The complete guide to the System Design Interview in 2022

At the beginning of the year, we published a guide to System Design in 2022 to help you navigate the world of System Design. It details the fundamental concepts of System Design and provides links to relevant resources to help you gain a deeper understanding. As a developer, you'll be increasingly expected to understand *and* apply System Design concepts to your job. As you gain more experience and move up levels, System Design will become a much larger part of your interview processes. So, it's important to learn the fundamentals of System Design to set yourself up for success in your career and in your interviews.

With this guide, we want to help you prepare for your System Design Interview. We hope to help you understand the following:

- What is the System Design Interview?
- How to prepare for the System Design Interview
- Fundamental concepts and patterns in the System Design Interview

    - PACELC theorem
    - Heartbeat
    - AJAX polling
    - HTTP long-polling
    - WebSockets
    - Server-sent events (SSEs)
- Walkthrough of common System Design Interview questions
    - Design Uber
    - Design TinyURL
    - Design Instagram
- List of common System Design Interview questions
- How to approach *any* System Design Interview question
- System Design Interview cheat sheet
- System Design Interview resources

Let's get started!

What is the System Design Interview?

On his blog, Educative's co-founder and CEO, Fahim ul Haq, recounts how the coding interview has evolved in the last 25 years and shares what we can expect going forward. Some of the key points include:

- In the early 2000s, companies would focus on coding portions of the interview and also brain teasers. The underlying belief here was that creativity and problem-solving skills tied into successful coding. The problem with these brain teasers was that many of the questions required knowledge of real-world scenarios that didn't really reflect actual programming problems.
- Then there was what's known as the "Web 2.0" and the rise of social media. Internet use became mainstream and the need for highly efficient

and scalable large-scale systems arose. The development of consumer-facing apps was moving toward the web and cross-platform operations.
    - At this time, interviewing began to change. Google stopped using brain teasers and had candidates solve actual problems with code. Distributed systems existed, but there weren't established best practices yet. Scaling issues were handled organically rather than algorithmically, and "System Design" had *just* started entering software circles.
- In the mid-2000s, things began to change. MapReduce was introduced, along with BigTable and scalable data management, and practices for large systems began to be established. Then came the rise of the cloud.
    - At this time, interviewing began to change even more. Facebook added "System Design" to their interview cycles in 2012. "Toy" problems became standardized. This all led to the present age of the coding interview.
- Now, we find that the coding interview is a living thing – **constantly changing in response to product and market needs and architecture changes**. When interviewing today, it's critical to go back to the basics of software development, System Design, data structures and algorithms, and to understand how components relate to each other in object-oriented programming and multi-entity systems.
    - We also know that interviews vary a bit from company to company. For example, the Amazon interview incorporates its leadership principles into its questions. It's important to **study the company you're interviewing with** to prepare for the various nuances that exist at different companies.

In The complete guide to System Design in 2022, we outline *why* it's so important to learn System Design as a developer. In short, it's important because, throughout your career, you'll be increasingly expected to understand System Design concepts and how to apply them. In the early stages of your career, System Design concepts will help you tackle software design challenges with confidence. In the later stages of your career, System Design will become a

much larger part of your interview process. **Companies want to see that you can work with scalable, distributed large-scale systems**.

Now that we know a little bit more about the System Design Interview, let's discuss how to prepare.

How to prepare for the System Design Interview

In How to prepare for the System Design Interview in 2022, Educative's CEO and cofounder, Fahim ul Haq, shares what he learned through **leading hundreds of System Design Interviews at Microsoft, Facebook, and now Educative**. A few key takeaways include:

- In System Design Interviews, companies aren't trying to *test* your experience with System Design. Successful candidates rarely have ample experience working on large-scale systems. Interviewers know this. The key to success is to prepare for your interview with the *intent to apply that knowledge*.
- System Design Interviews are different than standard coding interviews in that they're conducted as **free-form discussions** with no right or wrong answers. The interviewer is trying to evaluate your ability to lead a conversation about the different components of the system and assess the solution based on the given requirements.
- Three major focus areas in your prep plan should include the **fundamentals of distributed systems, large-scale web application architecture, and how to design distributed systems**.

Why it's important to prepare strategically

As mentioned earlier, the way you prepare for an interview at Amazon will probably differ from the way you'd prepare for one at Slack, for example. While the overall interview process shares similarities across various

companies, there are also distinct differences that you must prepare for. This is one of the reasons why preparing strategically is so important. If you're intentional and thorough when creating an interview prep plan, you'll feel more confident in the long run.

> CodingInterview.com is a great *free* resource with over 20 company-specific interview guides to help you maximize your chances of success.

