream processing since we need it in real-time. For example, if I have a database row to increment count, that row will be hot.

Option 2: Batch Job

In this solution, I'm going to consider a batch job that runs every hour. For now, I'll use MapReduce to calculate the count. First, the view_video metrics are written to logs with video_ids, each indicating that it has been watched once.



Conclusion

Since the non-functional requirement indicates one to two hours delay is acceptable, batch jobs will eliminate all the complexity of a near real-time update. This is a straightforward choice.

Scale for the Write Throughput - Near Real-Time

Interviewer:

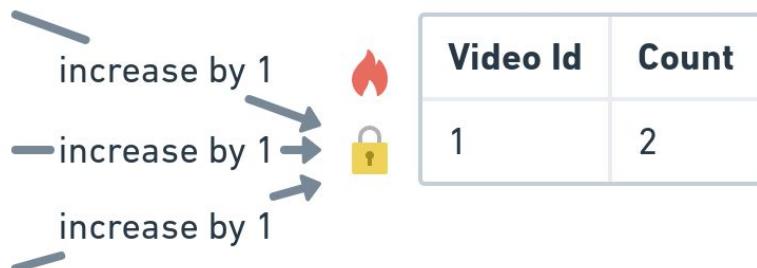
Ok, great, I'm glad you're choosing your implementation based on the requirements. Now, what happens if we want the result to be near real-time?

Candidate:

Ok, I guess I can see that coming. With the changed requirement, I need to consider how to scale for near real-time handling.

Option 1: Partition the Database Table

In this solution, I will partition the databases to scale for the high QPS based on hash using consistent hashing because range-query isn't necessary. Once a video hits a table, it will acquire the row-level lock, increment the count by 1, and release the lock. We can rely on atomic writes provided by the database, but internally, it needs to handle it serially. The lock is necessary because there will be concurrent threads trying to update the same count for a hot video.



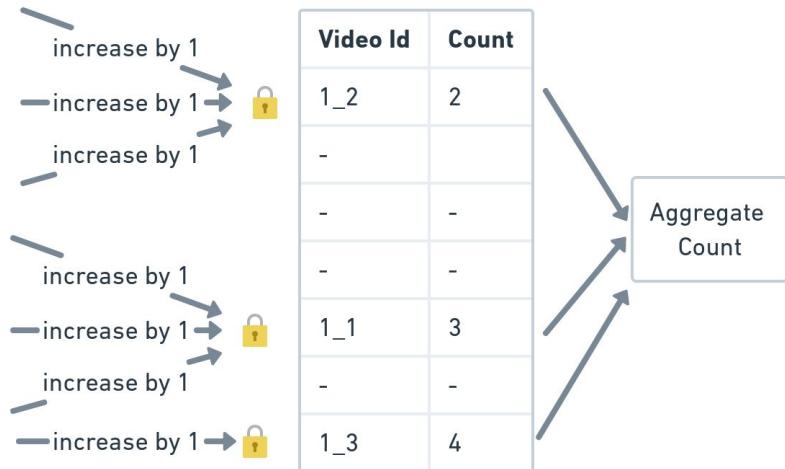
The advantage of sharding is that less popular videos can scale since more machines are taking in the write for 500,000 QPS. However, as you can see, contention for a given row will still be hot for popular videos.

Option 2: Shard Further into Multiple Rows

Instead of relying on a single row, we can partition even further for a particular video_id key to increase the overall throughput. For example, if I want the throughput to increase by 10, I can specify multiple rows to handle that key. I will need another mapping to store:

Video Id	Row Shard Count

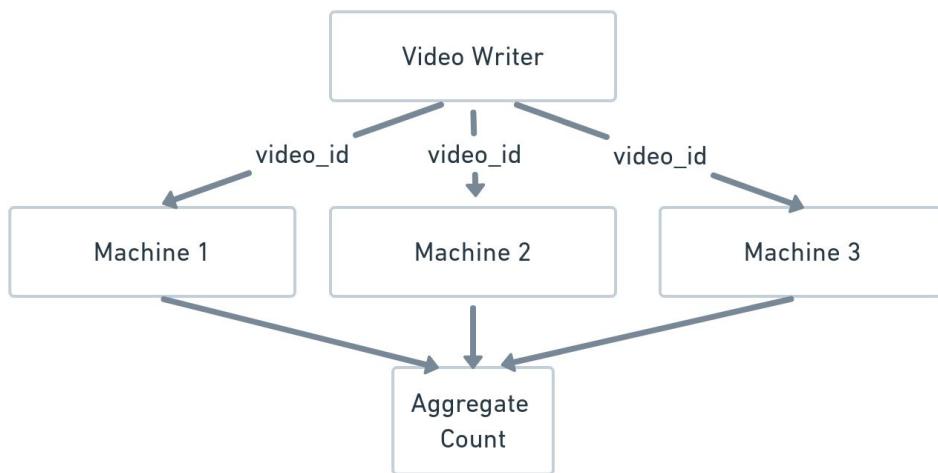
If row_shard_count is 3, we will create row IDs of [video_id]_1, [video_id]_2, and [video_id]_3, and each row will have its row lock. But at least the throughput just tripled. On read, we will aggregate all three rows up.



The downside is that the read query will be slightly slow with scatter-gather, but the difference should be small unless the row count per key is high. Another potential downside is if the particular table is super hot, we need to scale a key beyond a machine.

Option 3: Shard a Video Key Across Machines

For each `video_id`, use a round robin algorithm to distribute the count to each machine. And within each machine, we can use option 2 to increase the throughput with multiple rows. On read, we fetch from all the three partitions and aggregate them up.



The upside for this approach is the ability to scale beyond a single machine. The downside to this approach is the high latency read across multiple

machines, and on each machine, we need to aggregate the count across multiple rows. Also, availability may suffer because if any machine is down, the client will need to retry the write until a new leader is chosen for a shard.

Option 4: Conflict-Free Replicated Data Type (CRDT)

Similar to option 3, each machine will continue to take writes. However, instead of conducting scatter-gathers from multiple hosts, each host will periodically asynchronously publish their count per video to all the other nodes. Each node has an eventually consistent view of what other nodes see.

Option 5: Sample the Video Count

We can sample the count where the client has a 10% probability of sending in an event, and the server will multiply the number by 10 in the backend. Sampling will reduce the QPS by 90% since we are only handling 10% of the traffic. However, this sacrifices accuracy since we use a probabilistic model, and accuracy is a strong requirement.

Conclusion

I would choose option 4 since the requirement for strong consistency isn't strong. We can choose an eventually consistent data structure like CRDT. It meets the latency requirement needed since we just need to query a single host. We can handle the write throughput, and the design is highly available since there are multiple nodes for reading. The downside is extra complexity to implement CRDT, which is a reasonable expectation since we have such a big scale.

Interviewer:

Ok, that's reasonable. Anything else you would worry about regarding your design?

Idempotency of Metrics

Candidate:

Yes, one of the things that worries me the most is the metrics' idempotency, and you said accuracy is very important. When the client calls the server, if

the server commits but the client never acknowledges, the client may decide to retry and double count. Since we can never be sure if two calls are intended to be two views or the same view. Let's zoom in on the view queue as a point of idempotency discussion.

Option 1: At-Most-Once

One of the ways is to use the at-most-once semantic. For example, if the producer sends an event, it doesn't wait for the response and just moves on to the next metric to omit. The advantage is the event is never sent twice, and the throughput is better. However, some events might end up getting dropped.

Option 2: At-Least-Once

Another way to handle this is to use the at-least-once semantic. For example, if the producer never receives an acknowledgment, it will try again. At-least-once will lead to overcounting. The throughput will also be slightly worse since the producer needs to wait for an acknowledgment.

Option 3: Exactly-Once

We can make the call idempotent by tying a user session and timestamp with each metric. If we do end up retrying, we would ask if that user session has already omitted an event for that timestamp since it's impossible to watch a video twice on the same timestamp for the same session. Exactly-once requires another temporary storage to check for idempotency and requires more metrics about the session to be collected. Also, the throughput will worsen as a result of the check.

Choosing the right semantic is a really tough one since you said accuracy matters, but freshness also matters. If we have to pick, is overcounting worse than undercounting?

Interviewer:

What do you think about it?

Candidate:

Well, I think the retry can cause significant overcounting. And for undercounting, if it doesn't happen often, it wouldn't be as bad. My approach would be to go for option 1 since it's simpler, with better throughput without the risk of overcounting. In the long term, I think we need a variant of option 3 anyway, so when a user refreshes a video page, the metric doesn't get counted multiple times within a timeframe.

Interviewer:

Ok, the arguments are reasonable. It is indeed a hard problem. How would you change your design if you have to keep track of the number of unique users?

Number of Unique Users

Candidate:

Good question. I need the metrics to pass the user_id now:

view_video(user_id, video_id)

Can I assume user_id is a BigInt incrementally increasing number? And how many unique videos are there?

Interviewer:

Yes, that's fine to assume about the ID. Let's assume there are 100 million videos on YouTube.

Candidate:

Ok, well, here are the options regarding the data structures that I'm aware of:

Option 1: Use a Set Data Structure

From a data structure standpoint, I need a set for every video_id.

video_id → { user_id }

Let's assume, on average, there are 50,000 unique users viewing each video, and each user_id is 8 bytes. I will round it up to 10 bytes. With 100

million videos, let me calculate the memory needed for one instance:

$$1 \times 10^8 \times 5 \times 10^4 \times 10 = 5 \times 10^{13}$$

Which is 50 TB.

Option 2: Use HyperLogLog

Based on the user_id, we can make some probabilistic guess as to how many users we've seen so far. The rarer the number, the more likely we've seen more. Let's assume that each count is a BigInt with 8 bytes, and we'll again round to 10 bytes. With 100 million videos and each has one BigInt count:

$$1 \times 10^8 \times 10 = 1 \times 10^9$$

Which is 1 GB, and we can fit that into memory!

Warning

HyperLogLog is an advanced data structure that isn't always expected in a system design interview. Still, it's useful to know if similar questions come up.

Conclusion

Depending on the accuracy needed for the unique user count, we can use HyperLogLog to significantly improve the memory footprint at the cost of less accuracy. For now, I'll use HyperLogLog. If we need more accuracy, we can go into how we will scale it. Should I go into the system more since I've just talked about the data structure?

Interviewer:

No, you're good. I wanted to see if you saw a difference between total count and unique user count. I'm surprised you know HyperLogLog, though. Ok, but going back to the system, we haven't talked about scaling

the read path for total view count. Can you describe in more detail how that works?

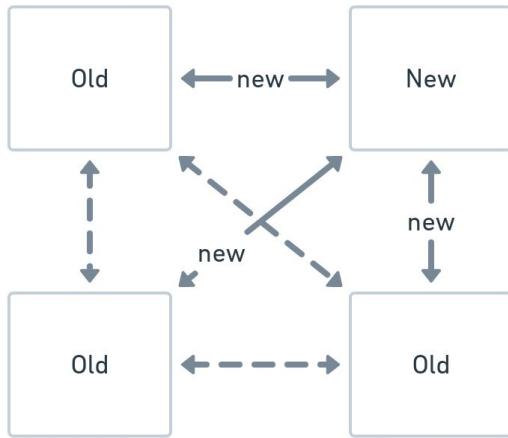
Scaling for Read

Candidate:

Yeah, that's a good question. I'm assuming we're using the CRDT data structure and assuming each node stores each count on disk. From our previous discussion, the read throughput is 500,000 QPS, which is a lot.

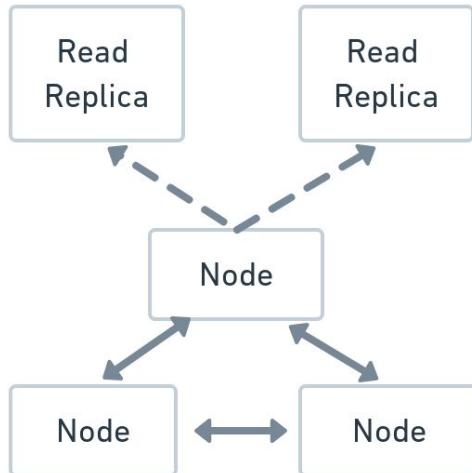
Option 1: Add More Nodes to CRDT

We can shard the databases even further so we have more nodes to read from. The issue with CRDT is the more nodes we add, the more fan-out is needed, since each machine has to push to other machines.



