**Week 4 Assignment:**

**Designing a URL Shortener System:**

**Introduction to URL Shorteners**

Imagine you have a super long web address that’s difficult to share or remember. A URL shortener turns that long link into a much shorter, more manageable one. This is especially useful on platforms like social media, where space is limited.

**Generating Short URLs**

The process behind creating these short URLs often involves hashing, a technique that consistently converts a long URL into a shorter, unique code. This ensures that every time you shorten the same link, you get the same short URL, making it reliable for users.

**Handling Collisions**

However, sometimes two different URLs might end up with the same short code (a collision). To prevent this, the system checks the database to ensure each short URL is unique.

**Database Structure**

The database itself is pretty straightforward. It usually has just a few columns: one for the original long URL, one for the generated short URL, and an ID to keep everything organized.

**Boosting Performance with Caching**

To make everything faster, especially when the same short URLs are being accessed repeatedly, a caching system like Redis is used. This means users can get to their desired pages quicker, and the database doesn’t get overwhelmed.

**Redirect Types**

When you click on a short URL, the server decides how to send you to the original site. There are two main types of redirects: 301 (which is permanent) and 302 (which is temporary). Knowing which one to use helps with tracking users and managing server resources.

**Looking Ahead**

This topic is just the beginning. The discussion hints at moving on to more complex system designs, like how to build a notification system, which will dive deeper into how to structure these systems efficiently.

**Functional Requirements**

The main goal is to create a system that shortens long URLs and then redirects users back to the original link when they click on the shortened version.

**Non-Functional Requirements**

Beyond just working correctly, the system needs to be fast and always available. No one likes waiting, so low latency is key, and high availability ensures users can access the service whenever they need it.

**Short URL Length**

To avoid running out of unique short URLs, the length of these URLs needs to be carefully planned based on how much traffic the service expects. The idea is to make sure there are plenty of unique combinations available.

**Token Service**

To prevent any bottlenecks or failures in generating these URLs, a token service is introduced. This service assigns unique number ranges for creating short URLs, which helps keep the system robust and prevents collisions (where two URLs end up with the same short code).

**Analytics**

Tracking how and where the URLs are being used is important. The system should gather data on URL usage, user location, and device type while still being fast and responsive.

**Data Storage**

Since this system will handle lots of data, it needs a scalable database, like Cassandra, which is designed to manage large volumes of information efficiently.

**Batch Processing**

To make the system even more efficient, especially in handling analytics, data can be processed in batches. This reduces the load on the system and helps keep everything running smoothly.

1. **Understanding Traffic**: Knowing how much traffic to expect helps in designing the system, particularly in deciding how long the short URLs should be to avoid running out of combinations.

2. **Avoiding Collisions**: Using a token service with unique number ranges ensures that there won’t be any conflicts when generating short URLs.

3. **Global Distribution**: Placing data centers in different locations based on where users are coming from can speed up redirects, making the service faster for everyone.

4. **Latency Management**: By processing analytics data asynchronously (in the background), the system remains quick and doesn’t keep users waiting.

5. **Data-Driven Decisions**: Analytics isn’t just for tracking; it also helps in making informed decisions to improve the service and keep users engaged.

6. **Scalability**: The system should be built to grow, with features like database sharding (splitting the database to handle more data) and load balancing (distributing traffic evenly) to ensure it can handle more users in the future.

7. **Flexibility in Design**: The system’s architecture should be adaptable, ready to evolve as the needs of the service and its users change over time.

**Designing a Chat Application: System Design**:

**Overview**

The video dives into how to design a chat application, focusing on the essential features it needs, how to handle large numbers of users, designing the APIs, choosing the right database, and creating a solid overall architecture.

**Key Features**: The video talks about the must-have features for a chat app, like being able to chat one-on-one or in groups.

**Real-Time Communication**: For a chat app to be effective, it needs to deliver messages instantly with minimal delay, making real-time communication a priority.

**Capacity Planning:\*\*** The app is designed with the expectation of 5 million daily users, so it’s important to estimate how much traffic and how many messages the system needs to handle.

**Database Choice:\*\*** The video suggests using NoSQL databases like Cassandra because they can easily scale to handle lots of data.

**API Design**:\*\* APIs are the backbone of how users interact with the app, handling everything from joining a chat group to sending messages.

**High-Level Architecture:\*\*** The architecture includes websocket servers for real-time messaging and notification services to keep users updated.

**User Presence**:\*\* The system also needs to track who is online and optimize performance to ensure smooth user experiences.

**Key Insights**:

**Feature Requirements**: To keep users happy and coming back, the app needs to include features like private chats, group chats, and the ability to see who’s online.

**Real-Time Communication:** Fast, real-time messaging is crucial—users expect their messages to be delivered as quickly as apps like WhatsApp.

**Scalability Needs**: With 5 million users in mind, the app needs a database that can grow horizontally, meaning it can handle more data by adding more servers.

**API Structure**: Well-organized APIs are essential for smooth operations, covering all user actions like joining groups, sending messages, and more.

**High Availability vs. Consistency**: In chat apps, it’s more important for the system to be available (users can always access it), even if it means slight delays, compared to systems where every action must be perfectly consistent immediately.

**Global Distribution**: The app’s architecture should ensure that no matter where users are in the world, they receive their messages quickly by having servers distributed globally.

**Optimizing Message Delivery:** For group chats, using smart algorithms can help deliver messages efficiently, balancing the load on servers while keeping the user experience smooth.

The video breaks down how WhatsApp’s system is designed, focusing on key features like messaging, group chats, and read receipts.

**Group Messaging**: WhatsApp supports group chats with up to 200 users.

**Media Sharing**: Users can send images and videos in their messages.

**Read Receipts:** Messages show when they’re sent, delivered, and read.

**Last Seen Status**: Displays the last time a user was active.

**Temporary Chats**: Some chats are stored temporarily for added privacy.

**WebSockets**: Enables real-time communication between users.

**Load Balancing**: Ensures the system handles large volumes of messages efficiently.

**Key Insights:**

**Decoupling Services**: Splitting up functions into microservices (like session management) makes the system more efficient and scalable.

**Real-Time Communication**: Web Sockets allow instant message delivery, which is crucial for a smooth chat experience.

**Reliable Messaging**: Idempotency ensures messages can be sent multiple times without issues, preventing message loss.

**Privacy vs. Compliance**: Balancing temporary chat storage with data retention needs depends on privacy policies and regulations.

**Feature Prioritization**: During heavy usage, prioritizing core features like messaging over non-essentials (like last seen) ensures performance.

**Load Management**: Strategies like load balancing and message queues keep the system running smoothly under heavy traffic.

**User Activity Tracking**: Accurate tracking of user actions, like last seen, is carefully designed to distinguish real activity from automated app requests.

This video covers how to design a large-scale chat app like WhatsApp, focusing on key components, requirements, and ways to optimize the system.

Importance of System Design: Understanding system design is crucial for interviews, as it shows your ability to break down complex systems.

**Core Requirements**: Identifying essential features like messaging, notifications, and media sharing sets the stage for a solid design.

**Client-Server Architecture**: This model is key for delivering messages efficiently between users and servers.

**Message Flow**: Ensures smooth message delivery and acknowledgment, even when users are offline.

**Push Notifications**: Boost user engagement through methods that balance performance and experience.

**Scalability**: Designing for growth and potential issues like bottlenecks ensures the system remains reliable under heavy traffic.

**Continuous Improvement**: Regularly optimizing features like encryption, monitoring, and user activity can greatly enhance the app’s performance.