Fundamental concepts in the System Design Interview

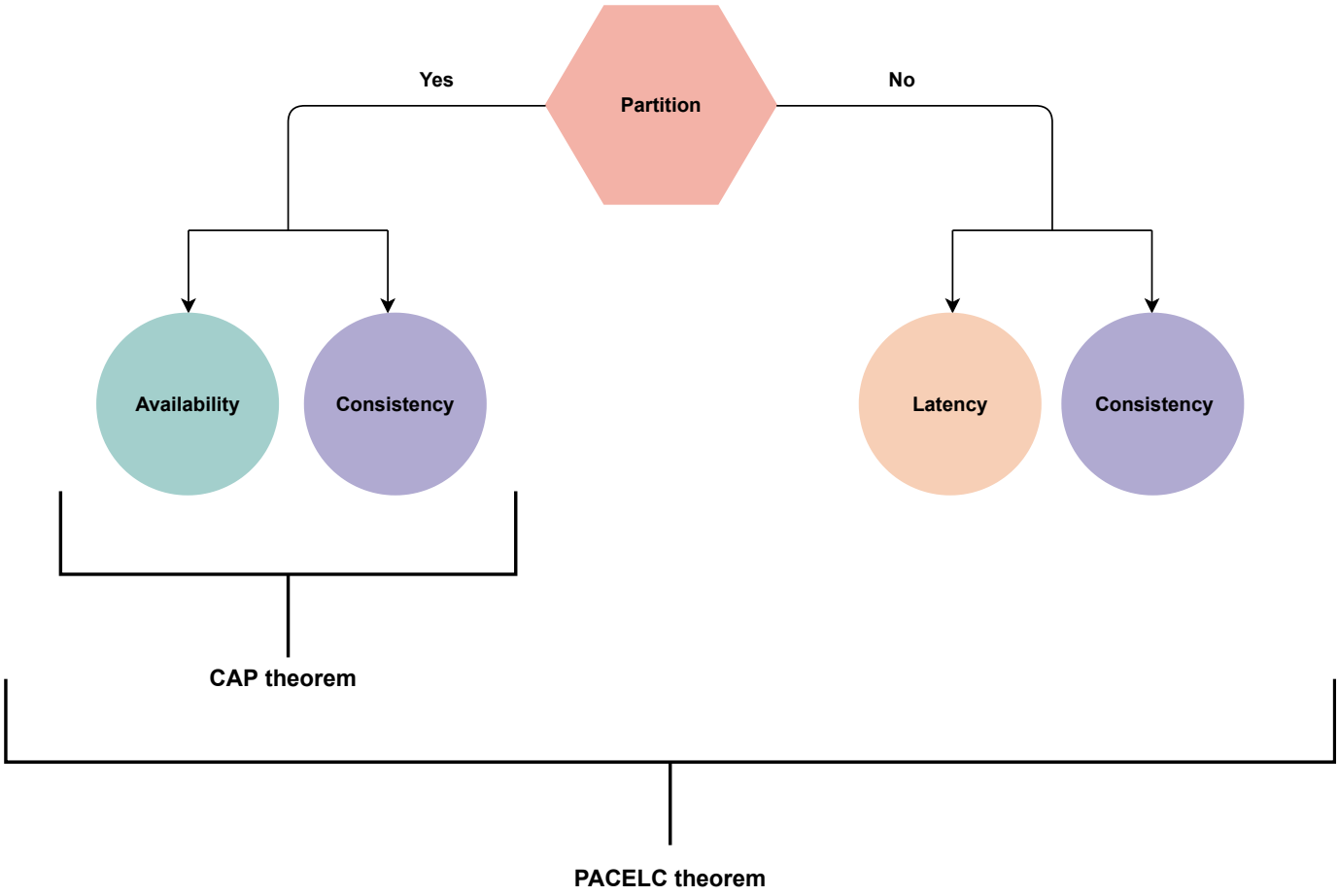
The complete guide to System Design in 2022 covers most of the fundamentals of distributed systems and system design. I recommend starting your journey there. In *this* guide, we'll explore some concepts that aren't covered in the other guide.

PACELC theorem

A question that the CAP theorem doesn't answer is ***"what choices does a distributed system have when there are no network partitions?"***. The PACELC theorem answers this question.

The PACELC theorem states the following about a system that replicates data:

- `if statement` : If there's a partition, a distributed system can trade off between availability and consistency.
- `else statement` : When the system is running normally in the absence of partitions, the system can trade off between latency and consistency.



The first three letter of the theorem, PAC, are the same as the CAP theorem. The ELC is the extension here. The theorem assumes we maintain high availability by replication. When there's a failure, the CAP theorem prevails. If there isn't a failure, we still have to consider the tradeoff between consistency and latency of a replicated system.

Examples of a PA/EC system include BigTable and HBase. They'll always choose consistency, giving up availability and lower latency. Examples of a PA/EL system include Dynamo and Cassandra. They choose availability over consistency when a partition occurs. Otherwise, they choose lower latency. An example of a PA/EC system include MongoDB. In the case of a partition, it chooses availability, but otherwise guarantees consistency.

To learn more about these systems, check out Distributed Systems for Practitioners.

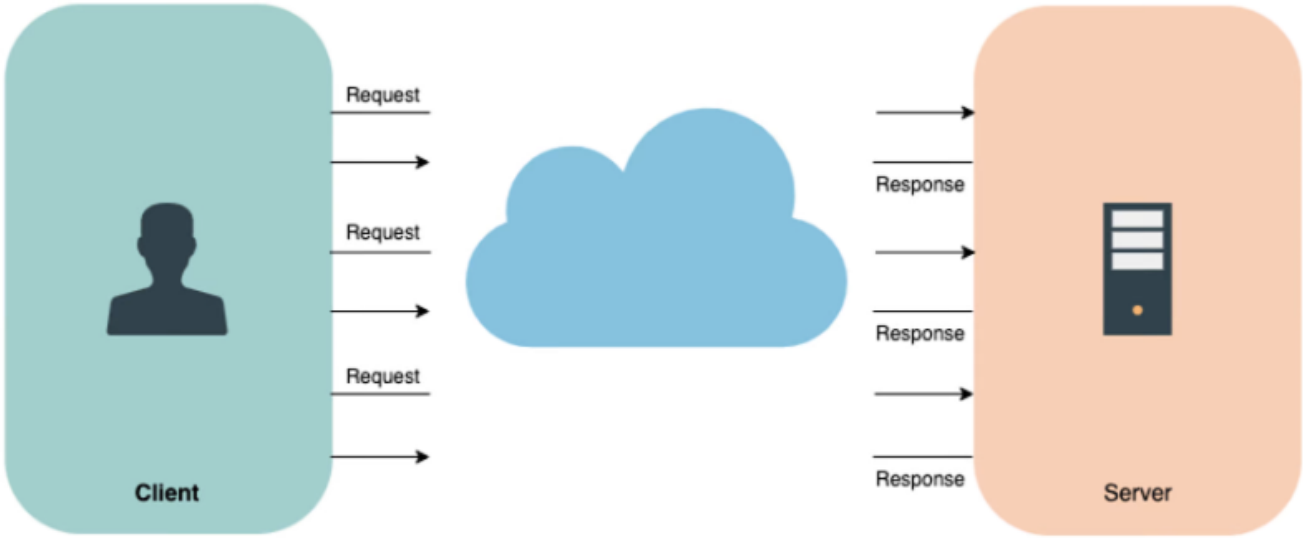
Heartbeat

A heartbeat message is a mechanism that helps us **detect failures in a distributed system**. If there's a central server, all servers periodically send a heartbeat message to it to show that it's still alive and functioning. If there's no central server, all servers randomly select a set of servers and send that set a heartbeat message every few seconds. This way, if there are no heartbeat messages received for awhile, the system can suspect there might be a failure or a crash.