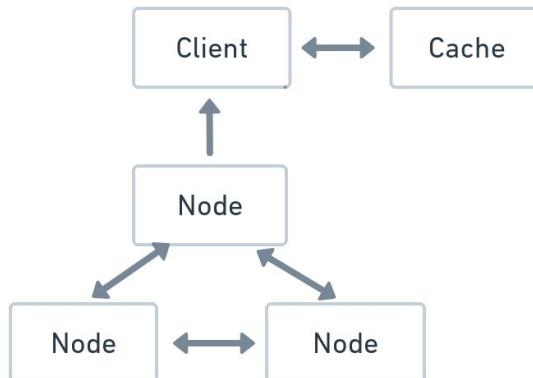
Option 2: Replicate to Read Replicas

For each node, you can replicate to a read replica to increase the read throughput. The advantage is redundancy, so when a node goes down, the system can rebuild the node count. Also, each read replica can scale for read QPS. The downside is the read replica will be eventual-consistent if we use asynchronous replication. That's ok because CRDT is already eventual consistent.



Option 3: Read-Through Cache

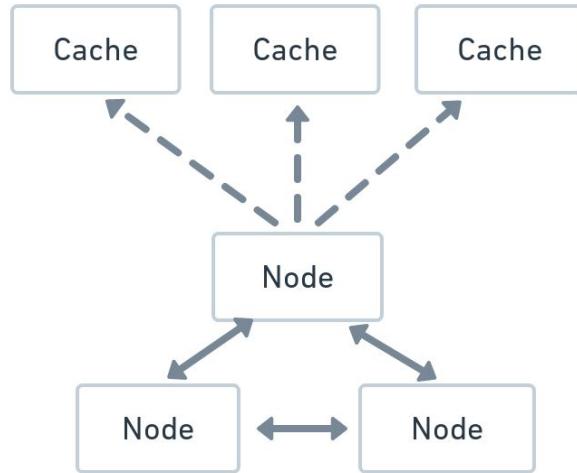
We can have a read-through cache, and upon a cache miss, we can retrieve the data from the node and update the cache.



The advantage of this is we can scale the high number of reads through cache replicas. However, the major disadvantage to this is that cache invalidation is very difficult since the count is constantly being updated. You pretty much need to invalidate the cache every sub-second.

Option 4: Periodic Update to Cache

For systems like this, we can't afford for the cache to be cold. Instead of read-through, we will attempt to update the cache with the latest counts periodically.



The upside is we'll be able to scale well with the number of caches. The downside is the periodic update will cause some delay. But it is something we can adjust to fit our needs. If the cache goes down, we can warm up the cache on the next update.

Conclusion

Since we are fine with eventual consistency as discussed in the non-functional requirement, I would go with option 4. I would do option 2 for redundancy but not to scale for read throughput. We can just scale up the cache cluster for that.

Interviewer:

Ok, that was some deep technical discussion. We are out of time, and thanks for your time!

Cloud File Storage

Interviewer:

Please design file storage where users can store and organize the files into folders.

Step 1: Gather Requirements

Functional Requirement

Candidate:

I'm assuming users need this to store their files like they do in their local file system but instead, they want to store the files in the cloud for durability. Is that correct, or are there other pain points we are trying to solve?

Interviewer:

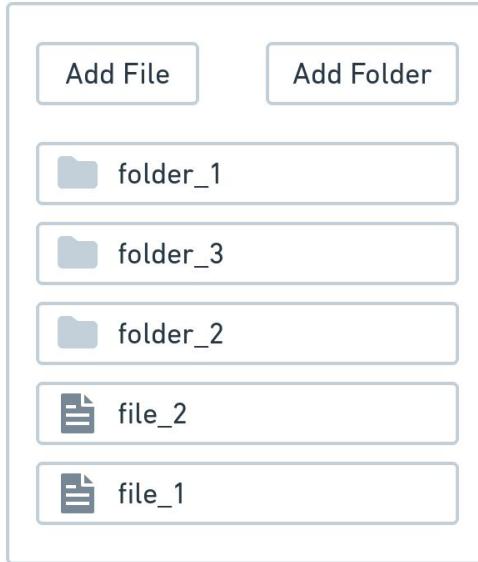
Right, another use case is the ability to work with the same set of files no matter where you are.

Candidate:

Right, that makes sense. So the user can upload files into folders and create folders within folders. I do have a couple of questions. What kind of files should we support? Do we need to display thumbnails if it's an image or video? Should we support features like search, and the labeling of files? How should the dashboard be displayed to the user?

Interviewer:

For now, we support any file and assume it to be safe. We can worry about security later on. Don't worry about thumbnails, and we don't need to support search and labeling for now. The dashboard looks similar to the Google Drive list view, something like this:



The folders are always shown first, followed by the files.

Candidate:

Do I need to support sorting by name or other attributes based on this wireframe? Also, do I need to worry about the user experience once they click into a file?

Interviewer:

No, just worry about displaying the folders, files, add files, and add folders. Don't worry about clicking into a file, but you need to think about when they click into a folder. Initially, you should just worry about adding files. We can focus on file modification later.

Candidate:

Ok, last question, should we think about permission and the ability to share the folders and files with other people, or can users only view their files?

Interviewer:

Don't worry about any of that. We can address them if time allows.

Non-Functional Requirement

Candidate:

Ok, thanks. How many daily active users are there, and what can I assume about the distribution of the user base? How much memory does each user

have to store the files? How many folders and files can I expect within a folder?

Interviewer:

For now, let's just worry about users in North America, and we have 100 million daily active users. Each user can have up to 50 GB, and you don't need to worry about pricing tiers for more storage. It wouldn't be unlikely to see thousands of folders and files within a folder.

Candidate:

What can I assume about the file sizes?

Interviewer:

Let's assume the files could be as big as 1 GB. The distribution is very specific to the file type. Some can be as small as 10 KB, but let's assume 1 KB to 1 MB is common.

Candidate:

Can I assume we will have a high read-to-write ratio? Some numbers like 20 to 1 make sense because you have to visit the page every time you upload a file or folder.

Interviewer:

Yeah, that's fine with me.

Candidate:

Ok, I'm going to assume it's highly important to store exactly what they send us, and it would be a terrible experience if there's corruption in the file. I'm also going to assume consistency is very important if the user has other devices. If I change one file in one device, it should show up on the other devices. I'm also assuming durability is highly important since losing their file would be a really bad experience. What about latency to fetch for a folder's content? I would think it needs to be relatively fast with about 300 - 400 ms for p99. Does that sound fine?

Interviewer:

Yeah, latency is important, you don't want the users to wait when they click into a folder.

Candidate:

Ok great, makes sense to me.

Step 2: Define API

Candidate:

Here are the APIs I think I need:

```
add_folder(user_id, folder_id, name) → status  
add_file(user_id, folder_id, file_bytes) → status  
view_folder_items(user_id, folder_id) → [folder_item]  
download_file(user_id, file_url) → file_bytes
```

For `add_folder`, the user creates a folder in a `folder_id` with a name.

For `add_file`, the user passes the file into the folder; for now we will grab the file name for the name to be displayed.

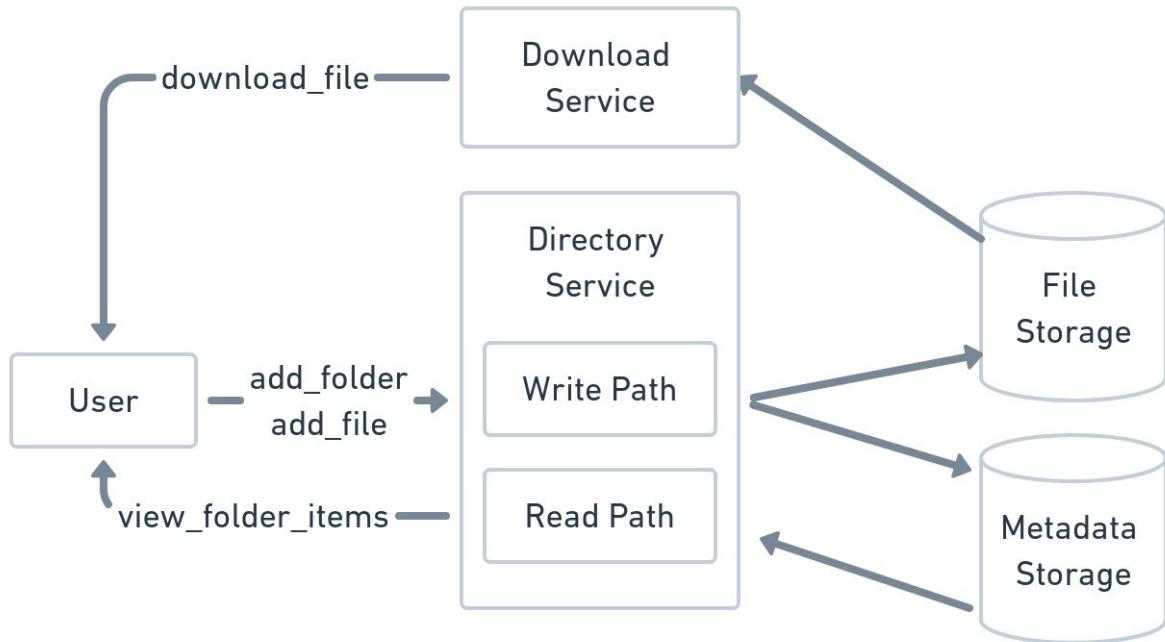
For `view_folder_items`, when the user initially visits the dashboard, it will show the files for the home folder. When they click into a folder, it will call `view_folder_items` with the clicked `folder_id`. For each `folder_item`, it is either a file or a `folder_id`.

For `download_file`, we will pass the `file_id`.

Step 3: Define High-Level Diagram

Candidate:

So those are the APIs, let me think about the high-level diagrams.



It is quite simple with a light directory service to pass through the requests. We may think about bringing the file storage closer to the user, but we can talk about that later.

Step 4: Schema and Data Structures

File Storage: File Bytes Table

File ID	File Bytes
---------	------------

Metadata Storage: Folder Table

User ID	Folder ID	Parent Folder ID	Created At	Folder Name
---------	-----------	------------------	------------	-------------

Metadata Storage: File Table

User ID	File ID	Parent Folder ID	File Name	Created At	File URL
---------	---------	------------------	-----------	------------	----------

Step 5: Summarize End to End Flow

Candidate:

Now that I have the high-level diagram and schema, let me go through the flows to discover some of the challenges.

For **add_folder**, it calls the directory service and writes to the metadata storage to create a record of the folder table.

For `add_file`, it passes in the bytes and the `folder_id`. It will first persist to the file blob store, where the store will generate a `file_url`, then pass the URL and save it in the file table.

For `view_folder_items`, it will pass the `folder_id` of interest. Then the directory service will fetch a list of folders and files with the `parent_folder_id`. Now that I think about it, we will need to paginate the results with an offset. The paginated chunk can just be a constant. We will sort it by `created_at timestamp` to paginate. For now, we will use server generated timestamp for file and folder.

`view_folder_items(user_id, folder_id, offset) → [folder_item]`

For `download_file`, we will use the `file_url` to download the bytes from the download service, a passthrough to the file blob storage.

I think this should address all the features. I have a couple of topics I want to talk about, is there any particular direction you would like me to go in?

Step 6: Deep Dives

Candidate:

The high-level diagram is pretty straightforward, but I think the challenges are going to be the following:

1. Schema design, indexing, transaction.
2. When users change files between different sessions, how do you keep the files in sync?
3. Some files are big. If there's a partial change to the file, how do you optimize for bandwidth?
4. How do you scale for the databases since it needs strong consistency?

Is there one you would like me to talk about?

Interviewer:

No, that list seems solid. Go right ahead. I'm going to add a new minor requirement regarding transactions. Think about how you would support adding multiple files or folders in one request.

Normalized and Denormalized

Candidate:

I just want to bring up that since reading is important, we can consider another schema.

Option 1: Normalized Schema

This is what I have with the folder and file table. There's a slight inefficiency where I need to union the two tables together before making the query.

Option 2: Denormalized Schema

Instead, I can create another table called "folder file table," a unioned table between the file and folder table.

Item ID	Type	Parent Folder ID	Created At	Name	File URL
---------	------	------------------	------------	------	----------

Type is either file or folder, and only file type will have file URL. Otherwise, both entities share similar columns with name and parent_folder_id. The advantage of this is you don't need to union the tables every time the client calls view_folder_items.

Conclusion

To start, I would go with option 1, since the separation of the entities is more clear. However, if performance becomes an issue, we can consider a unioned table like in option 2. However, the entity is strongly coupled to the UI wireframe, so it might not be a good idea from an extensibility standpoint.

Secondary Indexing

Candidate:

The primary keys are file_id and folder_id. Unfortunately, we are querying by parent_folder_id, so I need to speed up the read query.