To learn more about heartbeat, check out Grokking the System Design Interview.

AJAX polling

Polling is a standard technique used by most AJAX apps. The idea is that **the client repeatedly polls a server for data**. The client makes a request and waits for the server to respond with data. If no data is available, the server returns an empty response.

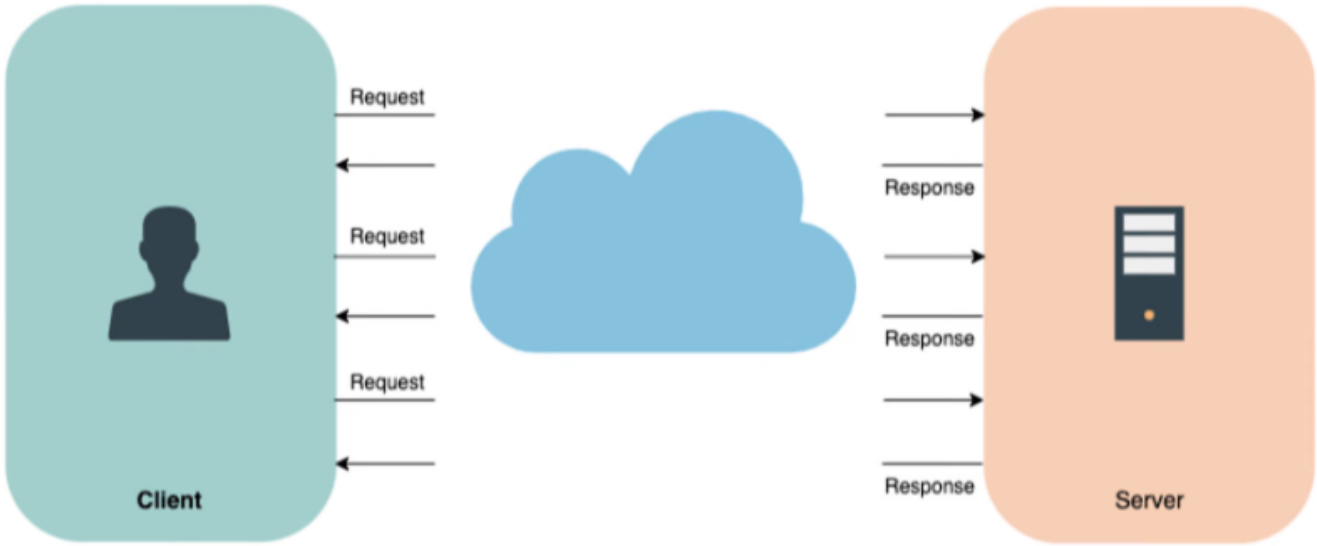


AJAX polling

To learn more about AJAX polling, check out Scalability & System Design for Developers.

HTTP long-polling

With long-polling, the client requests information from the server, but the server may not respond immediately. This technique is sometimes called **“hanging GET”**. If the server doesn’t have any available data for the client, it’ll hold the request and wait until there is data available instead of sending an empty response. Once the data becomes available, a full response is sent to the client. The client immediately re-requests information from the server so that the server will almost always have an available waiting request that it can use to deliver data in response to an event.



HTTP long polling

To learn more about HTTP long-polling, check out Scalability & System Design for Developers.

WebSockets

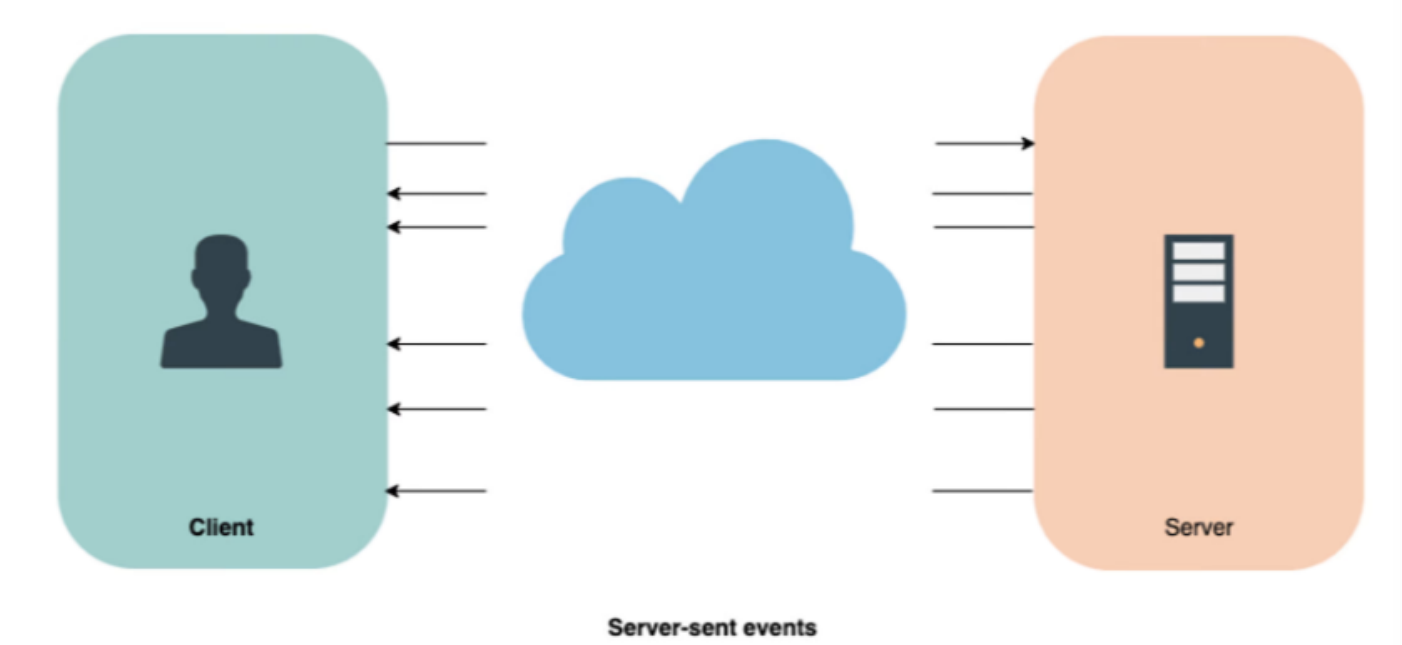
WebSocket provides **full duplex communication channels over a single TCP connection**. It provides a persistent connection between a client and server. Both parties can use this connection to start sending data at any time. The client establishes a connection through a WebSocket handshake. If the process succeeds, the server and client can begin exchanging data in both directions at any time.