Option 1: Full Table Scan

We will go through every entry to look for a file or folder that belongs to the right parent folder ID in this option. The downside is that it's slow. The upside is the write will be faster without a secondary index.

Option 2: Secondary Index on Parent Folder ID

In this option, create a secondary index on parent_folder_id on both the folder and file table. When you search for files and folders that belong to a folder, the read will be faster. However, the write will be slower with a secondary index. To do the pagination, we need to sort the result by timestamp and display the top N folder_items.

Option 3: Secondary Index on Parent Folder ID and Timestamp

I will create a secondary index based on (parent_folder_id, create_at) so I can fetch the top folder_item efficiently since created_at is sorted for a given parent_folder_id. Like option 2, write will be slower, and read will be faster given our query pattern of pagination by timestamp.

Conclusion

I will assume the write is significantly less than read because it's more likely people will visit the page to view it, rather than to perform operations on the folder and files. Therefore, I will pick option 3.

Database Choices

Candidate:

As you mentioned, we now want to support adding multiple files at the same time. There are two reasonable product experiences. Imagine a scenario where you upload 10 files and 6 succeed:

Option 1: Roll back the 6 that succeeded.

Option 2: Commit the 6 and ask the client to retry the rest.

Conclusion

As a user, I prefer option 2, so I don't have to retry the 6 successfully committed files.

Interviewer:

Sure! But what if we want to support all-or-nothing with option 1?

Candidate:

Then I want to discuss some of the database options:

Option 1: Use a Wide Column Store

Typically, wide column stores are LSM based databases that are more optimized for writes and don't support cross row transactions as we need here. If a partial row fails, we need to roll back. Also, wide column stores like Cassandra have weak consistency with leaderless replication where conflicts will result in lossy writes. I don't think this is a good idea for what we're trying to achieve here.

Option 2: Use a Document Store

It will be difficult to do a union between the file and folder tables based on parent_folder_id, since parent_folder_id will be modeled as part of the blob. Generally, a document store doesn't support join and union. You will have to do it on the application level.

Option 3: Use a Database with Strong Transaction Support

Because we're assuming the read-to-write ratio is high, using a B-Tree based database seems like a good choice. We should also consider using a strong transactional support relational database since we need to support cross-row transaction support. Also, we need to join between the file and folder tables. The above properties make relational databases like MySQL a good choice.

Conclusion

It looks like a strong transactional database that supports join and union like MySQL is the obvious choice here.

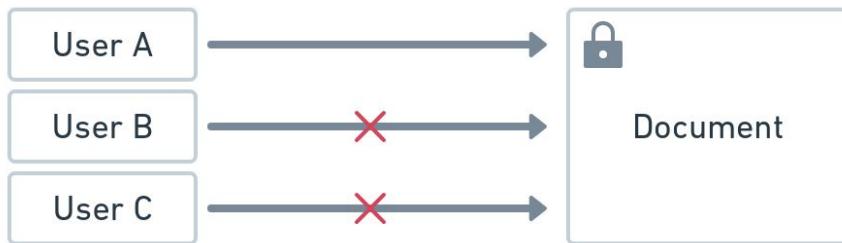
Out of Sync Files

Candidate:

Since we are supporting this for multiple sessions and potentially for multiple users accessing and modifying the same file simultaneously, there's a concurrency challenge that you have to deal with. There are multiple options, so let's go through them:

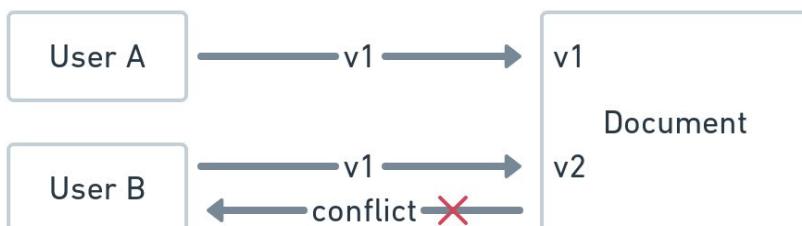
Option 1: Pessimistic Lock on the File

Whenever a user session accesses a file, a lock is acquired for that file, and only that session can access and modify the file. The other sessions have to wait for the user to release the lock by closing the file. The upside is there's no chance of concurrent modification, but the huge downside is only one person can modify at a time. I'm doubtful this would be a good user experience.



Option 2: Optimistic Lock on the File

Whenever user A views and edits a file, a version number is acquired for that file. User A can modify and save the file. If there's another user, B, who edits the file and saves it before user A, user A's save will fail, and the system will ask user A to try again with the latest version. The upside is concurrency is solved, and multiple users can modify the files at the same time. The downside is it'll be frustrating if the user has to restart the whole process again. In a highly concurrent environment, this can lead to a lot of retries.



Option 3: Display Both Versions

When there's a file conflict, display both versions to both the users. Then the system allows them to choose which version to use going forward. The advantage is the system keeps both versions, but the user experience will be confusing. As more conflicts happen, the divergence can branch out more, leading to complexity and confusion.

Conclusion

I would choose option 2 here with an additional feature of displaying the conflict and require the second user to resolve with the conflicted copy. An example message would be, "Another user has already updated the file, here are the conflicts and please try again."

Interviewer:

Ok, that sounds fine to me.

File Upload

Candidate:

One of the things that I think is worth addressing is how files are passed from the client to the server since there can be a lot of bytes to pass through. For the add operation, you will still have to pass all the bytes. Where it gets interesting is file modification. Should I dig into that?

Interviewer:

Sure!

Candidate:

Ok, imagine I have a file, and I modify it. How do I communicate that change, assuming the file is non-trivially big?

Option 1: Pass the Whole File

In this option, I will pass the whole file every time the document is changed. Passing the whole file has the advantage of simplicity but at the cost of necessarily passing unchanged bytes.

Option 2: Pass the Chunks

A famous algorithm called rsync is used in Linux to transfer a file from the client to the server, where you only pass in the new chunks. The pro is that if the files are similar, they don't need to pass many bytes. If the files are completely different, it introduces the overhead of checking the checksum.

Reminder

We discussed the rsync algorithm in detail in the toolbox chapter. Rsync is just one way of determining a difference between two files. Sometimes applications have their diffing algorithm based on their specific use case.

Option 3: Lossless Compression

I'm not an expert, but depending on the file type, there is lossless compression that can be applied to reduce the bytes transferred. For example, you can apply run-length encoding for lossless compression to reduce image files. The upside is there are fewer bytes to be passed at the expense of encoding and decoding with a call.

Conclusion

I will include option 3 if the compression ratio is good without too much compression CPU overhead. Since I expect the files to be modified instead of complete new copies, I would use option 2 as I expect better performance. We can conduct A/B testing to test the performance difference for the diffing algorithm and compression to see which one is better in practice.

Interviewer:

Ok, that sounds fine, I learned something new today. I usually use scp but will consider rsync now. Can you think about the scalability of your database since you're using a relational database?

Scale It Up

Candidate:

Ok, let me do some math to see if scalability is an issue.

We are assuming 100 million daily active users. Let's say each user visits the site two times a day. I will assume a 1.5 peak factor as I don't think Cloud Storage will see bursty traffic.

$$1 \times 10^8 \times 2 \times 1.5 = 3 \times 10^8 \text{ QPD}$$

$$3 \times 10^8 / 10^5 = 3 \times 10^3$$

The read QPS is 3,000, and the write QPS is a fraction of that. We do expect the traffic to hit the disk since we want consistency. If we assume that each database can handle 300 - 500 QPS, we will need at least 10 machines which means we need to shard. I will just use the folder table to drive the discussion.

Option 1: Handle the Read with Cache

We can consider cache that will help us quickly scale the QPS we're looking at. The big downside is that because we want strong consistency, the extra complexity of cache invalidation and eviction will make it challenging to achieve that with a cache miss and keep the cache and database in sync. Imagine a user reads the cache when the underlying storage has already changed.

Option 2: Shard by Parent Folder ID

The first proposal is to shard by parent_folder_id. The advantage is that you can fetch all the folders in that parent folder in the same shard. The disadvantage is that you might have a hot shard for parent_folder_id 0, which we're assuming to be the home directory.



user id	folder id	parent folder id
1	1	0
3	4	0
3	5	0
4	6	0

user id	folder id	parent folder id
1	2	1
1	3	1
4	7	6
4	8	6

Option 3: Shard by User ID

If you shard by user_id, you will not have a hotspot associated with a particular folder like folder 0. However, you may end up with a hotspot at the superuser's shard if you have a superuser.

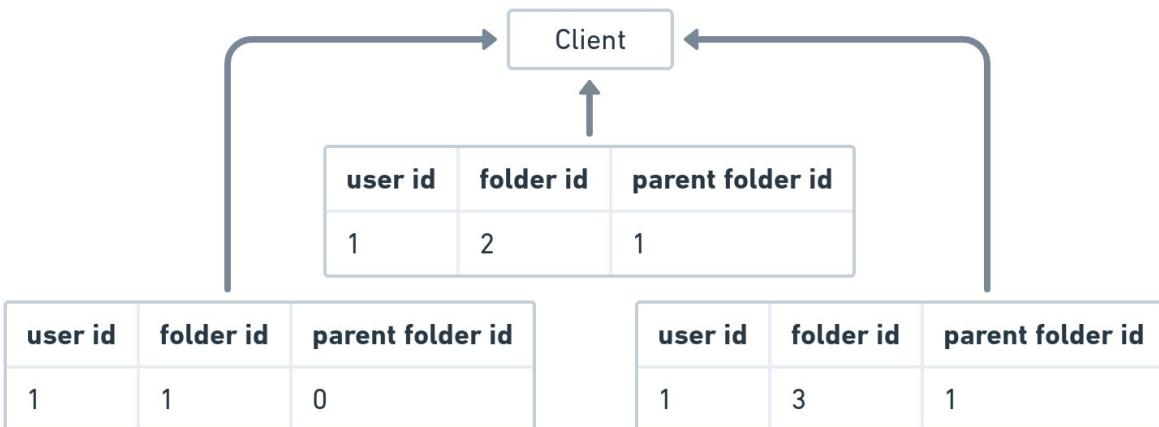


user id	folder id	parent folder id
1	1	0
1	2	1
1	3	1
1	4	1
1	5	1

user id	folder id	parent folder id
3	4	0
3	5	0
4	6	0
4	7	6
4	8	6

Option 4: Shard by Folder ID

You can shard with folder id. The downside is when you fetch for a parent folder ID, you may need to scatter-gather from all the tables. But you wouldn't have a hotspot with a parent folder ID or user.



Option 5: Shard by User ID and Parent Folder ID

Another alternative is to shard by user ID and parent folder ID. When you query for a parent folder ID for a user, it should live on a shard which will be efficient. The potential downside would be if there's a hot user who loves to cram all their files in a particular folder, then you may still have a

hotspot, but this is less likely to be the case. Also, in the future, if you want to fetch multiple folders for a user, you may have to scatter-gather.

user id	folder id	parent folder id
1	1	0

user id	folder id	parent folder id
1	2	1
1	3	1

Conclusion

I am worried about sharding by parent_folder_id since it's a realistic use case where people cram all their files into the home directory folder. I am worried about power users as well. I will mitigate it with option 5 to shard by (user_id, parent_folder_id) to distribute a power user's load across multiple nodes. We will monitor if there's a (user_id, parent_folder_id) hotspot where the power user puts most of their files into a single folder.

Reminder

In a real interview, you may not need to come up with all these sharding algorithms. You can pick 2 or 3 and articulate the trade-offs and what the future might look like. As you can see, there's no perfect solution, and it depends on what your query pattern looks like.

Interviewer:

Cool, that's a lot of options. Thanks for going through them. I have one more question for you, what happens when a user deletes a folder?

Folder Deletion

Candidate:

Interesting, what is the expected user experience here? Is it more like an archive that the users can re-enable, or is it a recursive delete where the application files permanently remove the files?

Interviewer:

Good questions. Let's assume we want to remove the files within the folder permanently.

Candidate:

Ok, I see. Let me think of a couple of options:

Option 1: Recursively and Synchronously Remove All Files and Folders

In this option, when the user clicks “delete” on a folder, we immediately acquire a lock on all the folder and file rows and invoke DELETE operation to remove the rows. The upside is that it’s synchronous and reflective right away. The downside is if the folder is large, this operation will be unpredictably long, time out, and potentially take the database down. Also, if the user regrets it, the files are already gone.

Option 2: Mark and Sweep

We can call a new column called “status.”

Folder Table

User ID	Folder ID	Parent Folder ID	Folder Name	Status
---------	-----------	------------------	-------------	--------

We can introduce a status column where deleted files are TRASH and active files are ACTIVE. This way, it’s a soft delete, which enables more options:

1. If we wish to delete, we can schedule to delete at a lower traffic time.
2. We can keep the folder in a trash folder in case the user regrets it.
3. We can keep the deleted folder forever as more like an archive.

The downside would be that we need to store longer than necessary if the user wishes to delete right away.

Interviewer:

That was a comprehensive discussion, thanks. We are out of time. Have a great day!

Rate Limiter

Interviewer:

Please design a rate limiter.

Step 1: Gather Requirements

Functional Requirement

Candidate:

Ok, wow, there are so many different possibilities. Let me think, what is the purpose of this rate limiting?

Interviewer:

Can you think of some use cases?

Candidate:

Generally speaking, a rate limiter prevents a client from overwhelming a downstream system maliciously or accidentally. Without it, sometimes a malicious DDoS can bring the services down. Sometimes we may have a budget allocated to a specific client where they can not go over a certain number of calls within a time frame. Also, an unexpected thundering herd may bring the service down. We will need to drop the request rather than cause a cascading effect to the downstream service. Is there a particular use case we're designing for?

Interviewer:

Let's design for an external client who has a budget with us such that they can't call us more than a certain number of times within a specific timeframe. We are limiting them on the customer level. It doesn't matter if it's a different API.

Candidate:

Ok, what happens if they go over the limit?

Interviewer:

Good question. I'll leave that up to you to decide.

Non-Functional Requirement

Candidate:

How many customers are we serving here? How many requests do we anticipate getting from all the customers? Do we expect a spike in traffic?

Interviewer:

Let's say we have 10 million customers, and they will on average make 1,000 calls to our service per day. We don't know anything about their traffic pattern. We should anticipate anything.

Candidate:

Ok, sounds fine. What's the latency requirement for this rate limiter? Is it protecting against a latency-sensitive API?

Interviewer:

Let's just say the use case is a client invoking an asynchronous job, so latency is not that sensitive. The API has a p50 of 500 ms.

Candidate:

Ok, what about the accuracy of the result? If we limit to 10 calls per minute and the requests call it 11 times within a minute, what would be the implication for the business?

Interviewer:

That wouldn't be too bad, as long as they remain within that limit over a longer time horizon. We should still attempt to be accurate because we'll be paying out of our own pockets if we process too many or make the customers angry if we process too little.

Candidate:

So it sounds like latency isn't important, accuracy should be high over the longer time horizon, but the short-term is ok. I'm going to assume we have some leeway for availability since it's an asynchronous job. Durability is also important because long-term accuracy is important.

Interviewer:

Yup! That sounds right to me. We should still complete the asynchronous job as soon as possible if we can.

Step 2: Define API

Candidate:

Based on the requirement, I just have the following API:

invoke_api(customer_id) → status

There's only one API because that's the asynchronous API we're trying to throttle. The returned status will be ALLOW or DENY.

Step 3: Define High-Level Diagram

Candidate:

Here's the high-level diagram:



Step 4: Schema and Data Structures

[Skip this section, nothing to talk about]

Step 5: Summarize End to End Flow

Candidate:

The user invokes the API and gets rate limited before going to the request queue.

Step 6: Deep Dives

Candidate:

The diagram is pretty simple, so I'm going to dig deeper into the algorithm. First, let's talk about the architecture of the rate limiter.

Rate Limiter Product Design

Candidate:

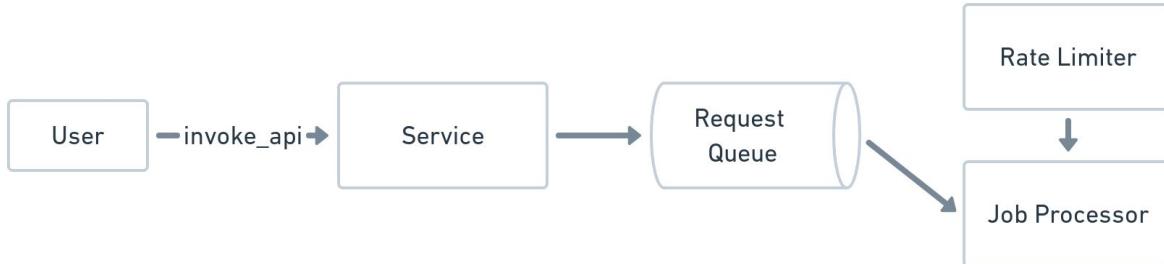
Since this rate limiter is for an asynchronous workflow, I have a couple of options:

Option 1: Throttle on the API Level

Like the high-level diagram, if the user goes over the limit, we will reject their API and ask them to retry again.

Option 2: Throttle After Request Queue

Another way we can handle it is to put the rate limiter after the request queue to throttle the requests. If it goes over, we will stop processing until it is ready to be processed again.



Conclusion

The user experience is a little different. Given this rate limiter's intention to ensure customers don't go over the budget, we can throttle their event processing instead of rejecting them and asking them to retry again. However, the contract needs to be clear to the customer, otherwise, we might be queuing many jobs that they expect to run immediately. Also, we need to monitor the backlog size and start rejecting the real-time request if the backlog becomes too big.

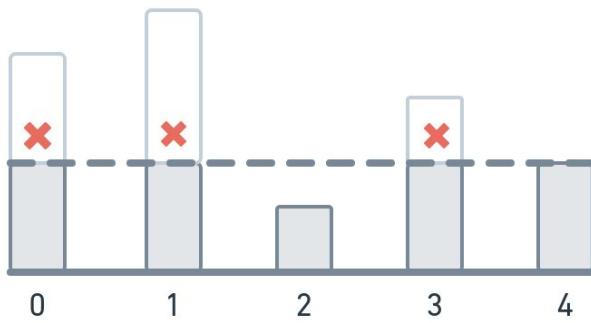
Rate Limiter Algorithm

Candidate:

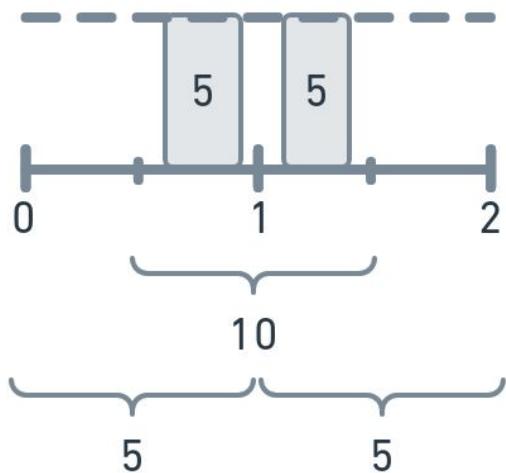
Option 1: Fixed Window

In the fixed window, each bucket maintains a count. If it crosses the threshold, it will drop the request. Once the period ends, the count restarts

from 0 again.



The advantage of this approach is it is simple and requires very little memory since you just need to maintain a count for the given window. The downside is inaccuracy when the traffic happens towards the end of the previous interval and the beginning of the new interval, creating an effective over-the-limit interval.



Like the image above, assume the limit is 5. From 0 to 1 and 1 to 2, both are within the limit. From 0.5 to 1.5, it's above the limit.

Option 2: Sliding Window

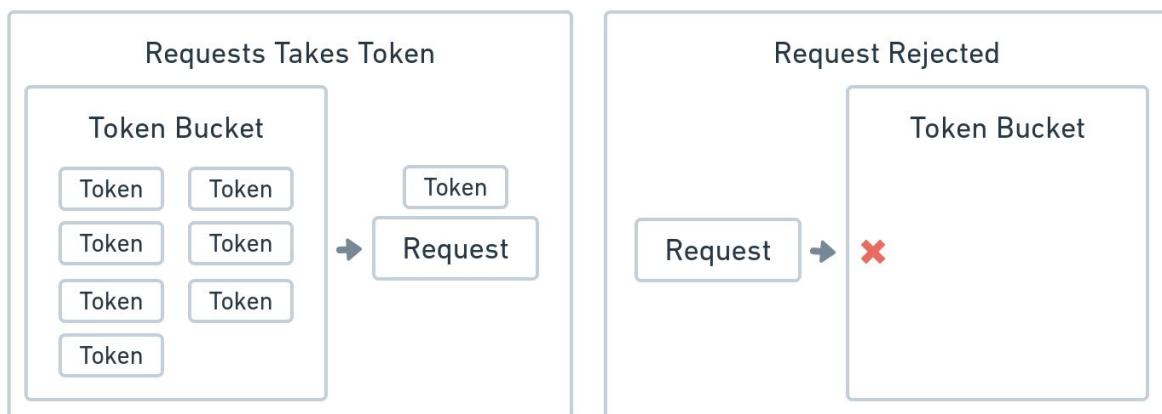
In this option, we will keep track of the timestamp of the requests. When a new request comes, update the current window such that the window maintains the interval of interest and counts the number of requests within that window.



In the above example, we are allowing 3 requests for 5 seconds. The advantage of this algorithm is accuracy since we're keeping track of all the timestamps. The disadvantage is that it is memory intensive to keep track of all the events.

Option 3: Token Bucket

In the token bucket algorithm, tokens are added to the bucket periodically. If the request wants to go through the rate limiter, it needs to acquire a token.



The system fills the token bucket periodically. For example, if the rate is 5 requests per minute, one possible way to refill is to refill up to 5 tokens every minute. The advantage is the simplicity of understanding a request requires a token to proceed. It also has the flexibility to adjust how we want to refill the tokens. Depending on the refilling algorithm, it can also run into the same inaccuracy issue as a fixed window. Imagine a bunch of requests consuming all the tokens only before the refill happens and immediately consuming everything again after the refill finishes.

Reminder

Depending on the token bucket refill algorithm, it can be very similar to a fixed window. Imagine you refill X tokens every Y minute up to X tokens. Algorithmically, it's effectively the same as a fixed window of X requests per Y minute. However, it's easier to reason about ad-hoc token refills than to adjust the fixed-window.

Conclusion

Even though the token bucket can be more intuitive and flexible, a fixed window is simpler to implement without having a periodic job to refill the bucket. I would go with the fixed window to start with.

Interviewer:

Ok, that sounds fine to me. Can you tell me how you would handle the situation when the rate limiter fails?

Rate Limiter Failure Scenario

Candidate:

Yeah, sure! When the rate limiter service fails, we cannot determine if we should let the request through.

Option 1: Fail to Close

Since we're designing this rate limiter to handle the processing of an asynchronous backlog when the rate limiter fails, and we fail to close, we won't determine their true rate, so we stop the processing. The upside is we won't be letting more requests through than they should because we're

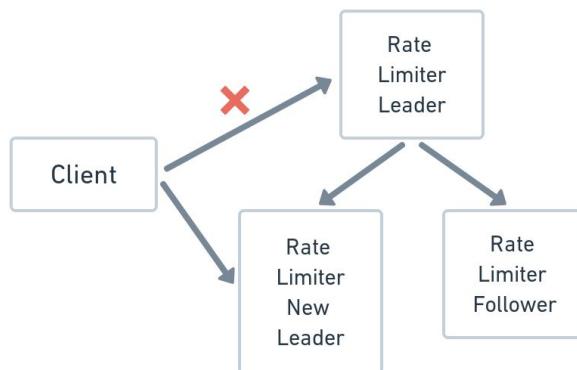
unable to calculate the rate. However, the major downside is the backlog will continue to accumulate, and the workers are just waiting unused.

Option 2: Fail to Open

Instead of terminating the pipeline when the distributed rate limiter is down, perhaps we can have a default rate limiter constant in each job processor machine such that we continue to drain. The upside is we will continue to drain the pipeline, but the downside is that we might be letting some customers through more or less often than their true rate. We also need to be careful with the constant. If it's too high, we might overwhelm our system.

Option 3: Leader Follower

One approach I can take is replicating the data written to the rate limiter leader into a rate limiter follower. When the leader goes down, we can perform a leader election to pick a new leader, preferably with the most updated data. We can perform asynchronous replication because having some replication lag and slight inaccuracy is acceptable.

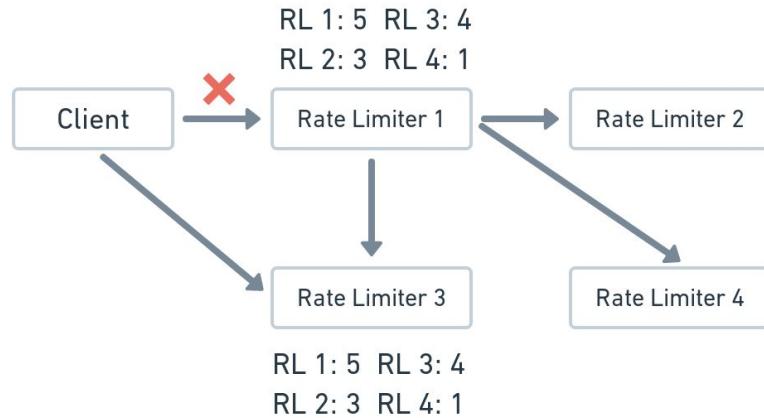


The interesting thing about rate limiting is that every request is also a write since you need to increment the count with every request. So you can't scale with read replicas. The advantage of a leader-follower is you can fall back to a new leader, but the leader election takes time, so there'll be a temporary stop.