WebSockets

To learn more about WebSockets, check out Scalability & System Design for Developers.

Server-sent events (SSEs)

A client can **establish a long-term connection with a server** using SSEs. The server uses this connection to send data to a client. If the client wants to send data to the server, it would require the use of another technology or protocol.



To learn more about server-sent events, check out Scalability & System Design for Developers.

> As mentioned above, for a more comprehensive overview of fundamentals, check out The complete guide to System Design in 2022. To get hands-on with these fundamentals, check out Scalability & System Design for Developers.

Walkthrough of common System Design Interview questions

There are numerous different questions you could encounter during your System Design Interview. Today, we'll walk through three common questions and their solutions. At the end of this section, we'll provide you with a resource to learn about more common SDI questions.

Design Uber

This problem walkthrough is adapted from another article. For a more detailed tutorial, I recommend checking out Design Uber. There, you'll dive deeper into use cases and advanced issues and considerations.

Problem overview

When designing Uber, there are two kinds of users our system should account for: drivers and users. The system requirements are as follows:

- Drivers can frequently notify the service regarding location and availability
- Users can see all nearby drivers in real-time
- Users can request a ride using a destination and pick-up time
- Nearby drivers are notified when a user needs a ride
- Once a ride is accepted, both the driver and user can see the other's location for the duration of the trip
- Once the drive is complete, the driver completes the ride and is available for other users

Constraints

Constraints are very important to consider before designing Uber. These constraints typically differ depending on the time of the day and the location. We'll design Uber with the following constraints and estimations:

- 300 million users and 1 million drivers in the system
- 1 million active customers and 500 thousand active drivers per day
- 1 million rides per day
- All active drivers notify their current location every three seconds
- System contacts drivers in real-time when a user requests a ride

Design considerations

- We need to update data structures to reflect drivers' locations every three seconds. To update a driver to a new location, we need to find the right

grid based on their previous location.

- If the new position doesn't belong to the current grid, we need to remove the driver from the grid and reinsert them into the appropriate grid. If the new grid reaches a maximum limit, we have to repartition it.
- We need a quick mechanism to propagate the current location of nearby drivers to users in the area. Our system needs to notify the driver and the user of the car's location for the duration of the ride.

Keeping those considerations in mind, we can determine that a QuadTree isn't ideal because we can't guarantee that the tree will update as quickly as our system requires. We can keep the most recent driver position in a hash table and update our QuadTree less frequently. We want to guarantee that a driver's current location is reflected in the QuadTree within 15 seconds. We'll call our hash table `DriverLocationHT`.

Solving the problem

`DriverLocationHT`

We need to store `DriverID` in the hash table, which reflects a driver's current and previous locations. This means that we'll need 35 bytes to store each record. Let's break this down into separate parts:

1. DriverID (3 bytes)
2. Old latitude (8 bytes)
3. Old longitude (8 bytes)
4. New latitude (8 bytes)
5. New longitude (8 bytes)

Since we assume one million, we'll require the following memory:

*1 million * 35 bytes => 35 MB*

Now we'll discuss bandwidth. If we get the DriverID and location, it will require (3 + 16 => 19 bytes). This information is gathered every three seconds from 500

thousand daily active drivers, so we will receive 9.5 MB every three seconds.

To randomize distribution, we could distribute `DriverLocationHT` on multiple servers based on the DriverID. This will help with scalability, performance, and fault tolerance. We'll refer to the machines holding this information as "driver location servers". These servers will do two more things. Once the server receives an update on a driver's location, it will broadcast it to relevant users. The server will also notify the respective QuadTree server to refresh the driver's location.

Broadcasting driver locations

We need to broadcast driver locations to users. We can use a Push Model so the server pushes positions to relevant users. We can use a notification service and build it on the publisher/subscriber model.

When a user opens the app, they'll query the server to find nearby drivers. On the server-side, we'll subscribe the user to all updates from nearby drivers. Each update in a driver's location in `DriverLocationHT` will be broadcasted to all subscribed users. This will ensure that each driver's current location is displayed.

We assumed one million active users and 500 thousand active drivers per day. Let's assume that five customers subscribe to one driver. We can store that information in a hash table for quick updates.

We need to store both driver and user IDs. We need 3 bytes for DriverID and 8 bytes for UserID, so we'll need 21 MB of memory:

$$(500{,}000 * 3) + (500{,}000 * 5 * 8) = 21\ MB$$

Now for bandwidth. For every active driver, we have 5 subscribers. In total, this reaches:

$$5 * 500{,}00 \Rightarrow 2.5\ million$$

We need to send DriverID (3 bytes) and their location (16 bytes) every second, which requires:

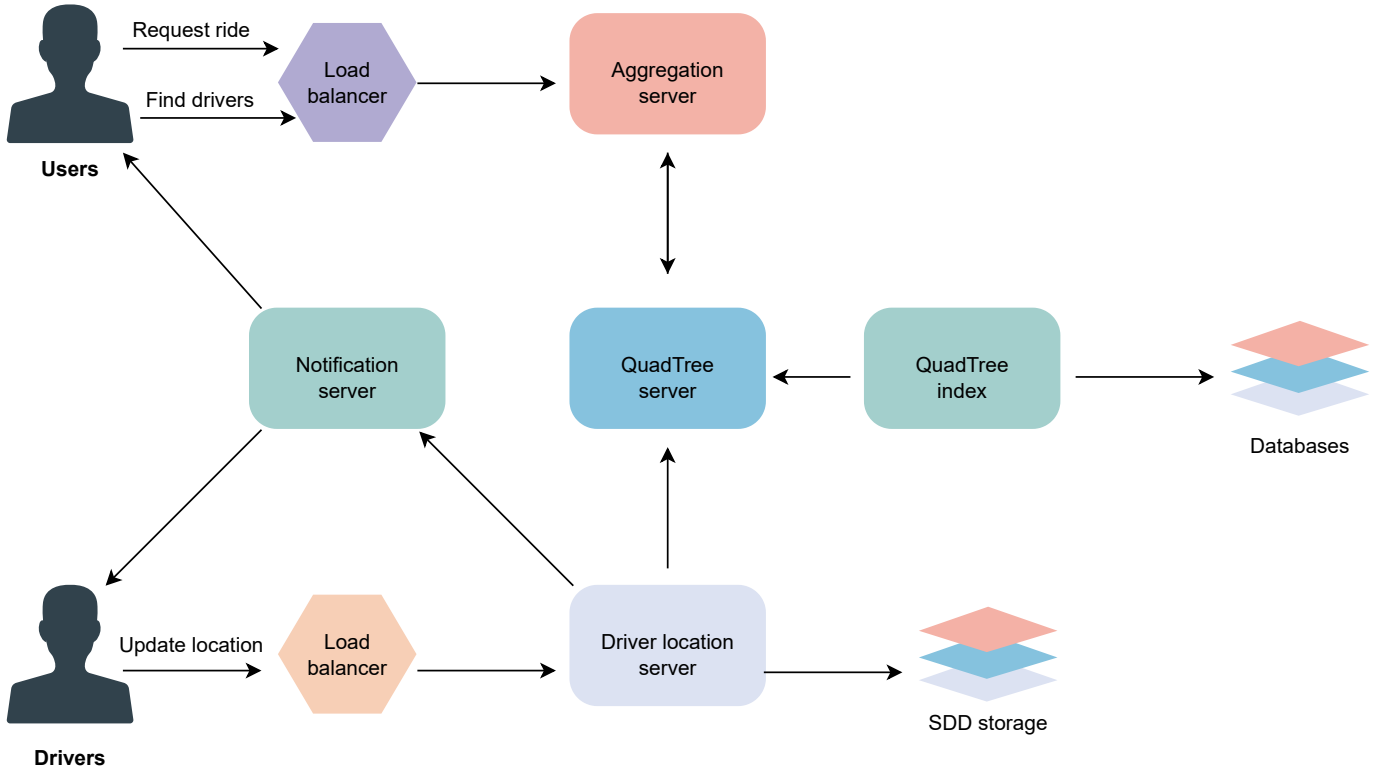
*2.5 million * 19 bytes => 47.5 MB/s*

Notification service

To efficiently implement our notification service, we can either use HTTP long polling or push notifications. Users are subscribed to nearby drivers when they open the app for the first time. When new drivers enter their areas, we need to add a new user/driver subscription dynamically. To achieve this, we track the area that a user is watching. However, this can get pretty complicated.

Instead of pushing this information, we can design the system so users pull the information from the server. Users will send their current location so that the server can find nearby drivers from our QuadTree. The customer can then update their screen to reflect drivers' current positions.

To repartition, we can create a cushion so that each grid grows beyond the limit before we decide to partition it. Assume our grids can grow or shrink by an extra 10% before we partition them. This will decrease the load for a grid partition.



Design TinyURL

This problem walkthrough is adapted from another article. For a more detailed tutorial, I recommend checking out Design TinyURL.

Problem overview

TinyURL is a URL-shortening service that creates shorter aliases for long URLs. Users select these shortened URLs and are redirected to the original URL. This service is useful because shorter links save space and are easier to type. Let's consider some functional and non-functional requirements for designing TinyURL:

Functional requirements:

- Our service will generate a shorter alias of the original URL that can be easily copied and pasted
- The short link should redirect users to the original link
- Users should have the option to pick a custom short link for their URL
- Short links will expire after a default timespan, which users can specify

Non-functional requirements:

- The system must be highly available
- URL redirection should happen in real-time with minimal latency
- Shortened links should not be predictable

Capacity estimations and constraints

Our system will be read-heavy. There will be a large number of redirection requests compared to new URL shortenings. Let's assume a 100:1 read/write ratio. Our reads are redirection requests, and our writes are new URL shortenings.

Traffic estimates

Let's assume we have 500 million new URL shortenings per month. Based on our 100:1 read/write ratio, we can expect 50 billion redirections during the same period:

*100 * 500 million = 50 billion*

We'll now determine our system's queries per second (QPS). We'll take the monthly amount of 50 billion to calculate the new URL shortenings per second:

*500 million / (30 days * 24 hours * 3,600 seconds) = 200 URLs/second*

Applying the 100:1 read/write ratio again, URL redirections per second will be:

*100 * 200 URLs/second = 20,000/second*

Storage estimates

Let's assume we store every URL shortening request and its associated link for five years. Since we expect to have 500 million new URLs every month, the total number of objects we expect to store over five years will be 30 billion:

*500 million * 12 months * 5 years = 30 billion*

For now, let's assume that each stored object will be approximately 500 bytes. This means that we'll need 15TB of storage for the five year period:

*30 billion * 500 bytes = 15 TB*

Bandwidth estimates

We expect 200 new URLs per second for write requests. This makes our service's total incoming data 100KB per second:

*200 * 500 bytes = 100 KB/s*

For read requests, we expect approximately 20,000 URL redirections every second. Then, total outgoing data for our service would be 10MB per second:

*20,000 * 500 bytes = 10 MB/s*

Memory estimates

We'll need to determine how much memory we'll need to store the frequently accessed hot URLs' cache. If we follow the 80/20 rule, then 20% of URLs will generate 80% of traffic. We would like to cache this 20% of hot URLs. Since we have 20,000 requests per second, we'll get 1.7 billion requests per day:

*20,000 * 3,600 seconds * 24 hours = 1.7 billion*

To cache 20% of these requests, we'll need 170 GB of memory:

*0.2 * 1.7 billion * 500 bytes = 170 GB*

It's worth noting that there will be a lot of duplicate requests for the same URL. This means our actual memory usage will be less than 170 GB.