Option 4: CRDT

I would use a CRDT data structure where the rate-limiter randomly assigns a request to a node, and each will keep track of its count. Then it will broadcast that count to other nodes. When calculating, you can sum up the

total number of requests for the same time frame. We will use an asynchronous broadcast so we can expect the numbers to be delayed and eventually consistent.



If any of the nodes go down, other nodes can still continue to take requests. We just have to be careful with a cascading effect since the other nodes will have to take more traffic on behalf of the failed rate limiter. This has a much stronger available guarantee since any node can take the traffic. The downside is the extra complexity and overhead of broadcasting.

Conclusion

I would go with a combination of options 2 and 3.

For option 2, we need to ensure a backlog config is a number that is small enough that we don't overwhelm our system. I understand there's more complexity and less accuracy but given accuracy isn't a strong requirement, I would prefer to utilize our worker when the rate limiter is down.

It's nice for CRDT to be much more available than leader-follower, but it has a higher broadcast overhead. We don't expect the leader to be down often, and when it's down, we will pick a new one in a couple of seconds through the leader election. Since the job allows for some downtime because they're asynchronous jobs, let's punt on CRDT for now.

Interviewer:

That's a creative idea, but as you said, we need to be careful about the implication of letting more or less thorough than we're supposed to. What else would you think about regarding your design?

Data Storage Long Time Horizon

Candidate:

I think it's worth talking about scaling this since rate limiting can have a pretty high write throughput. From my requirement gathering, we're assuming 10 million customers each with 1,000 calls a day. Let's assume a peak factor of 2. Let me calculate the QPS we have to scale for:

$$\begin{aligned}1 \times 10^7 \times 1 \times 10^3 \times 2 &= 2 \times 10^{10} \text{ QPD} \\2 \times 10^{10} / 100\text{k} &= 2 \times 10^5 \text{ QPS} = 200,000 \text{ QPS}\end{aligned}$$

200,000 QPS is quite significant. Let me think about the data storage solution. Before digging deeper, what is the time frame that we're limiting the customers to?

Interviewer:

Since some of their requests may be cyclical, we don't want to throttle them with a short time frame, but at the same time, we want to prevent a huge spike of requests. For now, let's design this in the time frame of an hour with requests per hour.

Candidate:

Ok, here's what I am thinking:

Option 1: Store it in the Database

When the request comes, I can increment the counter in the database with an atomic increment operation. The advantage of doing so is durability since we need to keep track of the count for an hour. If we keep it in memory and it goes down, we will lose all the data within the hour. However, the downside is the throughput challenge. We'll need quite a few machines with 200,000 QPS. Perhaps we can optimize using a write-optimized database like Cassandra with the trade-off of consistency and accuracy. Even then, 200,000 is still a lot of requests per second.

Option 2: Store it in the Cache

Instead of storing it in the database, we can also store the counter in the cache with an atomic increment operation. We can expect to scale much easier with a cache solution where we expect each to handle at least 50k to 100k QPS. However, durability is a major concern since we need to keep track of an hour's duration.

Conclusion

It looks like we need some sort of durability solution anyway because if we're 30 minutes in and the cache crashes, we will lose a significant amount of data during that 30-minute time frame. Still, we also like the performance of the cache. One solution I can think of is a write-back cache where we write to the cache first, and the data is either asynchronously backed up or replicated to a durable store in case of failure. We are okay with some data loss due to replication lag, which means we have a little leeway for accuracy.

Data Storage Short Time Horizon

Interviewer:

Ok, then what if we are designing for a short time horizon like requests per second?

Candidate:

In that case, I will use a cache without the extra complexity of a durable backup or store. If there's a cache failure, by the time I warm up the cache with the backup, the data is already outdated anyway.

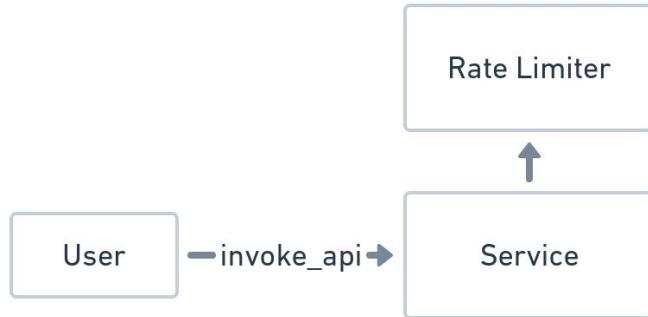
Client Facing Latency Sensitive Rate Limiting

Interviewer:

Ok, I like how you're adaptive to the changing requirements. Ok, so far, we've been talking about rate-limiting in the context of asynchronous processing. What if we want to design a client-facing and latency-sensitive rate limiting. For latency, think 15 ms for p99.

Candidate:

Ok, this is a major change in requirement. 15 ms p99 is indeed latency-sensitive. Let me rethink about the high-level diagram:

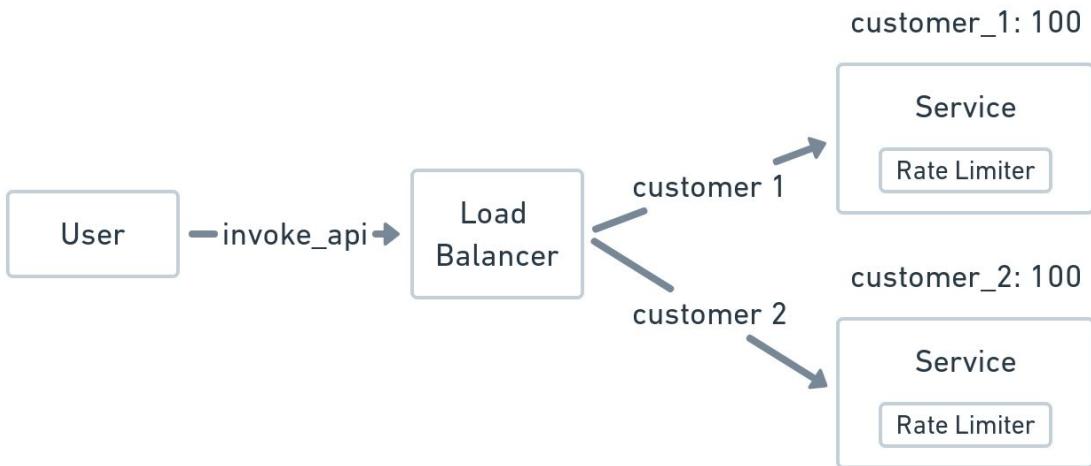


Option 1: Have a Distributed Cache

I'm going to assume I have to use a cache since each read to disk is at least 2 ms to 5 ms, if not more. 2 ms to 5 ms is a significant percentage of the 15 ms requirement. By having a distributed cache, the latency is about 1 ms to make an inter-data center call. All the network calls will cause a non-trivial hit to the latency.

Option 2: Instance Level Rate Limiter

Instead of calling a distributed rate limiter, we could conduct the rate-limiting on the app servers themselves, so no network call is needed. The issue is that the services themselves are now stateful. The load balancer needs to remember which instance has which customer's count.

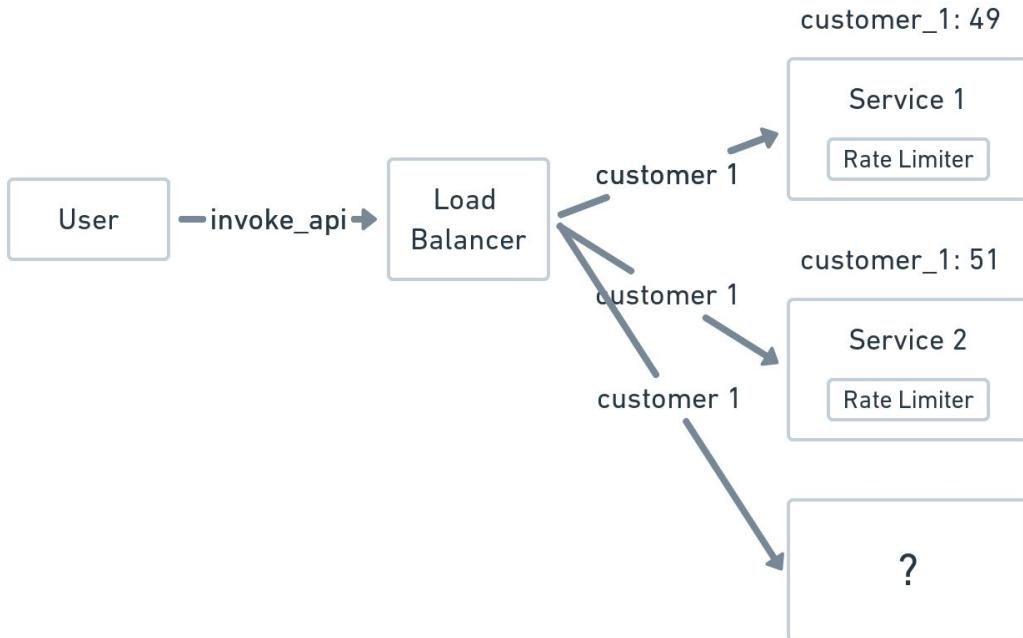


Because we're associating a stateful rate limiter with a stateless service, the stateless services are now stateful. So if a service goes down, instead of hitting another duplicate instance through load balancer round robin, we

have to wait to initiate the service for a specific customer. Availability suffers a bit here.

Option 3: Stateless Instances Through Approximation

Instead of stateful routing, we can continue to have a stateless load balancer.



When a request hits a service, we will be getting a subset of counts. Each instance will adjust the limit based on the number of healthy hosts available.

Instance Rate Limit = Global Rate Limit / # of Healthy Hosts per Second.

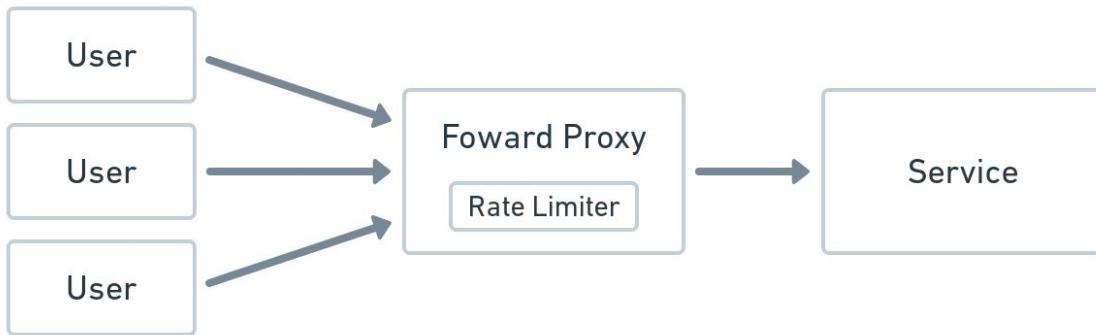
For example, if there are 2 healthy hosts and we have a limit of 100 requests per second, each healthy host will throttle on 50 requests per second.

This algorithm leads to some inaccuracy. For example, if we're limiting 100 requests per second, when we hit Service 1 in the diagram with 49, we will let it through even though it already has 100 requests. Also, if we add or remove a server, the math will temporarily be off. To mitigate, we may ask a cluster manager for the latest live hosts periodically, but of course, there

will be some inconsistency depending on that frequency. The major advantage is the servers are stateless, and we can host the limiters on the app servers.

Option 4: Forward Proxy on Client Side

Another option that is unlikely but worth thinking about is letting the clients throttle. The obvious downside is they can bypass the rate-limiting since we are not in control, and it's much more difficult to have the clients make the change. And if there are multiple client instances, the network call to a centralized forward proxy may also add latency.



Conclusion

This one is indeed quite challenging due to the low latency requirement. A distributed cache is the simplest, but I'm worried about the round trip time for latency, even if it's in the same data center. I think I would go with option 1, and adjust the SLA with the customer. If it's a no-go, we can consider option 3 if we are ok with occasional inaccuracy and very strict about the latency. I'm not a fan of option 2 because rate limiting changes the stateful design of the app servers when they should be independent.

Interviewer:

Ok, that was indeed quite technically challenging. Thanks for brainstorming, we are out of time and thanks for coming in!

Chat Application

Interviewer:

Can you design a chat application?

Step 1: Gather Requirements

Functional Requirement

Candidate:

What is the purpose of this chat application? What do people use it for?

Interviewer:

These are public chats where people can talk about any topic, including system design, sports, knitting, etc.

Candidate:

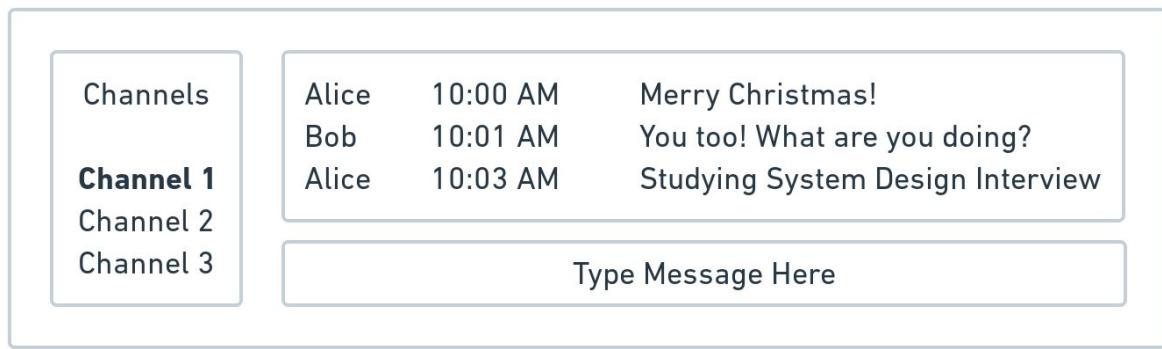
Are we supporting group chat or one on one chat?

Interviewer:

Let's say we're designing a group chat.

Candidate:

Just to help understand the APIs I might need later, let me draw out what I think the user experience should look like:



And the user can scroll up to see the historical messages. Should I worry about indicating whether the message was read?

Interviewer:

The wireframe looks fine, and don't worry about whether the message was read or not yet.

Candidate:

Sounds good. Should we worry about sending images and videos? I know some chats support permanent storage of the messages and some don't. I'm going to assume permanent storage is critically important here for collaboration.

Interviewer:

You are right. All those features are important for collaboration, but for now, don't worry about images and videos. We are interested in storing the messages permanently, though.

Candidate:

Ok, let me confirm a couple more features. Should we worry about modifying and deleting a message, and should we worry about online and offline status?

Interviewer:

Nope, don't worry about those for now.

Non-Functional Requirement

Candidate:

Ok, thanks for answering. How many users do we have, and how are they distributed across the world? How many channels are there? What's the distribution of the users across the channels? What's the maximum number of users that can be in a channel?

Interviewer:

Let's support 100 million daily active users and assume we have users across the world, but most users are in North America. There are about 100,000 channels. The distribution is a long tail where there can be thousands of people in a few popular channels. The maximum number of users is 5,000.

Candidate:

Do we expect bursty send message patterns and excessive message sizes that could potentially put our systems under pressure?

Interviewer:

Yes, since most users are in North America, we expect most people to chat about their hobbies at night. We will limit the number of characters to 50,000 for a message.

Candidate:

What's the latency requirement for sending the message? How quickly do we expect to deliver the message to all the other users? I would expect it should be low for a chat to be interactive. What about the loading time of messages for a given channel?

Interviewer:

That's right, let's target less than 100 ms from one user to another. For message loading time, around 200 ms p95 is acceptable.

Candidate:

Ok, this might be obvious, but I'm assuming we have to deliver exactly what the original user sends. The API to send a message should be available. Otherwise, it's a frustrating experience. The chat messages should be durable since people may look at the history.

Interviewer:

That's right.

Step 2: Define API

Candidate:

Based on the requirement, I have the following APIs:

```
send_message(user_id, message, channel_id) → status  
read_messages(user_id, channel_id, offset) → [message]  
receive_message(channel_id, message)
```

For `send_message`, a user sends a message to a particular user, and it will return whether it went through or not.

For `read_messages`, we will retrieve a list of messages back for a `channel_id`. Since the user can scroll up for historical messages, I will assume we need pagination with an offset. The paginated page size will be fixed, so the client doesn't need to pass it.

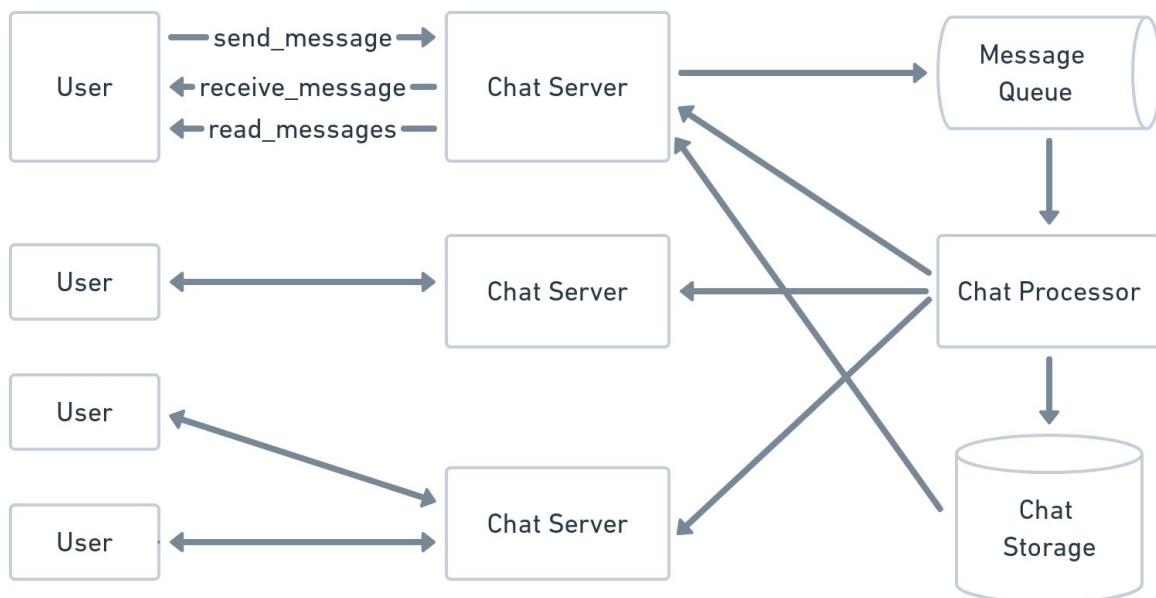
For `receive_message`, it is a call back function upon receiving a message from other users.

A message consists of a written text message, the author, and the timestamp when the user sent it.

Step 3: Define High-Level Diagram

Candidate:

Now that I design the APIs, let me figure out the high-level diagrams. I'm going to assume we need WebSockets to send and receive real-time chat messages. I will add a queue in anticipation of handling high write throughput since it's a chat application. We will dig into them more and modify them if my assumptions are wrong.



Step 4: Schema and Data Structures

Candidate:

So here's my initial take on the schemas:

Message Queue

User ID	Channel ID	Message	Author	Timestamp
---------	------------	---------	--------	-----------

Chat Storage: Message Table

Channel ID	Timestamp	Message	Author
------------	-----------	---------	--------

Chat Storage: Channel Table

Channel ID	Name
------------	------

Step 5: Summarize End to End Flow

Candidate:

When the user initially logs into the chat application, the user calls `read_messages` for a particular default channel from the chat server. The chat server will retrieve that list of messages from the chat storage, filtering for the latest messages for the `channel_id` of interest.

When the user sends a message, the message gets passed to the chat server and forwarded into the message queue. The chat processor will pull from the chat processor and look up which chat servers to forward the message to. Eventually, the recipients will receive the messages through the WebSocket callback function `receive_message`. Maintaining the chat server WebSocket connection lookup is something we can discuss further later if we have time.

Interviewer:

Ok, the flow looks fine to me. What do you like and not like about your design?

Step 6: Deep Dives

Candidate:

Yeah, there are a couple of places I want to consider alternatives and trade-offs, and some aspects I would like to go a little bit deeper too. Here are

some topics I can think of so far:

1. Concurrency of a channel.
2. Storage solution.
3. Efficiency in querying for the latest messages.
4. Real-time data.
5. Scaling for write throughput.

Is there one that you would like me to talk about?

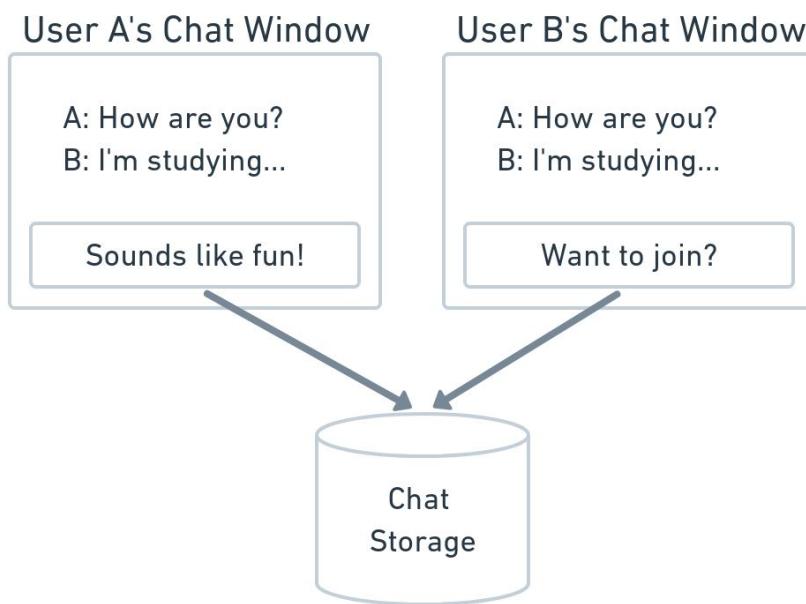
Interviewer:

Nope, feel free to focus on whichever one you think is important.

Concurrency of Channel

Candidate:

You can think of a channel as a shared resource, and multiple writers write to the same channel resource. Luckily, messages don't affect each other. We still have a concurrency problem when two users write simultaneously concerning the ordering of the messages.



In the above example, it's unclear if it should be:

A: How are you?	A: How are you?	A: How are you?
-----------------	-----------------	-----------------

B: I'm studying...	B: I'm studying...	B: I'm studying...
A: Sounds like fun!	B: Want to join?	A/B: Sounds like fun! / Want to join?
B: Want to join?	A: Sounds like fun!	

If we put A's message before B's message, it will be inconsistent when B reloads their dashboard since B saw their message before A's message when they typed "Want to Join?"

Option 1: Store the Ordering Per User

We can store a version number per chat message for each user, and the client-side will generate the versions based on the client-side ordering. Now, that sounds good in theory, but there can be multiple sessions for a given user in practice, which brings up the original concurrency problem. If you store it per session, it's unclear which session is the source of truth. On top of all that, it is storage intensive to store the versioning numbers per user.

Option 2: Optimistic Lock

Optimistic locking is unlikely to be a good solution but is worth going through since it's a concurrency problem. If a writer hits the database first, the second writer will fail, and the server will ask the client to retry again. The advantage is the consistency in the ordering per user. The disadvantage is in a highly concurrent environment, the number of retries will cause it to be almost unusable for a chat application.

Option 3: Pessimistic Lock

Every time someone writes a message, it acquires a lock for the channel, and everyone else can't write a message. Yeah, this will be the worst chat application there is, so no.

Option 4: Use Timestamp as Source of Truth

In options 1 to 3, we're attempting to have a consistent experience per user. What if we relax that and assume a hard refresh with a different ordering is acceptable?

There will be many messages in a highly concurrent environment, so it's unlikely the user will notice the ordering inconsistency. Also, concurrent messages don't have relationships with each other since the users are typing simultaneously. For those reasons, I will assume ordering inconsistency in a highly concurrent environment is acceptable.

With that user experience in mind, we will use the server-generated timestamp as the source of truth. We don't want a client timestamp because it can be tampered with, and there can be clock skew on the client side. There can be clock skew on the server side too, depending on which server generated the timestamp, but we are in control. We can always synchronize the clocks periodically to bring the clocks closer together.

We do want to be careful about the granularity of timestamps. If the granularity is too coarse, like in the minute, we can't tell which messages came first within the minute. For now, I will use milliseconds. If there are conflicts within a millisecond, we're ok with either ordering since it's unlikely there will be meaningful ordering within a millisecond.