System APIs

We can use REST APIs to expose our server's functionality. This example shows the API's definition for creating and deleting URLs without service:

```
createURL(api_dev_key, original_url, custom_alias=None, user_name=None,
 expire_date=None)
```

Parameters

- `api_dev_key` (string): A registered account's API developer key that will be used to throttle users based on their allocated quota
- `original_url` (string): Original URL to be shortened
- `custom_alias` (string): Optional custom key for the URL
- `user_name` (string): Optional username to be used in encoding
- `expire_date` (string): Optional expiration date for the shortened URL
- `returns` (string): A successful insertion returns the shortened URL

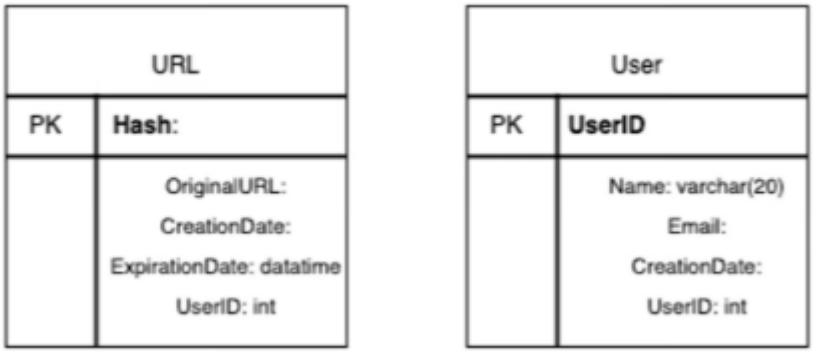
Database design

Let's look at some observations about the data we're storing:

- The service needs to store billions of records
- The service is read-heavy
- Each object is small (<1,000)
- There are no relationships between each record (except for storing which user created the short link)

Schema

We'll now consider database schemas. We need a table for storing information on URL mappings as well as a database for data on users who created short links.



Which database should we use?

The best choice would be a NoSQL database store like DynamoDB or Cassandra since we'll be storing billions of rows with no relationships between the objects.

Basic System Design and algorithms

Our main concern is how we'll generate a short and unique key when given a URL. For this example, we'll do this by encoding the URL. We can compute a given URL's unique hash, such as *MD5* or *SHA256*. The hash can then be encoded for display. This encoding could be *base36*, *base62*, or *base64*. Today, we'll use *base64*.

Now, we'll consider whether the short key should be six, eight, or ten characters long. Using a *base64* encoding, a six-character key would yield *64^6 = ~68.7 billion* possible strings. An eight-character key would result in *64^8 = ~281 trillion* possible strings. Let's assume the six-character key is sufficient.

Our hash function will produce a 128-bit hash value if we use the *MD5* algorithm. Each *base64* character encodes six bits of the hash value. This means we'll get a string of over 21 characters after encoding. We'll need to take a different approach for choosing our key because we only have space for eight characters per short key.

We can take the first six (or eight) letters for the key. We'll need to take steps to ensure we don't encounter key duplication. We might swap some characters or choose characters not already in the encoding string, but there are some potential obstacles:

- If multiple users enter the same URL, they'll get the same short link
- Parts of the URL can be URL-encoded

A workaround for this would be appending a number from a sequence to each short link URL. This would make each link unique, even if multiple users provide the same URL. Another workaround would be appending the user ID to the input URL, but this would only work if the user would sign in, and we'd also need to generate a unique-ness key.

Data partitioning and replication

Our database will store information about billions of URLs. We'll need to partition it to make it scalable. Let's consider two different partitioning approaches: range-based and hash-based.

Range-based partitioning: We can store the URLs in separate partitions based on the first letter of its hash key. We save all the URLs with the first letter of their hash key being `A` in one partition, and so on. This approach can lead to unbalanced database servers, which will create unequal load.

Hash-based partitioning: We can take the hash of a stored object and calculate which partition to use. The hashing function will randomly distribute data into different partitions. This approach sometimes leads to overloaded partitions. If it does, we can use consistent hashing to solve it.

Caching

Our service should be able to cache URLs that are frequently accessed. We could use a solution like Memcached to store full URLs with respective hashes.

Cache memory requirements: We can start with about 20% of the daily traffic and adjust it based on usage patterns. Our previous estimations tell us that we'll need 170 GB of memory to cache 20% of the daily traffic.

Cache eviction policy: We want to replace a link with a more popular URL. We can use the Least Recently Used (LRU) policy for our system.

Load balancing

We can add a load balancing layer to our system in three places:

1. Between clients and application servers
2. Between application servers and database servers
3. Between application servers and cache servers

We can start with the Round Robin approach to equally distribute incoming requests among servers.

Design Instagram

This problem walkthrough is adapted from another article. For a more detailed tutorial, I recommend checking out Design Instagram.

Problem overview

Instagram is a social media platform that allows users to share photos and videos with other users. Instagram users can share information either publicly

or privately. We'll design a simple version of Instagram that enables users to share photos, follow each other, and access the news feed. Users will see the top photos of the people they follow on their feed.

System requirements and goals

Let's consider some functional and non-functional requirements.

Functional requirements:

- The user should be able to search based on photo or video titles
- The user should be able to upload, download, and view photos and videos
- The user should be able to follow other users
- The system should be able to generate a news feed consisting of the top photos and videos of the people that the user follows

Non-functional requirements:

- The service should be highly available
- The system's acceptable latency should be around 200 milliseconds for the news feed
- The system should be reliable so that photos and videos are never lost

> Functions that are *not* within the scope of this project include adding tags to photos, searching for photos with tags, commenting, and tagging users.

Capacity estimations and constraints

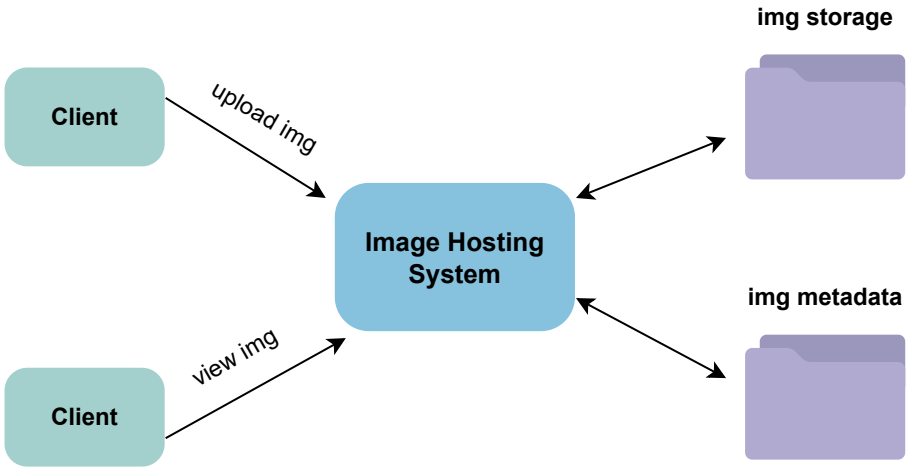
We will base our approach off these numbers:

- 500 million total users
- 1 million daily active users
- 2 million new photos every day (at a rate of 23 new photos/second)
- Average photo file size of 200 KB

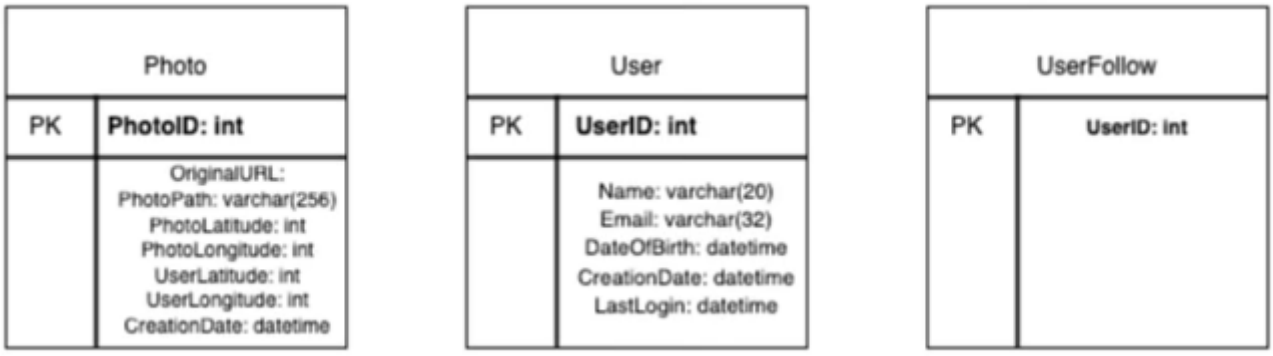
- Total space required for 1 day of photos: 2 million * 200 KB => 400 GB
- Total space required for 10 years: 400 GB * 365 days * 10 years => 1,425 TB

High-level system design

The system should be able to support users as they upload and view each other's media. This means that our service needs servers to store media along with another database server to store the media's metadata.



Database schema



We could take a straightforward approach by storing the above schema in a relational database management system (RDBMS). However, challenges often arise when using a relational database for a scaling application.

We can store the above schema with key-value pairs using a NoSQL database. The metadata for the photos and videos will belong to a table where the `key`

will be the `PhotoID`, and the `value` will be an object containing `PhotoLocation`, `UserLocation`, `CreationTimestamp`, and so on.

We can use a wide-column datastore, Cassandra, to store the relationships between users and photos as well as a list of a user's followed accounts. The `UserPhoto` table's `key` will be the `UserID`. The `value` will be the list of `PhotoIDs` the user owns. These will be stored in a different column. This pattern will be similar to the `UserFollow` table.

The photos and videos can be stored in a distributed file storage like HDFS.

Data size estimation

Let's estimate how much data will go into each table and how much total storage we'll need for 10 years.

User

Each row in this table will be 68 bytes, assuming each `int` and `dateTime` is 4 bytes:

UserID(4 bytes) + Name(20 bytes) + Email(32 bytes) + DateOfBirth(4 bytes) + CreationDate(4 bytes) + LastLogin(4 bytes) = 68 bytes

If we have 500 million users, we'll need 32 GB of total storage:

*500 million * 68 = 32 GB*

Photo

Each row in this table will be 284 bytes:

PhotoID(4 bytes) + UserID(4 bytes) + PhotoPath(256 bytes) + PhotoLatitude(4 bytes) + PhotoLongitude(4 bytes) + UserLatitude(4 bytes) + UserLongitude(4 bytes) + CreationDate(4 bytes) = 284 bytes

If 2 million new photos get uploaded every day, we'll need 0.5 GB of storage for one day:

*2 million * 284 bytes = 0.5 GB/day*

For 10 years, we'll need 1.88 TB of storage.

UserFollow

Each row in this table will be 8 bytes. We have 500 million users. If each user follows an average of 500 other users, we'll need 1.82 TB of storage for the UserFollow table:

*500 million users * 500 followers * 8 bytes = 1.82 TB*

The total space required for all tables for 10 years will be 3.7 TB:

32 GB + 1.88 TB + 1.82 TB = 3.7 TB

Component design

Photo uploads are often slower than reads because they go to the disk. Uploading users will consume all the available connections because of how slow the process is. Reads cannot be served when the system is loaded with write requests. To handle this bottleneck, we can split reads and writes to separate servers so the system isn't overloaded. This allows us to efficiently optimize each operation.