Option 5: Use Auto Increment ID

Another possible approach is to use a monotonically increasing unique ID generator like Snowflake. That way, we don't need to worry about the conflict of timestamps. However, a distributed ID generator is roughly ordered, so in the worst case, the ID might sort it a certain way that is inconsistent with the timestamp attached to the message.

Conclusion

Assuming we don't need a strong consistency for each user, options 4 and 5 are reasonable solutions. For this session, I'd like to keep it simple with option 4 since generating a timestamp is more straightforward than having a dependency on a distributed ID generator.

Interviewer:

Yeah, that's fine, I've seen both in production, but I agree with keeping it simple for now. Can you go a little bit deeper into your storage layer?

Database Solution

Candidate:

Ok, let's talk about the database solution I would choose. First, let me assume the write-to-read ratio is high because the read query only happens when the user logs in or visits a channel to retrieve the latest message. Once the user is in a channel, the user is expected to write multiple messages.

Option 1: Standard RDBMS

Let's consider RDBMS like Postgre. Generally, RDBMS is great for transactions and is well understood and supported. The disadvantage is that RDBMS is better for reading because it uses B-Tree instead of LSM. A chat application is write-heavy, so B-Tree is usually not a fit.

Channel ID	Timestamp	Message	Author
------------	-----------	---------	--------

Also, the way we retrieve the data is by a list of recent messages. Given the schema above, retrieving messages for a channel_id won't have good disk locality since data is stored on disk by row and messages for a channel will be stored in multiple rows.

Option 2: Wide Column Leader Follower

I know a wide-column store is generally better for write throughput due to the LSM based append-only property. There's a better disk locality since the system stores the messages in a column family for a channel_id. As far as replication goes, I can choose a leader-follower like HBase. In the case of failure, I can elect a new leader, which may result in temporary unavailability. However, we will have stronger consistency since writes are done on the leader.

Option 3: Wide Column Leaderless

Another wide column solution is to consider a wide column leaderless replication store like Cassandra. With leaderless, we need to worry about conflicts on write. Image two users modify or delete the same message simultaneously. It'll be unclear how to resolve it. However, leaderless

replication has better availability since we need to worry about the leader going down in a leader-follower replication scheme.

Conclusion

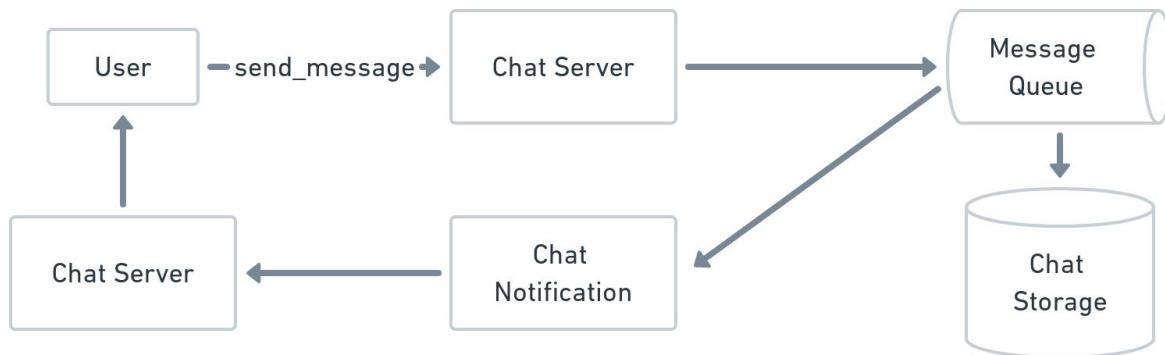
RDBMS sounds like a clear no unless we are operating at a small scale. Given we have a tenth of a billion users, I would like to consider the write performance. Ultimately I would favor option 3 because of better availability. Users can't modify and delete messages in our requirement assumptions, so conflicts aren't an issue yet. In the future, when multiple sessions can modify the same message, then we will need to worry about the complexity of conflicts.

Chat Architecture

Candidate:

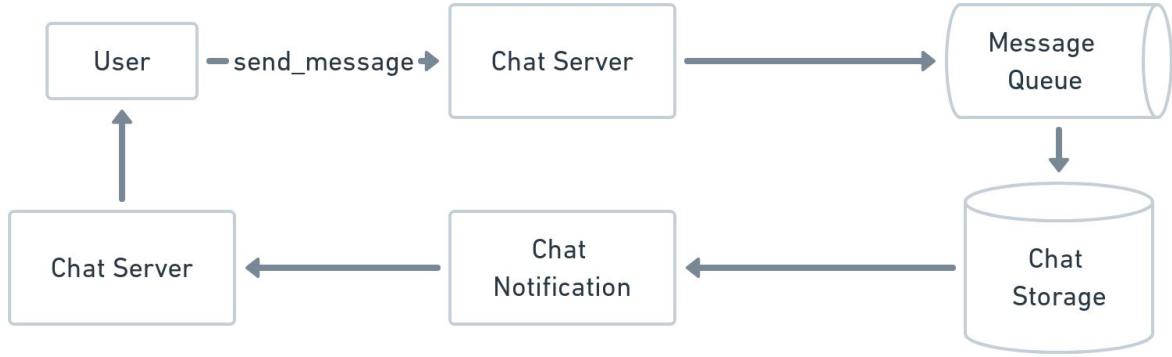
I would like to focus a little bit on the architecture since the end-to-end delivery is very time-sensitive.

Option 1: Simultaneous Broadcast and Persist to Disk



The message gets fanned out to both the chat storage and the chat notification when removed from the queue. The latency will be lower here since we don't need to wait for the message to go to disk first. However, when the storage can not persist, the system will notify a message that never made it to the database, leading to inconsistency.

Option 2: Write to Disk First, Then Broadcast



In this architecture, a message is written to disk first before notifying the user. The advantage is the notified message is guaranteed to be durably stored. If the message fails to persist to disk, the system will not broadcast the message to recipients. However, the end-to-end latency will suffer because it adds in the disk latency.

Conclusion

Given we need a very low latency application. I would still prefer option 1 and monitor for chat storage failure rate, then retry the failures and hope they go through. If failures become a problem, we should investigate to reduce it. For now, we'll assume a rare error event leading to an inconsistent notified message is acceptable.

Read Performance Consideration

Candidate:

Let's focus on the performance of the read query.

Option 1: Full Table Scan

Let's assume this is the schema:

Channel ID	Timestamp	Message	Author
------------	-----------	---------	--------

To satisfy `read_messages`, I need to read the latest messages for a channel ID. I will need to filter for all the messages in the `channel_id`, sort it by timestamp, and return the paginated list of messages. The advantage is we don't need to worry about indexing on write. However, the read query will be poor without the index.

Option 2: Compound Index

We can index the table by channel ID and timestamp. It will look like this:

Channel Id	Timestamp
1	10:00:00.000 AM
1	10:02:00.000 AM
1	10:04:00.000 AM
2	10:00:00.000 AM
2	10:00:00.001 AM
2	10:01:00.000 AM

For each channel ID, the timestamps are sorted already, so retrieving the latest N messages will be efficient. The downside to this approach is the write will be slower with an index.

Option 3: Read Cache

Another way to increase the read performance is the read cache . Given the read latency is 200-300 ms p95 and the read to write ratio is low, I'm not sure if it's worth dealing with the cache complexity. For every message write, if we do a write-around with read-through cache, the writes will frequently invalidate the cache. If we write to the cache first with a write-back cache, we risk losing the messages before persisting to the disk.

Reminder

Many candidates like to introduce cache as a default solution, but as you can see, there's quite a bit to think about and the conclusion can still be no cache.

Conclusion

I would choose option 2 since the read is much more performant. The write is slightly slower, but we're already notifying the users before disk, so the

disk latency isn't an issue.

Real-Time Protocol

Interviewer:

You mentioned WebSocket, are there other alternatives?

Candidate:

Sure, there are a couple of options.

Option 1: Short Polling

In short polling, the client will periodically call the backend for the latest messages. The advantage is simplicity since a simple HTTP would work at the cost of many unnecessary calls if there's no update.

Option 2: Long Polling

In long polling, the client will call the server and keep the connection open until a new message is available. If there's no response after some time in long polling, it will time out and require a client retry. The server also has to deal with the complexity to keep the connection open.

Option 3: WebSocket

The advantage of a WebSocket is to keep the connection on both sides. The advantage is if a new message comes, the message is forwarded to the client right away. The downside is the complexity of maintaining and building the WebSocket infrastructure.

Conclusion

WebSocket is gaining a lot of popularity these days, and given the inefficiency of short and long polling, we will just go with WebSocket.

Interviewer:

Sounds good. That sounds pretty straightforward. Can you go over how you would scale up your database?

Database Sharding Strategies

Candidate:

Sure, I'm going to assume by scale up, you mean whether the database can handle the write load. Should I do some back-of-the-envelope math to see how many machines are needed?

Interviewer:

No, let's just assume we need multiple shards. What is your sharding strategy?

Candidate:

Ok, let me think of a couple of options:

Option 1: Shard by Channel ID

In this solution, we will shard by channel ID using consistent hashing. The advantage here is to retrieve a list of messages for a channel ID. Sharding by `channel_id` will be very efficient since they're all in the same shard. The disadvantages are hotspots. If there is a lot of activity for a channel ID, that shard will be hot. Also, popular channels will lead to significantly more messages too.

Option 2: Shard by Timestamp Bucket

Sharding by timestamp bucket means having buckets of time ranges. There will be a bucket for the current time where all the channels write to.



The advantage of doing so is that you can fetch all the latest messages for all the channels in the same shard. Sharding by timestamp will lead to a hotspot since all writes for all channels are written to the same shard.

Option 3: Shard by Channel ID, Timestamp Bucket

Instead, we can shard by channel ID and timestamp bucket. The advantage of doing so is the ability to fetch the latest data for a channel effectively. Instead of having a shard that handles all messages for all channels, the scheme will distribute channels across shards. Another advantage is you're able to fetch a channel's recent message efficiently. The disadvantage is there's still a potential hotspot if a channel has unusually high activity. Also, the system will create new time shards consistently, so if a channel is inactive, you may need to hit multiple shards before hitting the shard with the latest messages.

Shard 1

Channel Id	Timestamp
1	10:00:00.000 AM
1	10:02:00.000 AM
1	10:04:00.000 AM
2	10:00:00.000 AM

Shard 2

Channel Id	Timestamp
3	10:00:00.000 AM
3	10:02:00.000 AM
4	10:04:00.000 AM
4	10:00:00.000 AM

Shard 3

Channel Id	Timestamp
1	9:00:00.000 AM
1	9:02:00.000 AM
1	9:04:00.000 AM
2	9:00:00.000 AM

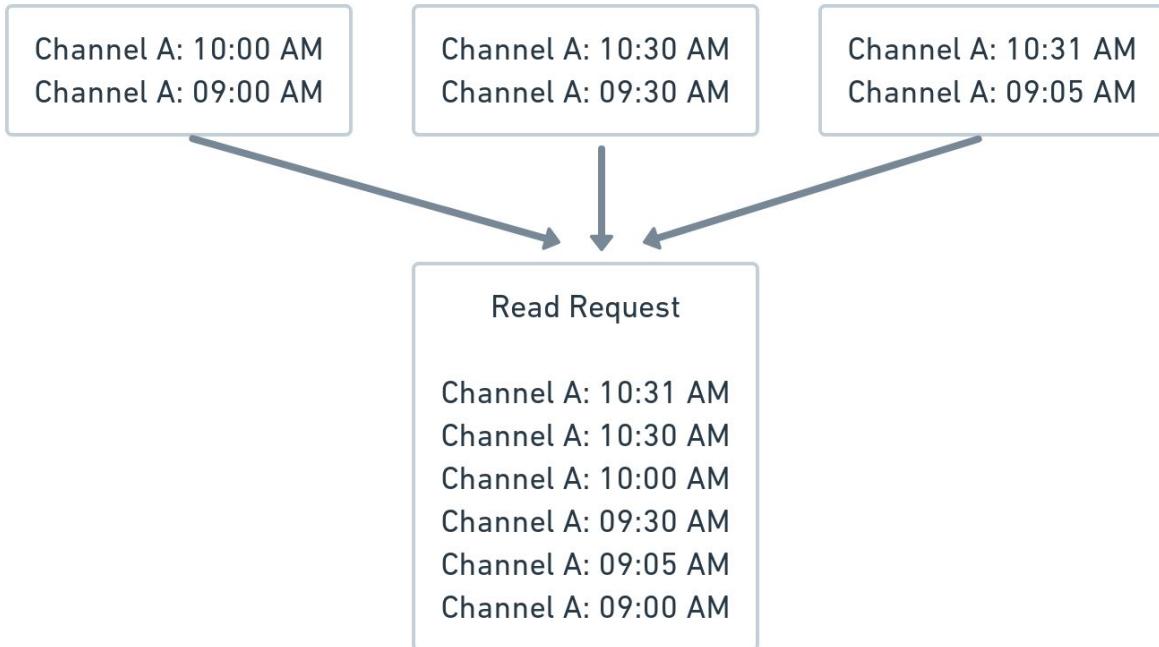
Shard 4

Channel Id	Timestamp
3	9:00:00.000 AM
3	9:02:00.000 AM
4	9:04:00.000 AM
4	9:00:00.000 AM

Using the example above, you would fetch for `channel_id` 1's recent messages in shard 1 and `channel_id` 4's in shard 2.