Reliability and redundancy

Creating redundancy in the system allows us to create a backup in the midst of a system failure. We can't lose any files and need a highly reliable application. We can achieve this by storing multiple copies of each photo and video so the system can retrieve media from a copy server in the event that a server dies. We'll apply this design to other components of our architecture. With multiple copies of services in the system, the system will run even if a service dies.

Data sharding

One possible scheme for a metadata sharding service is to partition based on PhotoID. To solve the above problems, we can generate unique PhotoIDs and then find a shard number through `PhotoID % 10`. We wouldn't need to append ShardID with PhotoID in this case because PhotoID will be unique throughout the system.

Generating PhotoIDs

We won't be able to define PhotoID by having an auto-incrementing sequence in each shard. We need to know PhotoID in order to find the shard where it will be stored. One solution could be to dedicate a separate database instance to generate auto-incrementing IDs.

If our PhotoID can fit into 64 bits, we can define a table containing only a 64 bit ID field. Whenever we want to add a photo to our system, we can insert a new row in this table and take that ID as the PhotoID of our new photo.

Ensuring reliability

This key-generating database could be a single point of failure. A workaround for that could be defining two such databases with one generating even-numbered IDs and the other odd-numbered. For MySQL, the following script can define such sequences:

```
KeyGeneratingServer1:
auto-increment-increment = 2
auto-increment-offset = 1
KeyGeneratingServer2:
auto-increment-increment = 2
auto-increment-offset = 2
```

We can put a load balancer in front of both of these databases. We can Round Robin between them to deal with downtime. One of these servers could generate more keys than the other. Luckily, if they are out of sync in this way, it won't cause an issue for our system. We can extend this design by defining separate ID tables for Users, Photo-Comments, or other objects present in our system.

Load balancing

The service would need a large-scale photo delivery system to serve data to users globally. We could push the content closer to users using cache servers that are geographically distributed.

List of common System Design Interview questions

- Design a global chat service like Facebook Messenger or WhatsApp
- Design a social network and message board service like Quora or Reddit
- Design a global file storage and sharing service like Dropbox or Google Drive
- Design a global video streaming service like YouTube or Netflix
- Design an API rate limiter for sites like Firebase or GitHub
- Design a proximity server like Yelp or Nearby Places/Friends
- Design a search engine related service like Type-Ahead
- Design Ticketmaster
- Design Twitter
- Design a Web Crawler

To learn how to solve these System Design Interview problems along with all of the System Design fundamentals, I recommend Scalability & System Design for Developers.

How to approach *any* SDI question

In the Top 10 System Design Interview questions for software engineers, we outline an effective approach to answering any System Design Interview question. Let's take a look at what that approach entails:

- **State what you know/Clarify the goals**: Start each problem by stating what you know. This could include required features of the system, problems you expect to encounter, and what kind of traffic you expect the system to handle. This process shows your interviewer your planning skills, and also allows them to correct any possible misconceptions before you jump into the solution.
- **Describe trade-offs**: Your interviewer will want insight into any system design choice you make. At each decision point, explain at least one positive and one negative effect of your decision.
- **Discuss emerging technologies**: Conclude each question with an overview of how and where the system could benefit from machine learning, for example. This demonstrates that you're not just prepared for current solutions, but future solutions as well.

> To learn how machine learning knowledge can benefit your System Design Interview performance, check out How Machine Learning gives you an edge in System Design.

System Design Interview cheat sheet

Here's a System Design Interview cheat sheet to use for quick reference:

System Design Interview Cheat Sheet

educative

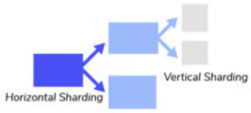
Distributed system fundamentals

Data durability and consistency
The differences and impacts of failure rates of storage solutions and corruption rates in read-write processes

Replication
Backing up data and repeating processes at scale

Consensus
Ensuring all nodes are in agreement, which prevents fault processes from running and ensures consistency and replication of data and processes

Partitioning
Dividing data across different nodes within systems, which reduces reliance on pure replication



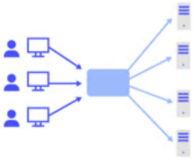
Distributed transactions
Once consensus is reached, transactions from applications need to be committed across databases with fault checks by each resource involved

Architecture of scalable web applications

HTTP
The API on which the entire internet runs

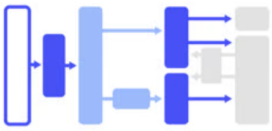
REST
The set of design principles that directly interact with HTTP to enable system efficiency and scalability

DNS and load balancing
Routing client requests to the right servers and the right tiers when processing happens to ensure system stability



Caching
Making tradeoffs and caching decisions to determine what should be stored in a cache, how to direct traffic to a cache, and how to ensure we have the appropriate data in the cache

N-tier applications
Understanding how processing tiers interact with each other and the specific process they control



Stream processing
Applying uniform processes to data streams to allow for efficient use of local resources

How to design large-scale systems

Step 1: Clarify the goals
Make sure you understand the basic requirements and ask any clarifying questions.

Step 2: Determine the scope
Describe the feature set you'll be discussing in the given solution, and define all of the features and their importance to the end goal.

Step 3: Design for the right scale
Determine the scale so you know whether the data can be supported by a single machine or if you need to scale.

Step 4: Start simple, then iterate
Describe the high-level process end-to-end based on your feature set and overall goals. This is a good time to discuss potential bottlenecks.

Step 5: Consider relevant DSA
Determine which fundamental data structures and algorithms will help your system perform efficiently and appropriately.

Step 6: Describe trade-offs
Describe trade-offs while explaining your solution to show you understand large-scale systems and their complexities.

*Ask clarifying questions at each step of the process!

Happy learning!