Option 4: Shard by Channel ID and Random

In this design, I will predefine a list of shards for a channel ID. Upon receiving a write request, I will randomly pick a server to write to. On read, I will have to scatter-gather from all the shards for that channel ID.



The advantage of doing so is there won't be a hot shard due to the latest timestamp. However, the disadvantage is that the read will be less efficient since we need to scatter-gather. If we add more nodes, the scatter will become even bigger.

Conclusion

Options 1 and 2 are probably not sharded enough and will cause storage and hotspot issues down the line. Option 3 is perhaps my choice, and I just need to be aware of a super popular channel and potentially handle it differently. Option 4 is an option if we absolutely can't handle the write throughput. In practice, super high QPS is unlikely. Imagine people are writing at 10,000 QPS. Who is going to be able to read any messages?

Interviewer:

We are out of time. Thanks for your time!

What's Next

We would love to hear your feedback! If there are any errors, missing fundamentals, incorrect assumptions, or if you have any suggestions, please let us know at software.interview.fundamentals@gmail.com.

Please take a moment to review the book on Amazon, as the review will help us improve in future editions and reach a wider audience.

Appendix

Appendix 1: Number Conversions

Raw Number	Scientific Notation	Query Units	Memory Units
1,000	1×10^3	Thousand	KB
1,000,000	1×10^6	Million	MB
1,000,000,000	1×10^9	Billion	GB
1,000,000,000,000	1×10^{12}	Trillion	TB
1,000,000,000,000,000	1×10^{15}	Quadrillion	PB

Make sure you use mnemonic techniques to map between scientific notation, query units, and memory units quickly and accurately. Here's ours for storage:

3 Kudos
6 Monkeys
9 Grapes
12 Tents
15 Pizzas

Scientific Notation to Query Units and Memory Units

For example, you should know 6 maps to million and MB and 12 maps to trillion and TB. If you're given 13, round down to the nearest multiple of 3, which is 12, and recognize that it is 10 times a trillion or TB. Here are some examples:

6×10^7 Queries → 60 Million Queries

3×10^{12} Queries → 3 Trillion Queries

6×10^7 Bytes → 60 MB

$$3 \times 10^{12} \text{ Bytes} \rightarrow 3 \text{ TB}$$

Query Units and Memory Unit to Scientific Notation

Given query units, you should quickly convert them to scientific notation. If you're dealing with a million, you need to think of the number 10^6 . Shift the power of 10 by however many 0s the base number has. For example, 80 billion gives you 10^9 , and since 80 has a multiple of 10 (8×10), multiply 10^9 by 10 and give you 10^{10} , which results in 8×10^{10} .

$$4 \text{ Million Queries} \rightarrow 4 \times 10^6 \text{ Queries}$$

$$700 \text{ Trillion Queries} \rightarrow 7 \times 10^{14} \text{ Queries}$$

$$4 \text{ MB} \rightarrow 4 \times 10^6 \text{ Bytes}$$

$$700 \text{ TB} \rightarrow 7 \times 10^{14} \text{ Bytes}$$

Appendix 2: Capacity Numbers

The purpose of the single machine capacity numbers is to provide a *rough* guideline and intuition to calculate the capacity. This guideline helps prevent you from coming up with an off-the-chart number. Your job is simply to calculate the capacity and provide some intuition on how you came up with the number.

In practice, people may experience different results due to several factors. Feel free to use *your* number if you feel more comfortable. Just make sure you explain the reasoning.

Type	Low	Medium	High
Database Read QPS	100	200	500
SSD Read QPS	1,000	3,000	10,000
Memory Read QPS	10,000	30,000	100,000
Bandwidth	10 Gbps	15 Gbps	25 Gbps
Database	512 GB	8 TB	64 TB
Memory	64 GB	256 GB	1 TB
Connections	50,000	200,000	1,000,000

The QPS number assumes it is single-threaded, and feel free to shift the QPS number based on the following factors (not an exhaustive list):

- Size of payload into or out of the memory or database.
- The amount of data in the datastore.
- The complexity of the query.
- Database selections (LSM vs B-Tree).

Warning

Bandwidth is usually measured in *bits* per second not *bytes* per second.

Since there are 8 bits in a byte, you can approximate it by dividing by 10 instead of 8 to derive bytes from bits.

Appendix 3: Math Number Assumptions

These numbers are intended to be used for rough approximation for back-of-the-envelope math. In practice, the numbers can vary due to several factors that aren't important in an interview context.

Datastore Latency

Storage	Latency
Fetching 1 MB from disk	Disk seek: 2 ms Read 1 MB sequentially on disk: 1 ms Total: 3 ms
Fetching 1 MB from SSD	SSD random read: 15 μ s Reading 1 MB sequentially from SSD: 200 μ s Total: 0.2 ms
Fetching 1 MB from memory	Memory reference: 100 ns Reading 1 MB sequentially from memory: 10 μ s Total: 0.01 ms

Note: The latency to fetch for 1 MB of data requires the fixed cost of seeking where the data is and sequentially reading the data.

Network Latency

Distance	Latency
California to the Netherlands	150 ms
California to Virginia	60 ms
Inter Datacenter	0.5 ms

Images

Quality	Size	Examples
Low	10 KB	Thumbnail, small website images
Medium	100 KB	Website photos
High	2 MB	Phone camera photos
Ultra High	20 MB	RAW photographer image

Video

Quality	1-Minute Storage	Examples
Low	30 MB	480p video
Medium	90 MB	1080p video
High	500 MB	4k video

Audio

Quality	1-Minute Storage	Examples
Low	700 KB	Low quality Mp3
High	2.5 MB	High quality Mp3

Storage Data Type

Type	Memory
Boolean	1 Bit
Char	1 Byte
Int	4 Bytes
Timestamp	4 Bytes
BigInt	8 Bytes

Appendix 4: Product Math Assumptions

It's useful to have some basic intuition on what reasonable numbers are. Most of the time, you should just ask the interviewer about the numbers. If they ask you to make your assumption, it would be slightly alarming if you give an off-the-chart number. For example, if you say there are 1 trillion DAU on Facebook or 100 DAU on Uber.

2021 Numbers	
World population	8 billion
US population	300 million
Facebook MAU	3 billion
Instagram MAU	1 billion
Uber MAU	100 million
Google searches per day	5.6 billion

MAU = Monthly active users

Appendix 5: Advanced Concepts

Do note that these are advanced concepts that are usually not required in an interview, but it's still useful to know how things work in real life in case they come up. Don't overly stress over these concepts. Optimize for the fundamentals first. If you have more time, you can dig deeper into these advanced concepts. Just know when and why you would use the concepts.

HyperLogLog

Problem It Is Trying to Solve

Maintaining a unique count while keeping memory low.

What Is It?

A probabilistic data structure to approximate the number of unique IDs. The intuition is if you see a high number, you're more likely to have seen more numbers, hence a higher count. For example, given a 3-bit number:

- 50% of the numbers' bits start with 1 (001, 101, 011, 111).
- 25% of the numbers' bits start with 10 (010, 110).
- 12.5% of the numbers' bits start with 100 (100).

If you see a number's starting bit of 100, that means you've probably seen eight numbers.

Applicable Design Considerations

- Count the number of unique visitors to a site, post, etc.

Count-Min Sketch

Problem It Is Trying to Solve

When you want to maintain a key to count tables, and it doesn't fit into memory. This data structure allows you to fit into memory by sacrificing accuracy for memory.

Applicable Design Considerations

- Calculate top K in near real-time.

Collaborative Filtering

Problem It Is Trying to Solve

Determine what a user might also like. A popular recommendation algorithm looks at similarities between users to make “users like you also bought” type of recommendations.

Applicable Design Considerations

- Design Netflix recommendation.
- Design Amazon “customers also bought” recommendation.

Operational Transform

Problem It Is Trying to Solve

Automatically resolve document collaboration conflicts.

Applicable Design Considerations

- Design Google Docs.

Z-Score

Problem It Is Trying to Solve

Define what is trending by measuring how many standard deviations away from the mean topics are. Z-Score is used to determine trending topics where an outlier from its norm defines trending. For example, if a video is viewed on average 5 times a day with a standard deviation of 1, it's most likely trending if viewed 100 times in the last hour.

Applicable Design Considerations

- Design a trending topic.

EdgeRank

Problem It Is Trying to Solve

Rank which feed will drive more engagement

Facebook's feed ranking algorithm functions by looking at user affinity, content weight, and time-based decay. The intuition is if you're close to a user, you're more likely to want to see that user's post. Content weight means how much activity is on the feed. More activity on the feed implies

it's more interesting. Time decay means the older the post, the less interesting.

Applicable Design Considerations

- Design Instagram.
- Design Twitter.
- Design News Feed.

Bloom Filter

Problem It Is Trying to Solve

Provides a fast and space-efficient way to determine if an element is part of a set of elements.

Bloom filter is a probabilistic data structure to return if an element exists in a set of elements. Bloom filters can return false positive results. For example, you want to know if an ID exists in the database. Since each disk-seek is expensive, you can ask the bloom filter instead. If the bloom filter says it doesn't exist, you can use the ID. If the bloom filter says it exists, you can check the database for the real answer because it can return false positives. The saving comes from the fast look up when it returns "ID doesn't exist."

Applicable Design Considerations

- Check if an ID exists in the database.
- Graph database where you might use a bloom filter to check if someone is a second-degree connection.
- For web crawlers, you can ask the bloom filter if the crawler crawled the website already.

TF / IDF

Problem It Is Trying to Solve

A full-text search provides relevance ranking for document results based on the rarity of terms in the set of documents.

TF stands for term frequency which calculates how frequently a term exists in a document. IDF stands for inverse document frequency, which

calculates how common the term is in the set of documents. The intuition is the more frequent the word exists and the rarer it exists across the documents, the higher the score is. Let's look at an example:

Document A: "please study system design"

Document B: "please study system design interview"

	A	B
please	0	0
study	0	0
system	0	0
design	0	0
interview	0	1

A TF/IDF score is given to a term document combination. Each cell is the corresponding TF/IDF score. When someone searches for "interview," document B has a higher number for "interview," hence should score higher.

Applicable Design Considerations

- Any questions related to text document search would benefit from TF/IDF discussion.

Page Rank

Problem It Is Trying to Solve

Search engines provide relevance ranking for webpage results based on how much other websites reference the website.

Websites usually have links to other websites. If many other websites reference a website, the intuition is that the website should be a popular one. The quality of the referencing website matters. The higher weight the referencing website has, the more weight the referenced website receives.

Applicable Design Considerations

- If you're designing a website search engine, you may consider page rank as a possible algorithm.

Appendix 6: Security Considerations

Here is a list of security topics that are useful to consider in a web application. They are useful to know but unlikely to show up in an interview. Come back and ramp them up if you have time.

1. JavaScript Injection
2. Cross site scripting
3. Firewall
4. PII and password storage
5. Phishing attacks
6. TLS
7. SQL injection
8. DDoS
9. CAPTCHA
10. OAuth
11. Access control

Appendix 7: System Design Framework Cheat Sheet

System Design Framework	
Step 1: Gather Requirements	Functional: 2-3 minutes Non-Functional: 3-5 minutes
Step 2: Define API	3-5 minutes
Step 3: Define High-Level Diagram	5-7 minutes
Step 4: Define Schema and Data Structures	5-7 minutes
Step 5: Summarize End to End Flow	2-3 minutes
Step 6: Deep Dives	15-20 minutes

* This time suggestion is based on a 40 minute interview.

Non-Functional Cheat Sheet	
Data Distribution Scale numbers Distribution of storage and query Geo distribution of users	Performance Constraint Consistency and availability Response time Latency Accuracy Freshness Durability

Deep Dive Cycle
Step 1: Identify the most important discussion point Step 2: Come up with options Step 3: Discuss the trade offs

Step 4: Conclude with one solution

Step 5: Active discussion with interviewer

Step 6: Go to Step 1